

Pedro Manuel Pinto Ribeiro

Efficient and Scalable Algorithms for Network Motifs Discovery



Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
2011

Pedro Manuel Pinto Ribeiro

Efficient and Scalable Algorithms for Network Motifs Discovery



*Tese submetida à Faculdade de Ciências da
Universidade do Porto para obtenção do grau de
Doutor em Ciência de Computadores*

Advisors: Prof. Fernando Silva and Prof. Luís Lopes

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
Junho de 2011

Thesis Committee:

Prof. António Porto, University of Porto (Chair)

Prof. Arlindo Oliveira, IST - Technical University of Lisbon (Examiner)

Prof. Marcus Kaiser, Newcastle University (Examiner)

Prof. José Cardoso e Cunha, New University of Lisbon

Prof. Vítor Santos Costa, University of Porto

Prof. Fernando Silva, University of Porto (Advisor)

Prof. Luís Lopes, University of Porto (Co-Advisor)

To Samuel and Liliana

Acknowledgments

First of all, I would like to thank my two advisors, Fernando Silva and Luís Lopes. They have always been there when I needed them, providing guidance, support and a lot of patience over the last years. Their collaboration was essential and we always had fruitful discussions on the emerging ideas on this subject.

I would like to thank the financial support of FCT that was allowed me to dedicate the initial 4 years pursuing this research (PhD grant SFRH/BD/19753/2004). I would also like to thank CRACS/INESC-Porto L.A. for support on the fifth year, allowing me to further improve my work and finish my PhD.

I would like to thank Paul Watson and particularly Marcus Kaiser for receiving me so well in Newcastle University for 3 months. This stay provided me with some of the initial motivation for entering the motif discovery and complex network analysis world.

I would also like to thank all persons that helped in providing me the computational infrastructures that I used during my work. In particular Hugo Ribeiro, my brother and local computer science department network administrator, and Enrico Pontelli, for the use of Inter Cluster in the New Mexico State University.

I also thank all other researchers from the CRACS research center for providing a very good environment for research and all my office colleagues during these years, which were too many to nominate them all.

Finally, I would like to thank all my friends and family, that have always supported me. In particular, my parents, my parents-in-law, my wife, Liliana, and my son, Samuel, which was born when I was beginning my work.

Abstract

Networks are a powerful representation for a multitude of natural and artificial systems. They are ubiquitous in real-world systems, presenting substantial non-trivial topological features. These are called *complex networks* and have received increasing attention in recent years. In order to understand their design principles, the concept of *network motifs* emerged. These are recurrent over-represented patterns of interconnections, conjectured to have some significance, that can be seen as basic building blocks of networks. Algorithmically, discovering network motifs is a *hard* problem related to graph isomorphism. The needed execution time grows exponentially as the size of networks or motifs increases, thus limiting their applicability. Since motifs are a fundamental concept, increasing the efficiency in its detection can lead to new insights in several areas of knowledge. To develop efficient and scalable algorithms for motifs discovery is precisely the main aim of this thesis.

We provide a thorough survey of existing methods, complete with an associated chronology, taxonomy, algorithmic description and empirical evaluation and comparison. We propose a novel data-structure, *g-tries*, designed to represent a collection of graphs. Akin to a prefix tree, it takes advantage of common substructures to both reduce the memory needed to store the graphs, and to produce a new more efficient sequential algorithm to compute their frequency as subgraphs of another larger graph. We also introduce a sampling methodology for *g-tries* that successfully trades accuracy for faster execution times. We identify opportunities for parallelism in motif discovery, creating an associated taxonomy. We expose the whole motif computation as a tree based search and devise a general methodology for parallel execution with dynamic load balancing, including a novel strategy capable of efficiently stopping and dividing computation on the fly. In particular we provide parallel algorithms for ESU and *g-tries*.

Finally, we extensively evaluate our algorithms on a set of diversified complex networks. We show that we are able to outperform all existing sequential algorithms, and are able to scale our parallel algorithms up to 128 processors almost linearly. By combining the power of *g-tries* and parallelism, we speedup motif discovery by several orders of magnitude, thus effectively pushing the limits in its applicability.

Resumo

As redes são uma poderosa representação para uma grande variedade de sistemas naturais e artificiais, sendo ubíquas em sistemas do mundo real. As *redes complexas* apresentam uma quantidade substancial de características topológicas não triviais e têm recebido uma atenção crescente nos últimos anos. O conceito de *network motifs* (padrões de rede) surgiu para ajudar a perceber como são estas redes desenhadas. Os *motifs* são padrões de ligações frequentes sobre-representados que se conjectura que possam ter algum significado e podem ser vistos como blocos básicos de construção de redes. Em termos algorítmicos, descobrir estes padrões é um problem difícil, relacionado com o isomorfismo de grafos. O tempo de execução necessário cresce à medida que o tamanho das redes e dos padrões aumenta. Como os *motifs* são um conceito fundamental, aumentar a eficiência da sua detecção pode levar a novas descobertas em várias áreas do conhecimento. Desenvolver algoritmos eficientes e escaláveis para a descoberta de *network motifs* é precisamente o principal objectivo desta tese.

Fornecemos um completo estudo do estado da arte das metodologias utilizadas, com a cronologia e taxonomia associadas, uma descrição algorítmica e uma avaliação e comparação empírica. Propomos uma nova estrutura de dados, *g-tries*, desenhada para representar colecções de grafos. Sendo aparentada a uma árvore de prefixos, tira partido de subestruturas comuns para reduzir a memória necessária para guardar os grafos e para criar um novo algoritmo sequencial mais eficiente para calcular as suas frequências como subgrafos de um outro grafo maior. Também introduzimos uma técnica de amostragem para as *g-tries* que troca com sucesso precisão por tempos de execução mais rápidos. Expomos toda a computação de *motifs* como uma pesquisa em árvore e desenhamos uma metodologia geral para a execução paralela com balanceamento dinâmico de carga, incluindo uma nova estratégia capaz de eficientemente parar e dividir a computação durante a sua execução. Em particular, fornecemos algoritmos paralelos para o ESU e para as *g-tries*.

Finalmente, fazemos uma avaliação extensiva dos nossos algoritmos num conjunto de redes diversificadas. Mostramos que temos melhor desempenho que todos os algoritmos sequenciais existentes e que somos capazes de escalar quase linearmente

os nossos algoritmos paralelos até 128 processadores. Combinando o potencial das *g-tries* com o seu paralelismo, conseguimos tempos de execução para descoberta de *motifs* várias ordens de magnitude mais rápidos, efectivamente aumentando os limites da sua aplicabilidade.

Résumé

Les réseaux sont une puissante représentation pour une variété de systèmes naturels et artificiels. Ils sont ubiquité dans les systèmes du monde réel et présentent une quantité substantielle de caractéristiques topologiques qui ne sont pas simples. Ces réseaux sont pour cela appelés réseaux complexes et ont reçu dès quelques années une reconnaissance croissante. Le concept de *network motifs* (patron de réseau) est né de façon à les mieux comprendre. Ce sont des patrons de liaisons fréquentes sur-représentés qui se conjecture pour avoir un sens quelconque et puisse être vu comme des blocs basiques de construction de réseaux. Du point de vue algorithmique, découvrir ces patrons est un problème difficile qui a rapport avec l'isomorphisme de graphes. Le temps d'exécution grandie à la mesure que la grandeur des réseaux et des patrons augmentent. Comme les *motifs* sont un concept essentiel, cela augmente l'efficacité de la détection qui mener à de nouvelles découvertes en plusieurs branches de la connaissance. Le principal objectif de cette thèse est précisément développer des algorithmiques efficaces et extensibles pour la découverte de *network motifs*.

On fait une étude complète de l'état de l'art dans les méthodes utilisées, avec la chronologie et la taxonomie, une description algorithmique et une évaluation et comparaison empiriques. On propose une nouvelle structure de données, *g-tries*, dessinée pour représenté des collections de graphes. Cette structure est similaire à un arbre préfixe et profite de sous-structures communes pour réduire la mémoire nécessaire pour garder les graphes, et pour créer un nouveau algorithme séquentiel plus efficace pour calculer ses fréquences comme sous-graphes d'un autre graphe plus grand. On introduit une technique d'échantillonnage pour les *g-tries* qui échange, avec succès, la précision du calcul pour un meilleur temps d'exécution. On montre que la découverte de *motifs* avec *g-tries* peut être vu comme une recherche en arbre. On dessine une méthode générale pour l'exécution parallèle avec répartition dynamique de charge, on incluse une nouvelle stratégie capable d'efficacement arrêter et repartir le calcul pendant l'exécution. En particulier, on fait des algorithmes parallèles pour ESU et pour les *g-tries*.

Finalement, on fait une évaluation extensive de nos algorithmes dans un groupe de plusieurs réseaux. On montre que notre algorithme a une meilleure performance

que tous les autres algorithmes séquentiels existants et que on est capable d'avoir une extensibilité presque linéaire jusqu'à 128 processeurs. Mélangeant *g-tries* avec son parallélisme, on réussit à avoir des temps d'exécution de plusieurs ordres de grandeur plus rapide pour la découverte de *motifs*. Avec cela, on réussit effectivement à augmenter les limites de son applicabilité.

Contents

Abstract	7
Resumo	9
Résumé	11
List of Tables	17
List of Figures	19
List of Algorithms	23
1 Introduction	25
1.1 Motivation	28
1.2 Goals and Contributions	31
1.3 Thesis Organization	33
1.4 Bibliographic Note	34
2 The Network Motifs Problem	35
2.1 Graph Terminology and Concepts	35
2.2 Network Motifs Problem	38
2.2.1 The Original Definition	38
2.2.2 Variations in the Definition	42
2.3 Applications of Network Motifs	48
2.4 Criticism	53
2.5 Summary	54
3 Algorithms for Motif Discovery	55
3.1 Algorithmic Approaches	55

3.1.1	The Motif Discovery Program Flow	56
3.1.2	Historical Overview	57
3.1.3	Comparison of Existing Sequential Algorithms	59
3.2	Sequential Exact Census	61
3.2.1	Original Algorithm - mfinder	61
3.2.2	ESU Algorithm	63
3.2.3	Grochow and Kellys' Algorithm	64
3.2.4	Kavosh Algorithm	67
3.2.5	MODA Algorithm	70
3.3	Sequential Approximation Census	72
3.3.1	Sampling Subgraphs - Kashtan algorithm	72
3.3.2	Rand-ESU Algorithm	73
3.3.3	MODA Sampling	74
3.4	Determining the Significance	75
3.5	Parallel Algorithms	76
3.5.1	Wang et al.	76
3.5.2	Schatz et al.	77
3.6	Summary	77
4	The G-Trie Data Structure	79
4.1	Motivation and Prefix Trees	79
4.2	G-Tries Definition	81
4.3	A G-Trie Implementation	83
4.4	Creating a G-Trie	85
4.4.1	Iterative Insertion	85
4.4.2	Canonical Representation of Graphs	86
4.4.2.1	The Need for a Canonical Form	86
4.4.2.2	Impact on the G-Trie Structure and Compression Ratio	86
4.4.2.3	An Efficient Custom Built Canonical Form	88
4.4.3	Insertion Algorithm	91
4.4.4	Reusing G-Tries	92
4.5	Computing subgraph frequencies	92
4.5.1	An Initial Approach	92
4.5.2	Breaking Symmetries	95

4.5.2.1	Creating a Set of Symmetry Breaking Conditions . . .	95
4.5.2.2	Using the Conditions to Constrain the Search	97
4.5.2.3	Reducing the Number of Conditions	100
4.6	Sampling Subgraphs	103
4.6.1	Uniform Sampling	103
4.6.2	Sampling Parameters	105
4.7	Motif Discovery with G-Tries	106
4.8	Summary	107
5	Parallel Network Motif Discovery	109
5.1	Opportunities for Parallelism	109
5.2	Motif Discovery as a Tree Shaped Computation	111
5.2.1	The Search Tree of ESU Census	113
5.2.2	The Search Tree of G-Trie Census	114
5.3	A General Parallel Approach	114
5.3.1	Terminology	115
5.3.2	Work Units	115
5.3.3	Parallel Strategies	117
5.3.4	Pre-Processing Phase	120
5.3.5	Work Phase	122
5.3.5.1	Master-Worker	122
5.3.5.2	Distributed Queues	124
5.3.5.3	Distributed Snapshot	127
5.3.6	Aggregation Phase	136
5.3.7	Parallel Sampling	138
5.3.8	Motif Discovery	138
5.4	Summary	139
6	Experimental Evaluation	141
6.1	Common Materials	141
6.1.1	Computational Environment	141
6.1.2	Complex Networks	142
6.1.3	Other Competing Algorithms	145
6.2	Sequential Algorithms	146

6.2.1	G-Tries Creation	147
6.2.2	G-Tries Census	150
6.2.2.1	Effect of Canonical Labeling	150
6.2.2.2	Effect of Symmetry Breaking Conditions	152
6.2.2.3	Asymptotic Behavior	153
6.2.3	G-Tries Comparison with Other Algorithms	155
6.2.4	G-Tries Sampling	166
6.3	Parallel Algorithms	169
6.3.1	Parameters Choice	169
6.3.2	Parallel Strategies for Pre-Processing and Aggregation	170
6.3.3	Load Balancing and Scalability	172
6.4	Summary	176
7	Conclusions and Future Work	179
7.1	Summary	179
7.2	Main contributions	181
7.3	Future Work	182
7.3.1	Overall Contributions to the Community	182
7.3.2	Efficiency and Flexibility of the Developed Methods	183
7.4	Final remarks	184
A	G-Tries Anatomy	185
B	G-Tries Implementation	189
	References	192

List of Tables

2.1	Different frequency concepts and respective matches.	44
2.2	Network motifs found in different types of networks.	49
3.1	Number of possible directed and undirected subgraphs.	57
3.2	Classification and comparison of sequential algorithmic strategies. . . .	59
5.1	Strategies for different phases of a parallel job	120
6.1	Topological features of the networks used to test the algorithms.	144
6.2	Execution time of our ESU and Kavosh implementations.	146
6.3	Execution time for inserting all undirected k -graphs in a g-trie.	147
6.4	Execution time for inserting all directed k -graphs in a g-trie.	148
6.5	Execution time for reading a g-trie with all undirected k -graphs.	149
6.6	Execution time for reading a g-trie with all directed k -graphs.	149
6.7	Memory needed for a g-trie containing all undirected k -graphs.	150
6.8	Memory needed for a g-trie containing all directed k -graphs.	150
6.9	Effect of canonical labelings on k -census computation.	151
6.10	Effect of symmetry breaking conditions on k -census computation.	153
6.11	Subgraph discovery ratio for 9-census on undirected networks.	155
6.12	Subgraph discovery ratio for 5-census on directed networks.	155
6.13	Comparison of g-tries with other algorithms when doing a full k -census on original undirected networks.	157
6.14	Comparison of g-tries with other algorithms when doing a full k -census on original directed networks.	158
6.15	Comparison of g-tries with other algorithms when computing a k -census in the similar undirected randomized networks.	162
6.16	Comparison of g-tries with other algorithms when computing a k -census in the similar directed randomized networks.	163
6.17	Average speedup of g-tries against competing algorithms.	165
6.18	First motif sizes k for which a sequential program takes more than 1 hour.	173

6.19	Parallel performance of motif discovery with g-tries.	175
6.20	Parallel performance with 128 processors as the number of random networks change.	176
B.1	Main C++ classes used in our implementation.	190

List of Figures

1.1	The neural network of <i>C. elegans</i>	26
1.2	An example pattern of connections in a social network.	27
1.3	Three different models of networks.	29
1.4	An example network motif of size 3.	30
2.1	A directed graph and its correspondent adjacency matrix.	36
2.2	The concept of an induced subgraph.	36
2.3	Four isomorphic undirected graphs of size 6.	37
2.4	Three undirected graphs and their respective automorphisms.	37
2.5	Two possible matchings of a graph H in another graph G	38
2.6	Example of similar random networks preserving degree sequence.	39
2.7	Example of frequency count.	40
2.8	Counting the frequency of colored subgraphs.	42
2.9	Two graphs and all the possible matchings of one into the other.	44
2.10	Non-induced subgraph matches.	45
2.11	Triad significance profiles.	49
2.12	Polar plots of motif fingerprints in Macaque Visual Cortex.	50
2.13	Clustering the Macaque Visual Cortex with motif fingerprints.	50
2.14	Three different types of motifs on the regulation network of <i>E. coli</i>	51
2.15	Using motifs as building blocks of the actual plotting of a network.	52
2.16	Example of a motif cluster.	52
2.17	Example motif generalizations of the feed forward loop.	53
3.1	The 13 different classes of isomorphic subgraphs of size 3.	57
3.2	Time line for motif discovery algorithms.	58
3.3	Search tree of mfinder algorithm looking for 3-subgraphs.	63
3.4	Search tree of ESU algorithm looking for 3-subgraphs.	64
3.5	Finding symmetry conditions on a graph with 6 vertices.	66
3.6	Search tree of Grochow algorithm looking for a 3-subgraph.	67
3.7	Kavosh combinatorial search tree.	68
3.8	The expansion tree of 4-graphs.	70

4.1	A trie representing a set of four words.	81
4.2	Common substructures in graphs.	81
4.3	A g-trie representing a set of 6 undirected graphs.	82
4.4	A g-trie representing a set of 4 directed graphs.	83
4.5	An implementation of g-tries using adjacency matrices.	84
4.6	Sequential insertion of 3 graphs on an initially empty g-trie.	85
4.7	Three different adjacency matrices representing the same graph.	86
4.8	Two different g-tries using lexicographically larger and smaller adjacency strings.	87
4.9	A g-trie containing 11 undirected 5-subgraphs.	90
4.10	An example of a partial program flow of the recursive g-trie <code>match()</code> procedure.	94
4.11	Symmetry breaking conditions for an example 4-graph.	95
4.12	Symmetry conditions computed for all undirected 4-subgraphs.	97
4.13	Filtering symmetry conditions: step #1.	101
4.14	Filtering symmetry conditions: step #2.	102
4.15	Filtering symmetry conditions: step #3.	102
4.16	An example g-trie matching search tree.	103
4.17	Associating a probability with each search tree depth.	105
5.1	Motif discovery algorithmic flow.	110
5.2	Motif discovery as a tree shaped computation.	112
5.3	Search tree of ESU algorithm (revisited).	113
5.4	An example g-trie matching search tree (revisited).	114
5.5	Relative execution time of work units applying ESU to a social network.	119
5.6	Relative weight of work units applying g-tries to a social network.	119
5.7	Master-Worker load balancing strategy.	122
5.8	Distributed load balancing strategy.	125
5.9	Diagonal work-queue splitting scheme example.	127
5.10	An example expansion of a work queue using ESU algorithm.	128
5.11	G-Trie recursive procedure frozen at a given time.	130
5.12	An array structure representing a g-trie snapshot.	131
5.13	ESU recursive procedure frozen at a given time.	134
5.14	An array structure representing an ESU snapshot.	134
5.15	7 processors organized in binary search tree.	137
6.1	Execution time for a census on a social network as the number of nodes increases.	154

6.2	Subgraph discovery ratio for a social network as the number of nodes increases.	154
6.3	Subgraph discovery ratio for power and metabolic as subgraph size increases.	156
6.4	G-Tries speedup over other algorithms on original network census. . . .	159
6.5	G-Tries speedup over other algorithms on the randomized set of networks.	164
6.6	Subgraph discovery ratio with RAND-GTRIE and RAND-ESU	167
6.7	Accuracy of g-tries sampling of 5-subgraphs.	168
6.8	Time spent in g-tries sampling of 5-subgraphs.	168
6.9	Communication between 128 processors with all_in_one pre-processing phase.	171
6.10	Communication between 128 processors with static_partition pre-processing phase.	171
6.11	Aggregation times for all strategies as the number of communicated frequencies increases.	172
A.1	A g-trie containing all 2 undirected 3-subgraphs.	186
A.2	A g-trie containing all 6 undirected 4-subgraphs.	186
A.3	A g-trie containing all 21 undirected 5-subgraphs.	187
A.4	A g-trie containing the 33 undirected 6-subgraphs found in circuit . .	187
A.5	A g-trie containing all 13 directed 3-subgraphs.	188
A.6	A g-trie containing the 24 directed 4-subgraphs found in metabolic . . .	188
B.1	Interactions between the C++ classes of our implementation.	191

List of Algorithms

3.1	Typical program flow of motif discovery.	56
3.2	mfinder enumeration of subgraphs.	62
3.3	ESU enumeration of subgraphs.	63
3.4	Grochow enumeration of an individual subgraph.	65
3.5	Kavosh enumeration of subgraphs.	69
3.6	MODA enumeration of undirected subgraphs.	71
3.7	Kashtan method for sampling one subgraph	72
3.8	Rand-ESU sampling of subgraphs	73
3.9	MODA sampling version of mapping procedure	74
3.10	Markov-Chain for creating similar randomized graphs	75
3.11	Direct method for creating similar randomized graphs	76
4.1	Converting a graph to a canonical form	89
4.2	Inserting a graph G in a g-trie T	91
4.3	Census of subgraphs of T in graph G	93
4.4	Symmetry breaking conditions for graph G	96
4.5	Inserting a graph G in a g-trie T [with symmetry breaking]	98
4.6	Census of subgraphs of T in graph G [with symmetry breaking]	99
4.7	Sample subgraphs of g-trie T in graph G	104
5.1	Expanding or computing a Work Unit (main part)	117
5.2	Expanding or computing a Work Unit (ESU part)	117
5.3	Expanding or computing a Work Unit (g-tries part)	118
5.4	Static division of work queue using round-robin scheme	121
5.5	Master procedure for master-worker load balancing	123
5.6	Worker procedure for master-worker load balancing	124
5.7	Distributed queue main worker procedure.	125
5.8	Distributed snapshot main worker procedure.	129
5.9	Distributed snapshot version of g-trie recursive procedure.	131
5.10	Resuming a g-trie snapshot.	132
5.11	Distributed snapshot version of ESU enumeration of subgraphs	135
5.12	Resuming an ESU snapshot.	135

*Awaken people's curiosity. It is enough to
open minds, do not overload them. Put
there just a spark.*

Anatole France

1

Introduction

While reading this thesis, the around 100 billion neurons in the reader's brain are working together to perform all the necessary cognitive functions, in order too see, read and understand the meaning of this text [oSA99]. Each of these neurons, by itself, is a relatively simple entity. However, as a whole, our neurons form an intricate network of interactions that produce a really complex system, and we still are very far from truly understanding it. Their collective behavior is much more than the simple sum of the individual parts. As Craig Reynolds wisely expressed, “a flock is not a big bird” [Ros06]. This is truth for a very wide variety of natural and artificial systems, which are not only defined by their individual entities but also by the set of complex interactions that occur between them.

Networks (or *graphs*) arise as simple yet powerful abstract models for trying to characterize these types of structures. In the same way a map describes the geography of a certain region, a network “maps” the set of entities and the relationships between them.

Networks are virtually everywhere. The Internet is a network of computers and the connections between them. The World Wide Web is a network of web pages and the links between them. An ecosystem is a network of life forms and the relationships between them. A transportation system can be seen as the actual physical location nodes and their connections. Our own existence can be seen as being part of a complex and intricate web of social relationships of many different kinds.

CHAPTER 1. INTRODUCTION

Most of these real-world existing networks appear to have substantial non-trivial topological features, with interconnections that are neither purely random nor purely regular. These are what we call *complex networks* [AB02, DM02, New03, BLM⁺06] and they recently received increased attention by the research community.

Figure 1.1 gives a graphical visualization of a real complex network, depicting the neural network of the *C. elegans*, a very small nematode (roundworm) which has been extensively used as a model organism for understanding biological networks.

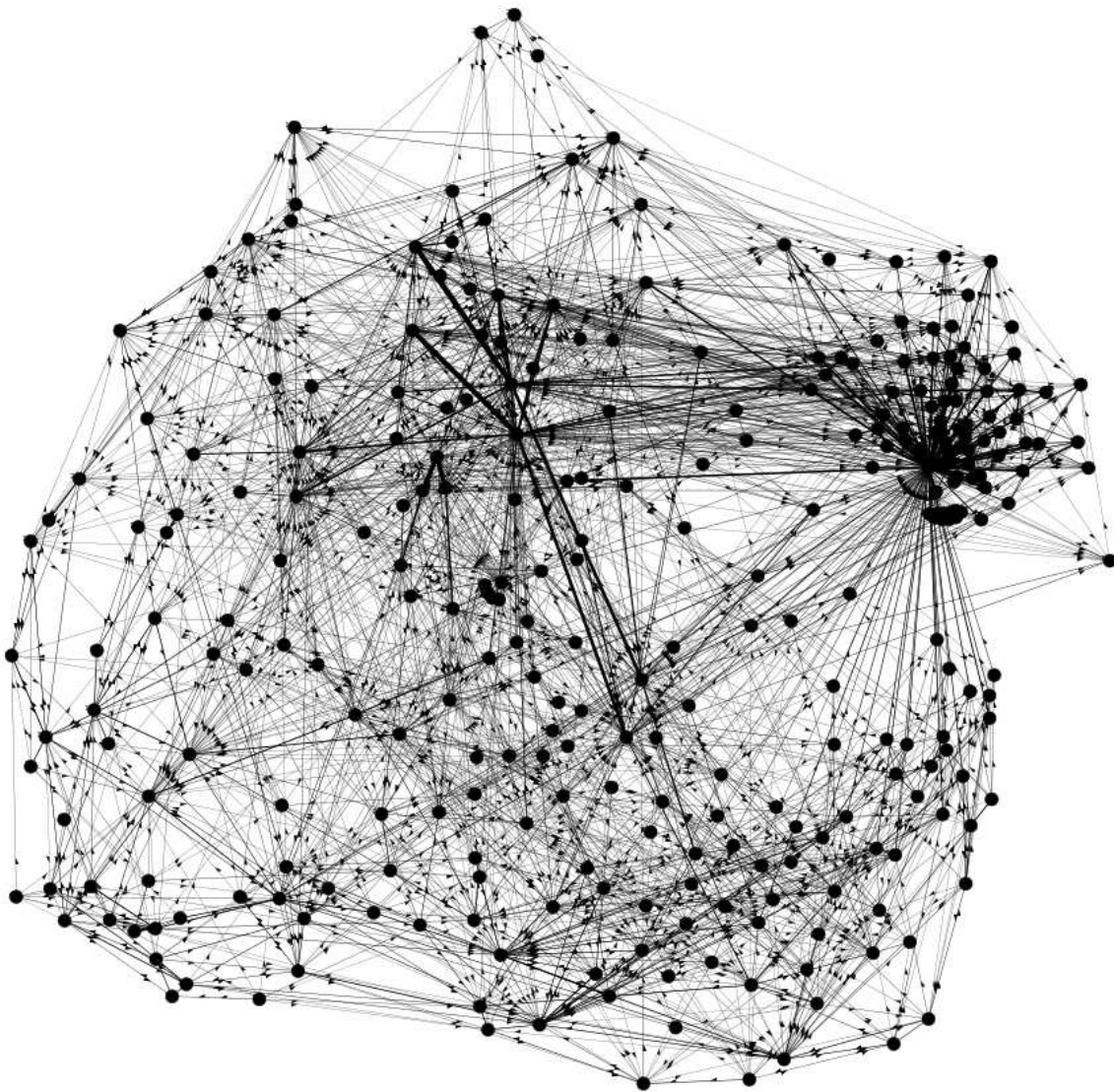


Figure 1.1 – The neural network of *C. elegans*. Drawn with Gephi [BHJ] with data from [WS98].

One way to analyze a complex network is to use a bottom-up approach, trying to

understand first small topological substructures, and how they fit in the global overall behavior. In an analogy with LEGO toys, we want to first identify the different bricks that form the basic building blocks of the networks. We want to discover patterns of interconnections and understand why they exist and what is their meaning.

Figure 1.2 shows an example of a pattern of connections between 3 nodes that appears repeatedly in a real complex network of friendships between members of a karate club. The pattern is represented by thick edges and represents a set of three friends that know each other. For clarity, the figure shows only some of the occurrences of this pattern.

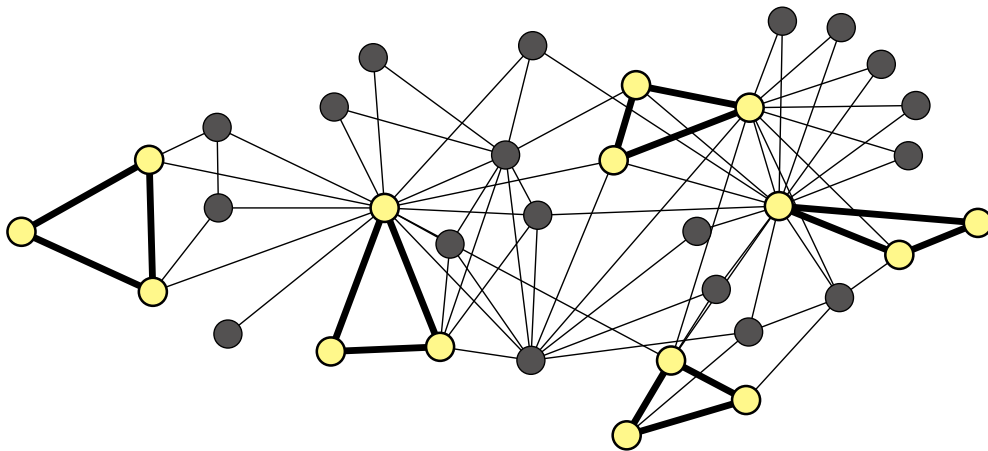


Figure 1.2 – An example pattern of connections in a social network. Edges represent friendship between members of a karate club. It was drawn with help from Gephi [BHJ] and data from [Zac77].

The term *network motifs* appeared first in 2002 [MSOI⁺02], referring to recurring subnetworks conjectured to have some significance. In particular, they are subnetworks that appear with a higher frequency than it would be expected in similar random networks (a more concise and formal definition can be seen in Section 2.2). This is a fundamental concept that has been used as a very useful tool to uncover structural design principles in networks from many different fields.

From a computer science point of view, discovering network motifs is a challenge. Algorithmically, this is a *hard* problem, fundamentally connected to the *graph isomorphism* problem, for which no general polynomial time algorithm is known [McK81]. As the size of the motifs increases, the time needed to compute them with current methodology grows exponentially. This effectively limits applicability to very small sizes in order to obtain results in a reasonable amount of time.

CHAPTER 1. INTRODUCTION

Being able to discover network motifs more efficiently would enable us to enlarge these limits. This would allow practitioners of several scientific fields to discover new angles in which to look at the networks of their respective areas. Pushing the limits in which motif discovery is feasible, both in execution time and motif and network size, is the main aim of this thesis.

1.1 Motivation

Studying and characterizing networks is an inherently interdisciplinary subject with a potential to impact several areas. Work in this area was historically started by Euler in 1735 with the famous problem of the Bridges of Königsberg¹. This laid out the foundations for *graph theory*, or the study of graphs. For more than two centuries graph theory was essentially a mathematical subject, with a major milestone in the *random graphs* concept, defined by Erdős and Rényi [ER59] in the late fifties. Many fundamental questions were asked, but the absence of reliable large empirical experimental data was a major hurdle.

It was only in the late nineties that complex network research really took off. The emergence of many reliable data sets of real networks, together with the maturation of the Internet as a powerful tool, the ever increasing computing power, and the breaking down of barriers between scientific disciplines, all helped in giving the ingredients for breakthroughs. It was now possible to empirically verify that real-world networks exhibited properties that distinguished them from simple random networks. Two well known classes of complex networks emerged as the canonical case studies: *small world* networks [WS98], with short average path lengths and high clustering; and *scale free* networks [BA99], in which the degree distribution (the number of connections of each node) follows a power law distribution.

Figure 1.3 illustrates the original network models formed to present these characteristics and compares them to the original random network model. All sub-figures have 24 nodes. Sub-figure (a), the random network, was generated using the model of Erdős and Rényi [ER59], with 60 connections. Sub-figure (b), the small world network, has all nodes connected to the first and second nearest neighbor in a ring, with 4 rewired connections (probability of rewiring $p = 0.08$). In sub-figure (c), the scale free network, the probability of a node having degree k is $P(k) \propto k^{-2.2}$.

¹Four different landmasses in the city of Königsberg were connected by seven bridges and the problem was to find out if there existed a path that would cross each bridge once and only once.

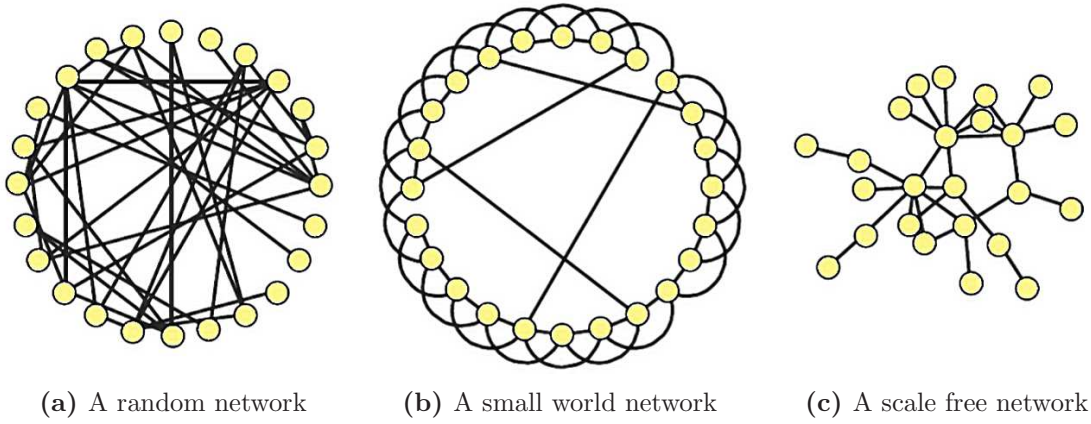


Figure 1.3 – Three different models of networks. Adapted from [Ros06].

Applications of the concepts, models and techniques developed have spawn in many different disciplines like sociology, biology, physics, mathematics and computer science [dFCOT⁺07]. Scientists have now available a very large apparatus of different measurements to mine interesting features from complex networks [ZKMW05, CF06, dFCRTB07].

From the beginning, the typical properties studied were either related to individual nodes of the network, like its degree (number of connections) or centrality (relative importance); or to the global features of the network, like the average inter-node distance or the global degree distribution. In order to go beyond these features one would need to bridge that gap by looking for something in-between, bigger than a single node, but smaller than the whole graph.

In 2002, Milo et al. [MSOI⁺02] were trying to discover basic structural elements that were particular to a determined class of networks. In order to do that they noted that some subnetworks appeared with a much higher frequency in the studied networks than it would be expected in similar randomized networks (they used random networks with the same degree sequence). They called *network motifs* to these over-represented topological patterns, introducing a new concept in the network science field, and presented it as a basic building block of complex networks.

Figure 1.4 exemplifies the concept of network motifs. The three random networks present the exact same degree sequence as the original one, with each vertex preserving its ingoing and outgoing degrees. The feed forward loop, indicated by thick black edges, appears exactly three times in the original network ($\{0, 1, 2\}$, $\{2, 3, 5\}$, $\{3, 4, 5\}$), but has at most one occurrence in each random network. Note that in the third

CHAPTER 1. INTRODUCTION

random network, $\{2, 3, 5\}$ is not an occurrence of the motif, since 2 and 3 have a mutual connection, which does not happen in the feed forward loop.

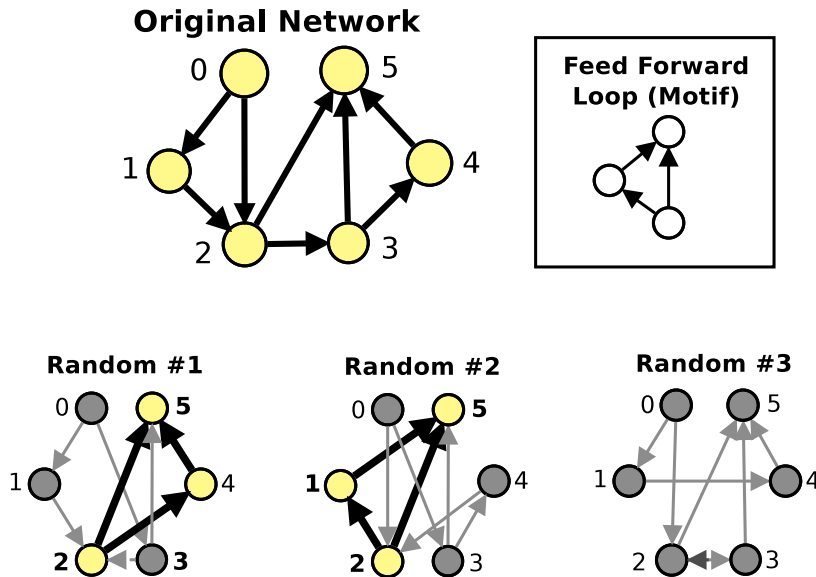


Figure 1.4 – An example network motif of size 3.

The seminal paper from Milo et al. gave origin to a multitude of definitions and studies. Network motifs have since been extensively used in various areas. This is particularly true for biological networks [AA03]. For example, it has been demonstrated that they can have functional significance in transcriptional regulatory networks [SOMMA02] or protein-protein interaction networks [AA04]. They have been applied in other biological areas, like brain networks [SK04] or food webs [Kon08]. They are also significant in networks from other domains, like electronic circuits [ILK⁺05] or software architecture [VS05].

Finding network motifs is however a computationally *hard task*, closely related to the *graph isomorphism* problem [McK81]. Current methods are almost all sequential in nature and consist in finding the frequency of all subgraphs of a determined size (that is, doing a *subgraph census*), both in the original network and in a random ensemble of similar networks. The execution time increases exponentially when we increase the motif or network size. Sampling has been introduced by Kashtan et al. [KIMA04a] as a way to trade accuracy for time spent, but the process can still be very time consuming. Analytical methods to estimate the significance of motifs are now appearing [MSB⁺06, PDK⁺08]. These methods would be able to escape the need for the ensemble of random networks and their respective census, but there is still a long path to be accomplished in order for them to be general and practical enough

1.2. GOALS AND CONTRIBUTIONS

to be used. In any case, we would still have to compute the subgraph census on the original network.

Scientists that are using motifs to characterize their networks typically use very small motif sizes for practical reasons: they want to obtain results in a reasonable amount of time. They would be glad to extend their analysis for larger motifs and larger networks if that would remain feasible for them. Indeed, even increasing the size of computable motifs by only one node can potentially lead to the discovery of a new structural motif that can have an important functional meaning. In a way, they are acting as paleontologists trying to uncover functionality by analyzing the skeleton, the backbone of the system. Increasing the scale in which the analysis is available can lead to a breakthrough. For instance, little is still known about large motifs and their relationship with smaller ones. Any contribution that can shed a light in this area has the potential to impact several fields.

1.2 Goals and Contributions

The main goal of this thesis is *to improve the efficiency of network motif detection*. This means much more than simply being able to speed up the discovery of the same kind of motifs already known. This work aims to contribute to an increased size of the networks which are feasible for computation in a reasonable time. It also aims to allow the discovery of larger motifs. Being able to do both these two things means that we pave the way for the discovery of new motifs that can uncover potential new functional substructures that were previously unknown.

Our approach is twofold. First, we aim to improve the sequential algorithms by exploring a specialized approach that relies on being able to use *sets of subgraphs* as the basic input data. By contrast, current approaches either need to analyze the global network and discover all subgraphs of a determined size, or they count the frequency of individual subgraphs. Our proposed strategy falls in-between. Second, it aims to exploit parallelism in the motif discovery process. The use of parallel algorithms for this purpose is still very scarce, although it has the potential to give large gains in the execution time.

The first contribution of this work is a thorough survey of the state of the art in the motif discovery field. A formal definition of the problem at hand is given, as well as the creation of an associated taxonomy, a time line and a quick overview table that

CHAPTER 1. INTRODUCTION

highlights the differences between current strategies. Pseudo-code is also given for all main strategies, allowing the reader to have a real grasp of the algorithmic inner works of the most important approaches.

A second major contribution of this work is a novel data-structure called *g-tries*. It is specially designed to deal with collections of subgraphs and its main conceptual idea is akin to a prefix tree, in the sense that we take advantage of common topologies by constructing a multiway tree where the descendants of a node share a common substructure. Pseudo-code is given for creating a g-trie, for using it to count sets of subgraphs at the same time and its role in motif discovery is defined. We also propose a sampling strategy for g-tries, allowing for approximate discovery, in which one can trade accuracy for quicker results. Compared to the previously best known sequential methods, g-tries lead up to significant performance gains, at least an order of magnitude faster.

A third major contribution is the identification of opportunities for parallelism in the motif discovery process. We propose a master-worker and a distributed control parallel strategy, providing dynamic load balancing capabilities. The previous best general algorithm and our g-tries approach are parallelized in this way, and pseudo-code with all algorithmic details is given. We obtain almost linear speedups up to 128 processors, showcasing the efficiency of our proposed parallel approach.

The fourth major contribution of this work is an extensive experimental evaluation of our proposed strategies, in order to further verify our claim that indeed we are improving the efficiency of motif discovery. The algorithms are implemented and tested against a large set of representative complex networks from several scientific domains.

Ultimately, by combining the power of the performance gains by using the g-tries and by using parallelism we can speed up the network motifs computation by several orders of magnitude. The possibility of deciding a trade-off between accuracy and execution time adds an extra touch of flexibility and usability of our methods. All of this work has a direct impact on the kind of motifs that are discoverable in a reasonable amount of time and effectively pushes the boundaries in the network motifs problem.

1.3 Thesis Organization

This thesis is structured into seven major chapters. A brief description of each one of them is now provided.

Chapter 1 - Introduction. Provides a brief introduction to the research area, the main motivation, goals and contributions, as well as the organization of the thesis and a bibliographic note.

Chapter 2 - The Network Motifs Problem. Introduces a common graph terminology that will be used throughout the thesis and uses it to formalize the problem being tackled. It also gives variations on the concept, application examples and possible criticisms.

Chapter 3 - Algorithms for Motif Discovery. Gives a detailed depiction of the state of the art, complete with a timeline, an overview table and pseudo-code for all known main algorithmic strategies for discovering network motifs, both sequential, parallel and also complete or approximate approaches.

Chapter 4 - The G-Trie Data Structure. Introduces the novel specialized g-trie data-structure, by giving the motivation behind it and a formal definition. Explains and gives pseudo-code for its associated methods, namely its creation, subgraph counting, subgraph sampling and motif discovery.

Chapter 5 - Parallel Network Motif Discovery. Describes the opportunities for parallelization on network motifs discovery and proposes different parallel strategies with different underlying data-structures and sequential algorithms, focusing mainly on g-tries. It details each step of the parallel strategy, explaining how the computation is started, how dynamic load balancing is achieved and how the results are aggregated.

Chapter 6 - Experimental Evaluation. Gives a thorough experimental evaluation, assessing the efficiency of the proposed g-tries sequential and parallel strategies on a large set of representative complex networks.

Chapter 7 - Conclusions and Future Work. Discusses the research done, summarizing the contributions made and gives directions for future work.

1.4 Bibliographic Note

Parts of the work of this thesis have already been published in international conferences, workshops and journals. A list of those is given next:

- A state of the art survey on network motifs discovery, with a precise formalization of the problem, pseudo-code description of the main existing sequential algorithms, their implementation and runtime comparison on a series of benchmark networks was published in e-Science'2009 [RSK09].
- The novel g-trie data structure, its formal description, creation and matching algorithms, their implementation and empirical evaluation on a series of benchmark networks was the subject of a paper published in ACM-SAC'2010 [RS10b].
- The sampling algorithm for subgraph frequency estimation based on g-tries and its implementation, along with empirical evaluation, was described in a paper published in WABI'2010 [RS10a].
- A parallel version of the g-tries matching algorithm for counting subgraphs, obtaining almost linear speedup in a series of networks, was described in Cluster'2010 [RSL10a].
- A scalable parallel master-worker algorithm for counting subgraphs based on the sequential ESU algorithm [Wer06] was presented in Bioinformatics'2010 [RSL10b], and a revised version was selected to appear in the post-conference proceedings published in the Springer CCIS series [RSL11].
- A thorough analysis of opportunities for parallelization in motif discovery, along with master-worker and distributed control strategies for parallelizing the sequential ESU algorithm [Wer06] and their empirical analysis on a set of benchmark networks was the subject of an article in the Journal of Parallel and Distributed Computing, published by Elsevier [RSLar].

Facts are the air of scientists.

Without them you can never fly.

Linus Pauling

2

The Network Motifs Problem

The main goal of this work is to improve the efficiency of network motif detection and we begin by clearly defining a network motif and its usefulness. The purpose of this chapter is therefore to introduce the reader to the motif discovery problem and give examples of its applicability in real world problems. It is the basis for understanding the remaining chapters.

2.1 Graph Terminology and Concepts

In order to establish a well defined and coherent network terminology throughout this thesis, this section reviews the main concepts used. A network is modeled with the mathematical object *graph*, and we will use these two terms interchangeably.

A *graph* G is composed of a set $V(G)$ of *vertices* or *nodes* and a set $E(G)$ of *edges* or *connections*. The *size* of a graph is the number of vertices and is written as $|V(G)|$. A k -graph is a graph of size k . Every edge is composed of a pair (u, v) of two *endpoints* in the set of vertices. If the graph is *directed*, the order of the pair expresses direction, while in *undirected* graphs there is no direction in edges.

The *degree* of a node is the number of connections it has to other nodes. In directed nodes, we can also define the **indegree** of a node as the number of ingoing connections to it, that is, the number of edges (u, v) in which v is that node. In similar fashion, we can define the **outdegree** of a node as the number of outgoing connections it has.

CHAPTER 2. THE NETWORK MOTIFS PROBLEM

A graph is classified as *simple* if it does not contain multiple edges (two or more edges connecting the exact same pair of nodes) and it does not contain self-loops (an edge connecting a node to itself). Unless otherwise explicitly stated, from now on we will assume we are only dealing with simple graphs.

The *neighbourhood* of a vertex $u \in V(G)$, denoted as $N(u)$, is composed by the set of vertices $v \in V(G)$ such that v and u share an edge. In the context of this thesis, all vertices are assigned consecutive integer numbers starting from 0. The comparison $v < u$ means that the index of v is lower than that of u . The adjacency matrix of a graph G is denoted as G_{Adj} , and $G_{Adj}[u][v]$ is 1 when $(u, v) \in E(G)$ and is 0 otherwise. This concept is illustrated in Figure 2.1.

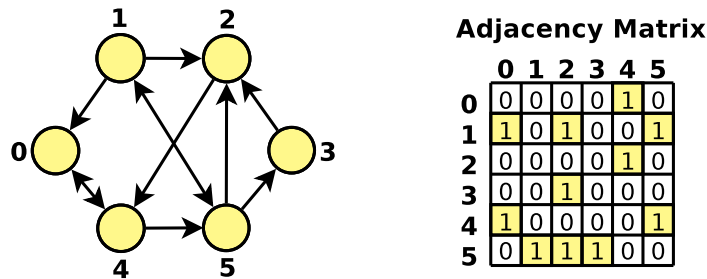


Figure 2.1 – A directed graph and its correspondent adjacency matrix.

A *subgraph* G_k of a graph G is a graph of size k in which $V(G_k) \subseteq V(G)$ and $E(G_k) \subseteq E(G)$. This subgraph is said to be *induced* if $\forall (u, v) \in E(G_k)$ if and only if $(u, v) \in E(G)$. The neighborhood of a subgraph G_k , denoted by $N(G_k)$ is the union of $N(u)$, $\forall u \in V(G_k)$.

Figure 2.2 shows the difference between an induced and a non-induced subgraph. The subgraphs are identified by the light vertices and thick black edges. (a) is not induced because the edge $(4, 5)$ does not belong to it.

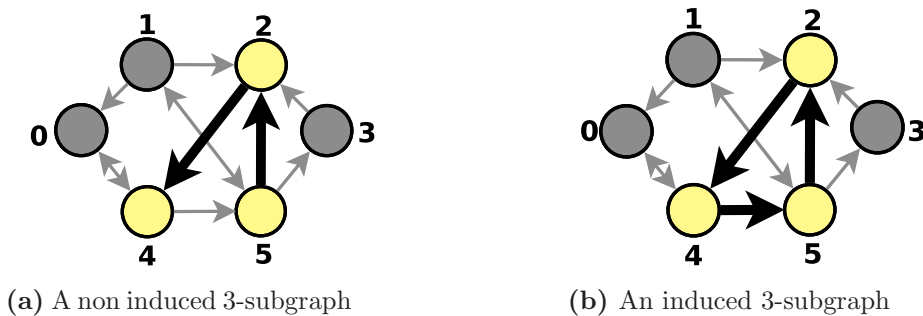


Figure 2.2 – The concept of an induced subgraph.

A *mapping* of a graph is a bijection where each vertex is assigned a value. In the

2.1. GRAPH TERMINOLOGY AND CONCEPTS

context of this thesis, since we label every node starting from 0 to $|V(G)| - 1$, a mapping can be thought of as a permutation of the set of node labels.

Two graphs G and H are said to be *isomorphic*, denoted as $G \sim H$, if there is a one-to-one mapping between the vertices of both graphs and there is an edge between two vertices of G if and only if their corresponding vertices in H also form an edge (preserving direction in the case of directed graphs). More informally, isomorphism captures the notion of two networks having the same edge structure, the same topology, if we ignore distinction between individual nodes.

Figure 2.3 illustrates this concept. Despite looking different, the structure of the graphs is the same, and they are isomorphic. The labels in the nodes illustrate mappings that would satisfy the conditions given for isomorphism.

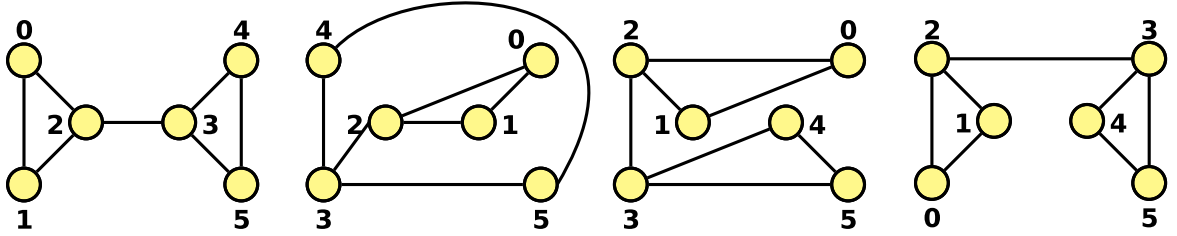


Figure 2.3 – Four isomorphic undirected graphs of size 6.

The set of isomorphisms of a graph into itself is called the group of *automorphisms* and is denoted as $Aut(G)$. Two vertices are said to be *equivalent* when there exists some automorphism that maps one vertex into the other. This equivalence relation partitions the vertices of a graph G into equivalence classes denoted as G_E . Informally, these concepts capture the notion of symmetry in the graph and are illustrated in Figure 2.4, where colors represent the equivalence relation, that is, nodes of the same color are in the same equivalence class.

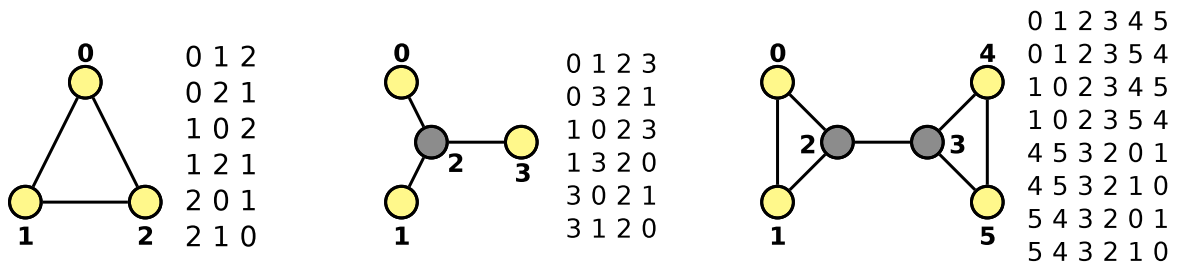


Figure 2.4 – Three undirected graphs and their respective automorphisms.

A *match* of a graph H in a larger graph G is a set of nodes that induce the respective

CHAPTER 2. THE NETWORK MOTIFS PROBLEM

subgraph H . In other words, it is a subgraph G_k of G that is isomorphic to H . This concept is illustrated in Figure 2.5.

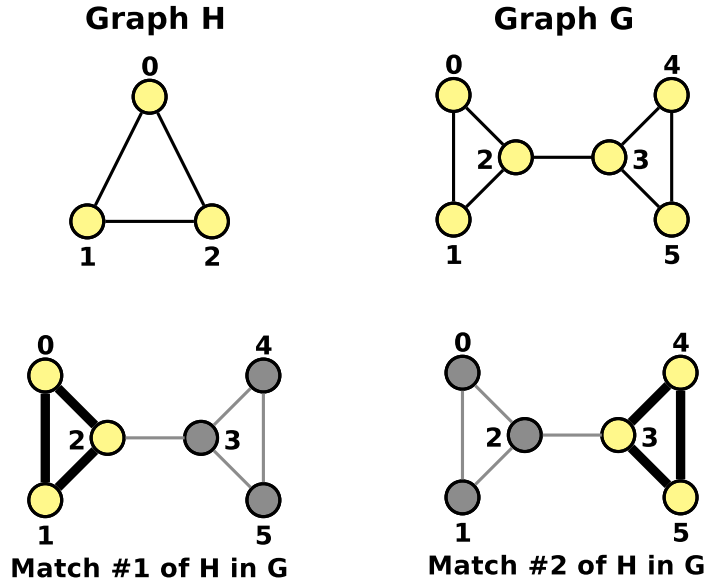


Figure 2.5 – Two possible matchings of a graph H in another graph G .

A **path** is a sequence of vertices such that there is an edge connecting any adjacent pair of vertices in the sequence. Two vertices are said to be **connected** if there is a path between them. A **connected graph** is a graph in which every pair of nodes is connected. An **articulation point** is a node from a connected graph that when removed disconnects the graph and creates two or more separated subgraphs.

For the sake of simplicity, the term *motifs* will refer to network motifs, except when another meaning is explicitly stated.

2.2 Network Motifs Problem

To define network motifs, we now overview existing definitions.

2.2.1 The Original Definition

Milo et al. [MSOI⁺02] first introduced in 2002 the terminology “network motifs” and provided the first core informal definition:

2.2. NETWORK MOTIFS PROBLEM

Definition 2.1 (Network Motifs - informal definition) *Network motifs are patterns of inter-connections occurring in complex networks in numbers that are significantly higher than those in similar randomized networks.*

The main initial motivation was to go beyond global features (such as the “small world” or “scale free” properties described in section 1.1) and try to find basic structural elements that were characteristic to each class of networks [ASBS00].

Their reasoning was also that the structural detection of motifs could give new insight into their dynamical and functional behavior. A possible interpretation was that the motifs appeared because of constraints in the way the network was developed [CHK⁺01], thus being related to the evolution of the whole complex system.

Definition 2.1 means that a motif is a subnetwork which is statistically over-represented. We will now describe how the informal definition was put into practice in the original paper [MSOI⁺02].

The key aspect to ensure statistical meaning is to be able to generate the random networks as similar as possible to the original one. We want to be sure that the intrinsic global and local properties of the network do not determine the motif appearance and that the motif is indeed specific to this particular network. The original proposal was therefore to maintain all single-node properties, namely the *in* and *out* degrees.

Figure 2.6 exemplifies this concept. Note that the number of incoming and outgoing edges for any node remains the same in all networks. Note also that both the original and randomized networks are simple graphs.

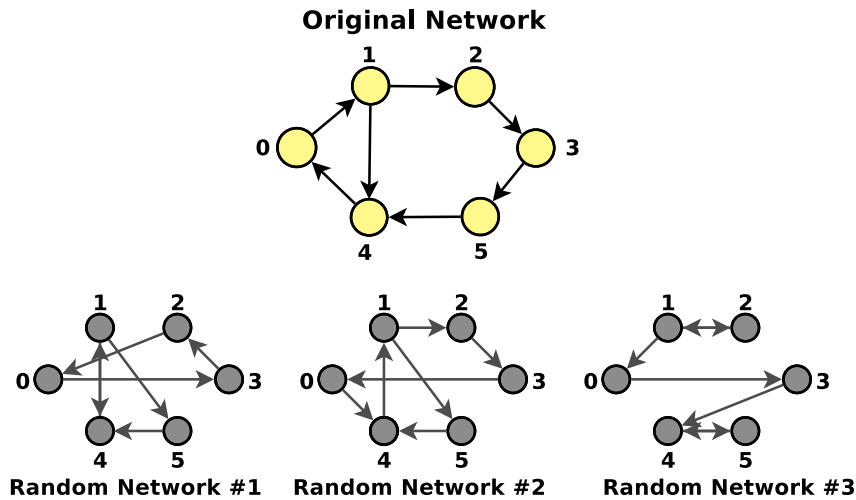


Figure 2.6 – Example of similar random networks preserving degree sequence.

CHAPTER 2. THE NETWORK MOTIFS PROBLEM

For the term *patterns*, Milo et al. [MSOI⁺02] used induced subgraphs. In order to count them, overlapping between two occurrences (matches) of a specific subgraph is allowed, that is, in order for two occurrences to be considered different it is a sufficient condition that they present a different set of nodes (even if they share a subset of nodes). Section 2.2.2 presents different possibilities for frequency counting.

Figure 2.7 presents an example of how to find the frequency of a subgraph, using the Milo et al. [MSOI⁺02] definition. The subgraph of size 3 is matched 3 times in the network: $\{0, 1, 4\}$, $\{0, 2, 4\}$ and $\{1, 4, 5\}$.

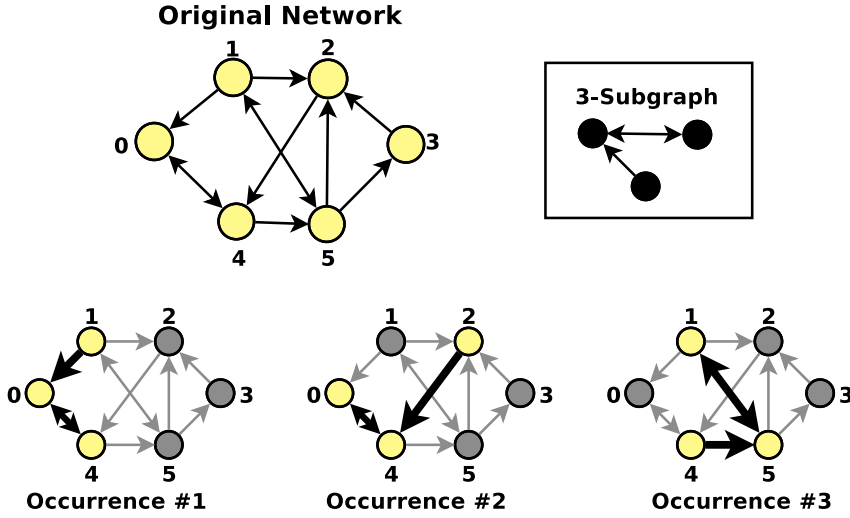


Figure 2.7 – Example of frequency count.

Given this frequency definition, a motif would be classified as such when three different properties hold at the same time.

Over-representation: The probability that the frequency of a motif in a randomized network is greater than the frequency in the original network should be smaller than a determined *probability* threshold P . This is empirically determined using an ensemble of a large number of similar random networks as described before and will ensure that the motif is over-represented in the original network. P is estimated by assuming a random null hypothesis and z -scores (on a standard normal distribution). Let $f_{original}$ be the frequency in the original network and f_{random} be the frequency in a random network. We can then define the z -score as in equation 2.1 and use a pre-calculated table to infer the desired probability (with σ being the standard deviation).

$$z\text{-score}(G_k) = \frac{f_{original} - \bar{f}_{random}}{\sigma(f_{random})} \quad (2.1)$$

2.2. NETWORK MOTIFS PROBLEM

Minimum frequency: The frequency of the motif on the original network should be higher than an *uniqueness* threshold U . This ensures a quantitative minimum to establish significance.

Minimum deviation: The third and last property is that the frequency of the motif on the original network is significantly larger than its average frequency on the similar random networks. This prevents the detection of motifs that have a small difference between these two values but have a narrow distribution in the random networks. D is a proportional *deviation* threshold that ensures the minimum difference between $f_{original}$ and \bar{f}_{random} , as formalized next.

These three properties lead us finally to a more formal version of the definition of motifs.

Definition 2.2 (Network Motifs - formal definition) *An induced subgraph G_K of a graph G is called a network motif when for a given set of parameters $\{P, U, D, N\}$ and a random ensemble of N similar networks:*

1. $Prob(\bar{f}_{random}(G_K) > f_{original}(G_K)) \leq P$ (**Over-representation**)
2. $f_{original}(G_K) \geq U$ (**Minimum frequency**)
3. $f_{original}(G_K) - \bar{f}_{random}(G_K) > D \times \bar{f}_{random}(G_K)$ (**Minimum deviation**)

The original paper from Milo et al. [MSOI⁺02] uses $\{0.01, 4, 0.1, 1000\}$ as the set of respective parameters $\{P, U, D, N\}$. This is the same as saying that they used 1000 similar random networks and considered a subgraph to be a motif if: the probability that it appears more often in a random network than in the original network is less than 1%; it appeared at least 4 times; the difference between its frequency in the original network and the average frequency in random networks is at least 10% of that average frequency.

Note that other values can be used for the parameters depending on what we want to accomplish. Its usage in the definition, rather than narrowing the concept, introduces extra flexibility. This definition constitutes the original complete formalization of motifs that is still regarded as the canon, and is used on the vast majority of motif related papers.

CHAPTER 2. THE NETWORK MOTIFS PROBLEM

2.2.2 Variations in the Definition

The definition given above is the standard for motifs detection and will be the one used in this thesis. However, several related variations have been proposed and applied, introducing different details and constraints. Next, we overview the existing variations to give the reader a better grasp of the entire concept.

General Applicability

The first point to note is that the concept is very broad. It can be applied equally to *directed* or *undirected* networks without changing its behavior and method of calculation.

One other possible application is to use the definition on colored networks, giving origin to *colored motifs* [LFS06, FFHV07]. In these networks, nodes have associated with it a label, the *color*, and therefore not all nodes are the same and indistinguishable. Different labels can represent for example different chemical compounds on a metabolic network [LFS06]. Two occurrences of the same motif must therefore not only preserve the same edge structure, but also the color structure of the nodes, as exemplified in Figure 2.8. $\{0, 4, 5\}$ is an occurrence of the subgraph but $\{3, 4, 5\}$ is not, because although edges have the same structure, colors in the nodes are not preserved.

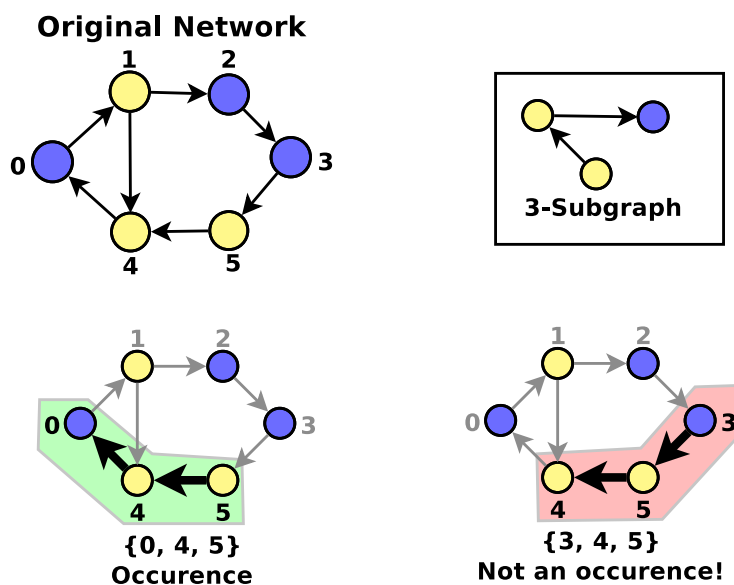


Figure 2.8 – Counting the frequency of colored subgraphs.

The original definition was intended for simple graphs, but it is also applicable to non-simple graphs. In that case, extra care should be taken on the generation of

2.2. NETWORK MOTIFS PROBLEM

randomized networks, with additional constraints. For example, in a network with self-loops but no multiple edges, one could determine that self loops must be preserved in the randomized networks, effectively adding “colors” to the networks, in the sense that nodes could be classified in two groups: with or without self-loops.

Frequency Concepts

The frequency of a subgraph is the number of different matches of that particular subgraph in the network. As said in the previous section, the original definition allows for arbitrary overlapping of edges and nodes on different matches of a motif. This has an associated functional meaning as in, for example, biological applications. In these networks it is possible for several different overlapping subgraphs to be active and functioning at the same time, with the same motif assuming different functions on each occurrence, as for example is the case of proteins in PPI networks [CG08].

Schreiber and Schwobbermeyer [SS04], however, introduce different concepts for the frequency, allowing more constraints, particularly no sharing of edges and/or nodes. This has the potential to drastically change the frequency of a motif, thus improving the tractability of the motif discovery problem. By adding more constraints, the number of matches is reduced. However, a new problem is introduced, as there are several possibilities for choosing the correct matches.

Figure 2.9 and Table 2.1 illustrate the four different frequency concepts introduced in [SS04]. F_1 is the one used in the original definition. With F_2 several possibilities for the matching arise, and we have the new problem of deciding which ones should be chosen (ambiguity in the set of matches found). Note that the number of matchings under F_2 is always smaller or equal than the ones under F_1 . F_3 is even more restrictive and potentially counts even less matches (at most it considers the same number of matches of F_2), and gives origin to cases with the same ambiguity problem of F_2 . Note also that a fourth frequency concept, allowing overlapping of edges but not of nodes, is not applicable because edges always connect nodes.

Under-Representation

If we consider over-represented subgraphs, we could also think of under-represented ones, that is, subgraphs whose frequency in the original network is much smaller than it would be expected in similar random networks. Such subgraphs are called *anti-motifs* [MIK⁺04], and may be meaningful for certain applications.

CHAPTER 2. THE NETWORK MOTIFS PROBLEM

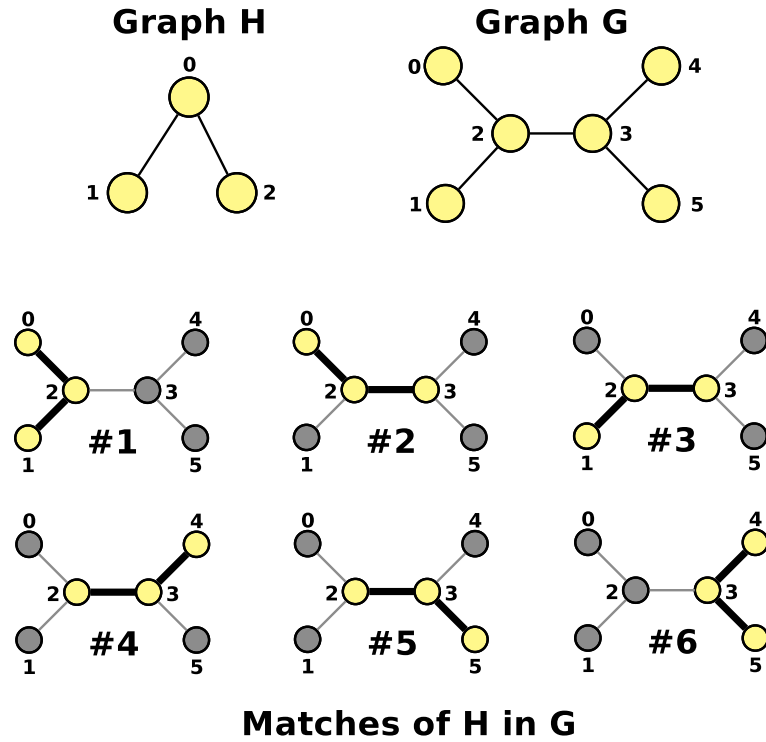


Figure 2.9 – Two graphs and all the possible matchings of one into the other.

Frequency Concept	Allow sharing		Maximum Matches in Figure 2.9
	Nodes	Edges	
F_1	yes	yes	$\{\#1, \#2, \#3, \#4, \#5, \#6\}$
F_2	yes	no	$\{\#1, \#4\}$ or $\{\#1, \#5\}$ or $\{\#1, \#6\}$ or $\{\#2, \#6\}$ or $\{\#3, \#6\}$
F_3	no	no	$\{\#1, \#6\}$

Table 2.1 – Different frequency concepts and respective maximum number of matches in the graphs of Figure 2.9. Adapted from [SS04].

2.2. NETWORK MOTIFS PROBLEM

Non-induced subgraphs

Some authors take into account subgraphs which are not induced [MZW05, CHLN06], meaning that the total number of possible subnetworks would be much larger. This is mainly because the same set of nodes could be a match for different motifs. In fact, each match for a specific graph H of size k in G would also be a match for all of its non-induced subgraphs H_k , as exemplified in Figure 2.10. Graphs H1, H2 and H3 are non-induced subgraphs of H. Therefore, if non-induced matches should be considered, any match for H would also be a match for H1, H2 and H3. In a way, this information can be redundant because information about non-induced subgraphs is contained in the induced subgraphs.

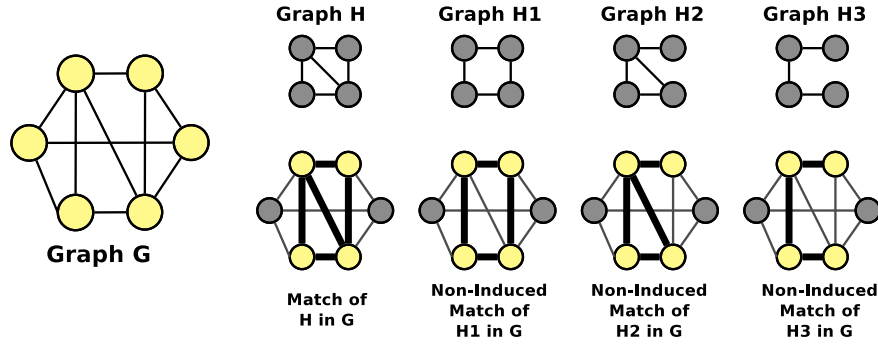


Figure 2.10 – Non-induced subgraph matches.

Statistical Significance

The notion of the statistic significance can have a different formulation. Instead of using the z -score, some authors also use the so called *abundance* (Δ) [MIK⁺04], as defined in equation 2.2 (ϵ is a very small constant to ensure that when the frequency is small the abundance will not be misleadingly large).

$$\Delta(G_K) = \frac{f_{original} - \bar{f}_{random}}{f_{original} + \bar{f}_{random} + \epsilon} \quad (2.2)$$

Another approach is to sample some of the subgraphs and then estimate the concentrations of the studied motifs in the original network [KIMA04a], as detailed in equation 2.3, where the denominator indicates the sum of the frequencies of all the k -subgraphs (more about sampling can be seen in Section 3.3). Calculating the average and standard deviation of the concentration in the random ensemble of networks will then give an estimate of the z -score.

CHAPTER 2. THE NETWORK MOTIFS PROBLEM

$$C(G_K) = \frac{f(G_K)}{\sum f(G_K)} \quad (2.3)$$

Similarity of Random Networks

Sometimes the generation of random networks can have more constraints than only keeping the degree sequence of the original network.

The original paper [MSOI⁺02] tries to guarantee that when searching for k -motifs, the frequency of $(k - 1)$ -motifs stays the same, ensuring that the significance of a particular pattern does not simply derive from its sub-patterns. This was however only possible for very small motif sizes, due to computational reasons, and therefore it is not a common practice to ensure this.

Other authors have also experimented with networks that follow specific connection rules, such as lattices [SK04], that is, graphs whose layout follow a grid pattern. They pursued the idea that the appearance of certain motifs could be favored by that inherent topology or physical location of the nodes.

The general rule for creating the set of similar randomized networks is that we should try to use a null model that best encapsulates what we know of the network. The given solution of preserving the degree sequence is general enough to be applicable in many situations, but in some cases other options may result in more accurate results.

Avoiding the Generation of Random Networks

The generation of a random ensemble of similar networks can be cumbersome and is certainly time consuming. Analytical methods to derive the desired probability for a subgraph to be over-represented would be most welcome. This would avoid the need to explicitly generate all the random networks and count subgraphs in them.

Recent work pursues this line of research [MSB⁺06, Wer06, PDK⁺08, SLS08]. These theoretical statistical approaches have an enormous potential, but they still require further development in order to be used in the general case and to have the necessary accuracy. Either they still take too much time to compute or they are still not able to model general random networks with specific constraints, like preserving the degree sequence.

In any case, even if we are able to analytically derive the significance of a particular subgraph, we would still need to compute its frequency on the original network, and therefore the core computational problem related to motifs calculation (as will be seen

2.2. NETWORK MOTIFS PROBLEM

in Section 3.1) still remains.

We should also note that Birmelé presented an alternative motif definition, in which significance is evaluated through local over-representation, instead of global over-representation, and introduced a statistical method for computing this measure without the generation of random networks [Bir11].

Weighted Networks

Application of the motifs concept to weighted networks is not straightforward. These are networks where a value, a weight, is associated with each edge. A possible direct adaptation would be to define a threshold and keep the edges with weights above that value and discard the others. Another option is given by Saramäki et al. [SpOKK05], that propose a way to fully incorporate weights in the computation of motifs. They start by defining the *intensity* $I(G_k)$ of a subgraph G_k as the geometric mean of its weights:

$$I(G_k) = \left(\prod_{e \in E(G_k)} \text{weight}(e) \right)^{1/|E(G_k)|} \quad (2.4)$$

Note that since weights are multiplied, if an edge has weight zero, then this will be the intensity of the respective subgraph. Similarly, if one weight is small, it will significantly reduce the respective intensity. The *total intensity* of a subgraph is then defined as the sum of all of its subgraph intensities, and from that they derive the statistical significance.

This approach does not discretize the frequency of subgraphs and instead creates a continuum of possible total intensities of a subgraph.

Networks over Time

Some complex networks are not static and change over time. In this case, the network motifs also change and they can help in understanding the dynamics of the networks. With that in mind, Jin et al. [JMA07] proposed the notion of *trend motifs*, which are basically recurring subgraphs that display similar dynamics over a predefined period, and presented algorithms for their discovery.

Related Concepts

There exists a vast amount of work on graph mining. Particularly, the field of *frequent subgraph mining* has been very prolific. Although related, this problem, which derives

CHAPTER 2. THE NETWORK MOTIFS PROBLEM

from the *frequent itemset problem* [PBTL99], is substantially different because its goal is to find the most frequent subgraphs that appear in a set of graphs, while in the network motifs we try to find the frequency of all subgraphs on a single graph.

Frequent subgraph mining has produced sequential algorithms like MOFA [BB02], gSpan [YH02], FFSM [HWP03] or Gaston [NK04]. Because of their goal, these algorithms differ substantially in concept to motif discovery approaches.

It should be noted that some authors use the term *motif* in the context of this related problem for subgraphs which simply occur frequently, but are not necessarily over-represented [WP04, HBP⁺06].

2.3 Applications of Network Motifs

This section gives a general overview of the types of possible applications using the concept of network motifs. It does not aim to be complete, but instead to show the purpose of the concept, demonstrating its usefulness, flexibility and general applicability.

Its original application usage was to distinguish classes of networks based on the types of motifs. The original paper [MSOI⁺02] found that the same small-sized motifs were found on networks of the same type, as illustrated by Table 2.2. N_{real} and N_{rand} indicates respectively the frequency in the original network and the average frequency in the random networks. Z-score is calculated as in Equation 2.1. The threshold values for $\{P, U, D, N\}$ (see Section 2.2) are $\{0.01, 4, 0.1, 1000\}$.

This approach was continued in [MIK⁺04], where superfamilies of networks were identified based on motifs. In order to do that, a systematic study of similarity in local structure was followed in which a significance profile (SP) of motifs was built. Basically, the SP is a vector of the normalized z-scores of a set of subgraphs (normalized in order to compare networks of different sizes). Figure 2.11 shows that different networks displayed very similar SPs for the set of all 3-subgraphs (triads). They were therefore grouped in similar families of networks.

In the figure, the normalized z-scores for the same set of 13 different directed subgraphs were plotted with lines to help visualization. Complete details of the networks can be seen in [MIK⁺04], but the names describe them in general terms. “TRANSC” indicate transcription networks, “SIGNAL” indicates signal transduction networks,

2.3. APPLICATIONS OF NETWORK MOTIFS

“NEURONS” indicate synaptic connections, “WWW” indicates hyperlinks between web-pages, “BI-PARTITE” indicates a bi-partite model graph and the names of languages indicate word-adjacency networks of texts.








Network	Nodes	Edges	N_{real}	N_{rand}	Z-Score	N_{real}	N_{rand}	Z-Score	N_{real}	N_{rand}	Z-Score
Gene Regulation (transcription)			 Feed-forward loop			 Bi-Fan					
E. coli	424	519	40	7 ± 3	10	203	47 ± 12	13			
S. cerevisiae	685	1052	70	11 ± 4	14	1812	300 ± 40	41			
Food Webs			 Three chain			 Bi-Parallel					
Little Rock	92	984	3219	3120 ± 50	2.1	7295	2220 ± 210	25			
Ythan	83	391	1182	1020 ± 20	7.2	1357	230 ± 50	23			
Electronic Circuits (digital fract. multipliers)			 3-node loop			 Bi-Fan			 4-node loop		
s208	122	189	10	1 ± 1	9	4	1 ± 1	3.8	5	1 ± 1	5
s420	252	399	20	1 ± 1	18	10	1 ± 1	10	11	1 ± 1	11

Table 2.2 – Network motifs found in different types of networks. Adapted from [MSOI⁺02].

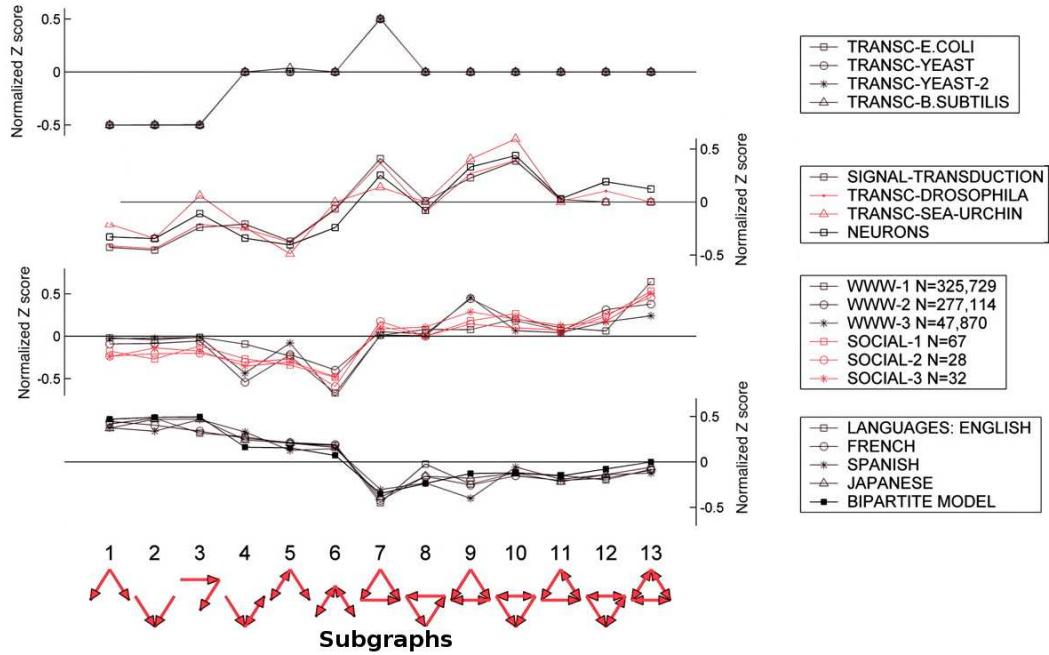


Figure 2.11 – Triad significance profiles. Adapted from [MIK⁺04].

CHAPTER 2. THE NETWORK MOTIFS PROBLEM

A measure similar to SPs, designated *motif fingerprint*, was also used to study brain neural networks [SK04]. Figure 2.12 exemplifies this on some cortical areas of the Macaque Visual Cortex. Numbers 1 to 12 are motif types, and the polar plots indicates their frequency. Five of those (V1, V3, V4, MSTd and DP) present very similar fingerprints, while other areas, such as V2, V4t and PIVt are different. These fingerprints were then used to cluster cortical areas based on their similarity, as can be seen of Figure 2.13.

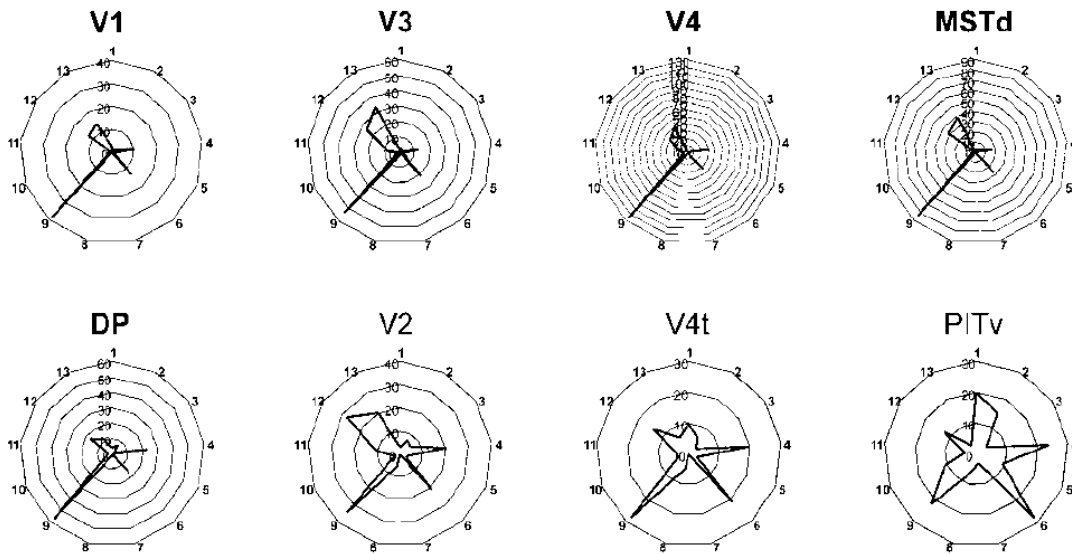


Figure 2.12 – Polar plot of motif fingerprints in Macaque Visual Cortex. Taken from [SK04].

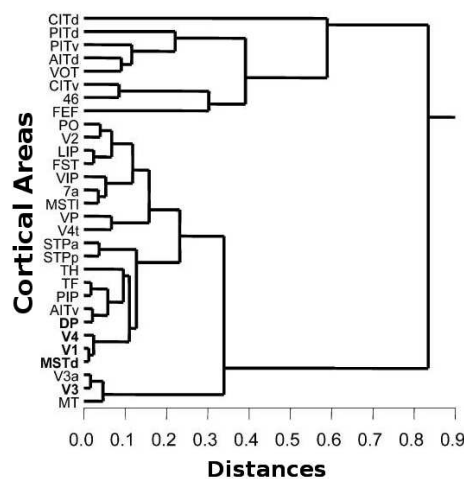


Figure 2.13 – Clustering the Macaque Visual Cortex with motif fingerprints. Taken from [SK04].

2.3. APPLICATIONS OF NETWORK MOTIFS

Motifs can also present specific information processing functions in the network. Shen-Orr et al. [SOMMA02] show that much of the network of transcriptional interactions of the bacteria *E. coli* is composed by three different types of highly significant motifs: feed-forward loops (FFL), single input modules (SIM) and dense overlapping regulons (DOR). The respective subgraphs are shown in Figure 2.14 and together they compose a large part of the network.

Explaining the biological meaning of these motifs is out of the scope of this thesis, but it was demonstrated that each has “*a specific function in determining gene expression, such as generating temporal expression programs and governing the responses to fluctuating external signals*” [SOMMA02].

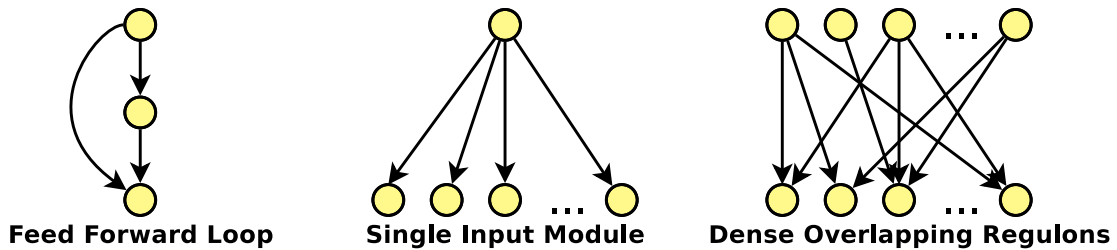


Figure 2.14 – The three different types of motifs found on the transcriptional regulation network of *E. coli*. Adapted from [SOMMA02].

Determining the function of motifs in information processing is recurrent in biological networks analysis and has been extensively applied [LRR⁺02, Alo03, MA03, MZA03, WOB03, MBV05],

The recurrence of motifs has also been applied graphically to represent the interactions as a whole and give a more human interpretable visualization of the network[SOMMA02]. Instead of plotting the actual nodes and connections, one can use specific symbols for determined motifs and simplify the visual depiction of the network, as exemplified in Figure 2.15, where part of the regulation network of *E. coli* is represented using symbols for specific motifs.

Motifs were also shown to be evolutionary conserved topological units [WOB03], in the sense there seems to be evolutionary pressure to maintain those motifs. These conservation property was in turn used to predict protein-protein interaction, by using machine learning techniques that try precisely to conserve those motifs [AA04]. This allows the generation of likely candidates for interaction.

Other work shows that in some networks the vast majority of motifs tend to overlap, generating *motif clusters* [DBBO04]. An example can be seen in Figure 2.16,

CHAPTER 2. THE NETWORK MOTIFS PROBLEM

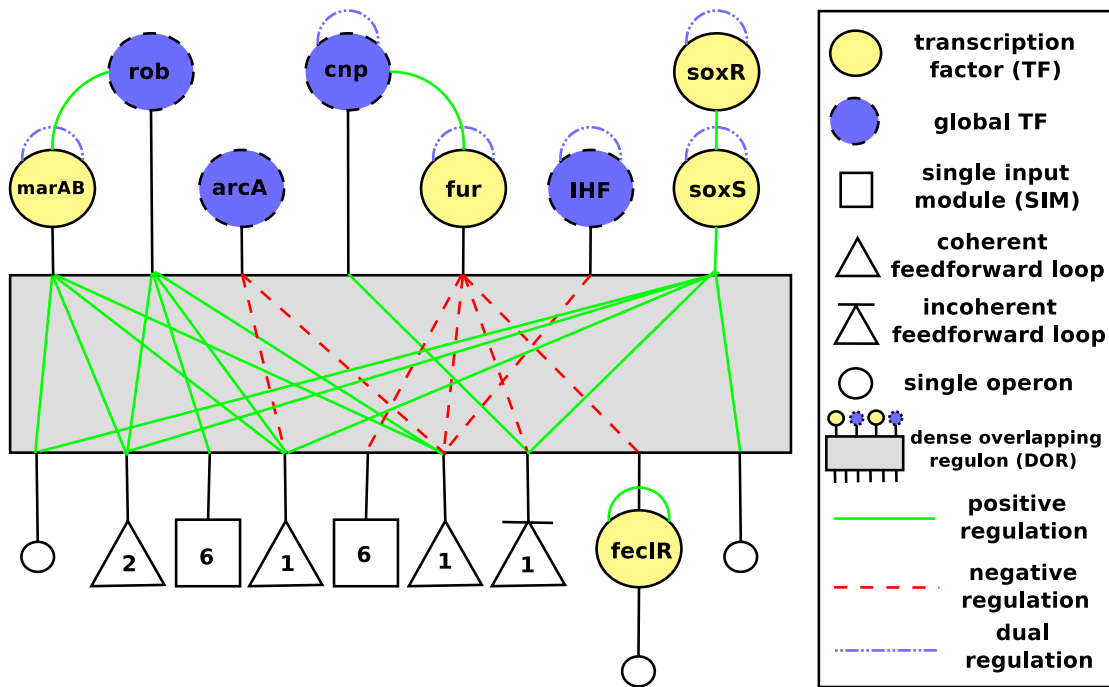


Figure 2.15 – Using motifs as building blocks of the actual plotting of the regulation network of *E. coli*. Adapted from [SOMMA02].

showing a cluster of feed forward loops. The dashed edges are only used by a single occurrence of the motif, while continuous lines are shared by at least two occurrences. The colors of the nodes denote the “role” of the node in the motif, that is, if it is the node with two outgoing edges (the *input*), the node with one incoming and one outgoing edge (the *intermediate*) or the node with two incoming edges (the *output*).

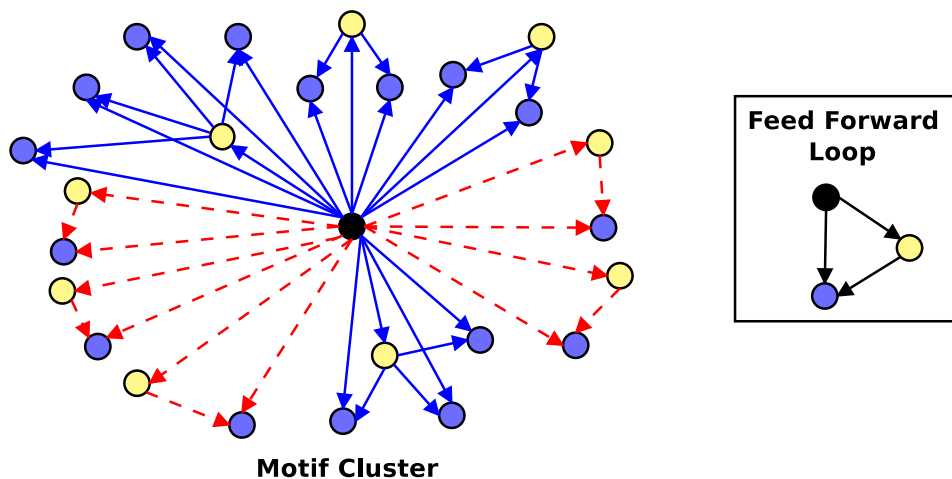


Figure 2.16 – Example of a motif cluster on the *E. coli* transcriptional regulatory network. Adapted from [DBBO04].

In order to better understand how motifs combine to form larger structures the notion of *motif generalizations* was introduced and studied [KIMA04b]. Based on the “roles” of each node, possible ways of combining into larger structures were defined, as exemplified in Figure 2.17, where 4-node generalizations of the feed forward loop are shown.

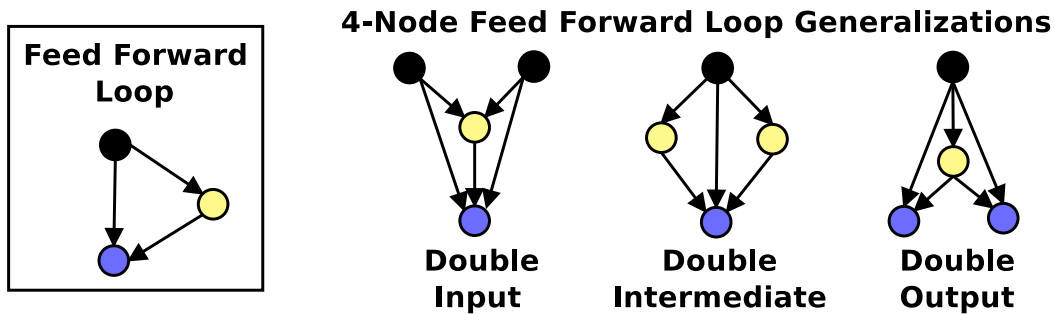


Figure 2.17 – Example 4-node generalizations of the feed forward loop. Adapted from [KIMA04b].

Motifs have also been used in other varied contexts, like in the following three examples. They were crucial in a work with the goal of disambiguating the appearance of personal names in the World Wide Web [YCLH06]. They were used to identify patterns in object oriented pieces of software, where nodes are classes and edges indicate static dependencies among them [VS05]. And finally, they were studied in geometric networks (like grids), whose layout is constrained by the physical position of the actual nodes [IA05].

2.4 Criticism

Even taking into account the general applicability of motifs, it should however be said that its usage did not come without criticism. The first comments arrived as soon as the original 2002 paper appeared, arguing that the original null hypothesis used (random networks with the same degree node) may not be able to identify evolutionary design principles in the appearance of the motifs [ARFBTS02, HJ05].

Other major point argued is that structure does not completely determine function [ISS06, KNS08]. The same motif may even have opposite functionality on a different context, and additional information, which may be difficult to obtain, can be necessary in order to evaluate the real function of some motif.

CHAPTER 2. THE NETWORK MOTIFS PROBLEM

Other work points out that global networks features, like the clustering coefficient, may indeed have a crucial influence on local characteristics such as the frequency of a determined type of subgraph [VDS⁺04, KL08].

This thesis does not try advocate any position on this matter, although it is made having in mind that motif discovery is useful and has already been demonstrated to be of practical usage. Instead we focus our attention on studying efficient strategies for finding network motifs, with particular emphasis on subgraph frequency discovery, which has an even larger applicability scope.

2.5 Summary

This chapter introduced the necessary graph concepts and established a terminology that will be used coherently in the following chapters. With these tools at hand, the network motif concept was formalized and concisely defined. A review of possible variations on the motif definition was made and a large set of example applications was given, in order to give the reader a better feel of the practical usability of the concept. Finally, in order to put some perspective, some criticisms on motif usage were presented.

*We are drowning in information but
starved for knowledge.*

Rutherford D. Rogers

3

Algorithms for Motif Discovery

The purpose of this chapter is to provide a thorough survey of the state of the art in algorithms for network motif discovery. We start with an historical perspective of the field, giving a time line for the existing algorithms. Then follows an overview, with a comparison table and an associated taxonomy. We continue by showing pseudo-code for all main existing algorithms, organized by their functionality. Finally, we overview the existing parallel approaches.

3.1 Algorithmic Approaches

Like many other subgraph problems (such as finding maximal independent or bipartite sets), discovering network motifs is a *computationally hard* problem. Fundamentally, we will be matching graph patterns with the desired motifs, which leads to the well known Graph Isomorphism Problem:

Definition 3.1 (Graph Isomorphism Problem) *Given two finite graphs G and H , determine if they are isomorphic.*

This problem is in the set of non-deterministic polynomial time (NP) problems, with no known fast and general solution in polynomial time. It is also one of the few NP problems that it is still not known to be NP-complete [McK81, KST93]. While it

CHAPTER 3. ALGORITHMS FOR MOTIF DISCOVERY

is true that for special cases of graphs and subgraphs more efficient solutions may exist [KKM00], it remains the fact that the general case is still not easily tractable.

The problem can be generalized even more in the Subgraph Isomorphism Problem, which is known to be NP-complete [Coo71], and therefore also computationally intractable.

Definition 3.2 (Subgraph Isomorphism Problem) *Given two finite graphs G and H , determine if G contains a subgraph that is isomorphic to H .*

We will later use these problem definitions to classify sub-problems of the motif discovery process, in order to show that indeed it is a computationally hard problem.

3.1.1 The Motif Discovery Program Flow

In the general case we do not know *a priori* which motifs we will find. Because of that, the typical motif discovery algorithmic pattern consists in choosing a subgraph size k and then finding all motifs of that size, that is, with k vertices. In order to do that, the typical program flow is the one depicted in Figure 3.1.

Algorithm 3.1 Typical program flow of motif discovery.

Require: Graph G and integers k and R

Ensure: Motifs of size k in graph G

```
1: SUBGRAPHCENSUS( $k, G$ )
2: for  $i := 1$  to  $R$  do
3:    $R_i := \text{GENERATESIMILARRANDOMNETWORK}(G)$ 
4:   SUBGRAPHCENSUS( $k, R_i$ )
5: CALCULATESIGNIFICANCEMOTIFS()
```

The algorithm first computes a *k-subgraph census* on the original network (line 1). This results in a histogram with the frequency of all the existing classes of isomorphic k -subgraphs. After that, an ensemble of R similar random networks is generated (line 3) and a *k-subgraph census* is applied to each of those networks (line 4). Finally, after knowing the frequency of each existing isomorphic class of subgraphs on all networks (original and randomized), its significance is calculated (line 5), with the over-represented subgraphs being reported as motifs.

3.1. ALGORITHMIC APPROACHES

Exemplifying, if k is 3, and if we were considering simple directed subgraphs, the 13 different classes of isomorphic 3-subgraphs depicted in Figure 3.1 would be counted in the network (with the frequency of some of them being potentially zero). Note that if non-simple graphs would be considered, the number of possible subgraphs would be infinite, since for example there could be an arbitrary number of edges between any pair of vertices.

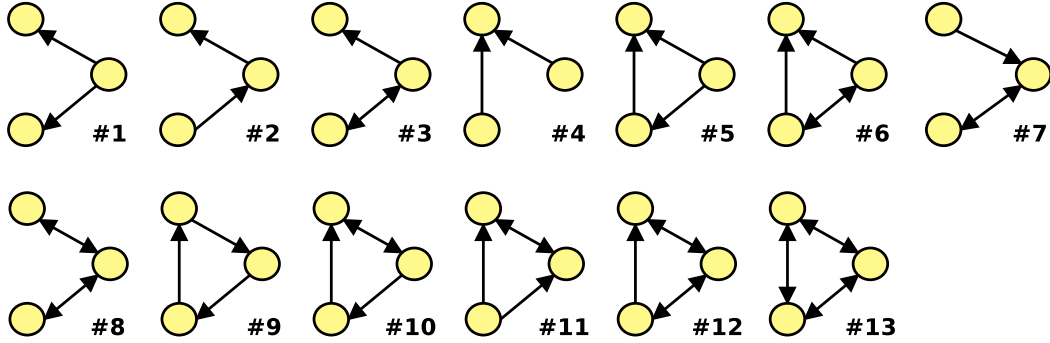


Figure 3.1 – The 13 different classes of isomorphic subgraphs of size 3. Adapted from [MSOI⁺02].

One problem with this approach is that the number of possible k -subgraphs grows super exponentially as we increase the size k , as can be seen in Table 3.1. Naturally, some of these subgraphs may not be found at all in the networks, but it remains that they may all appear in the worst case scenario.

Size k	3	4	5	6	7	8	9	10
Number of Undirected Subgraphs	2	6	21	112	853	$\approx 10^4$	$\approx 10^5$	$\approx 10^7$
Number of Directed Subgraphs	13	199	9364	$\approx 10^6$	$\approx 10^9$	$\approx 10^{12}$	$\approx 10^{16}$	$\approx 10^{20}$

Table 3.1 – Number of possible different connected directed and undirected subgraphs with k vertices (up to 10).

3.1.2 Historical Overview

We first overview the algorithmic advances in motif discovery, and in the following sections we give a more detailed analysis of specific algorithms. For an experimental

CHAPTER 3. ALGORITHMS FOR MOTIF DISCOVERY

comparison on the performance and execution times of the algorithms please refer to Chapter 6.

Figure 3.2 gives an historical time line for the appearance of all the main motif discovery algorithms. From a computer science point of view this is a relatively young field, with the concept of motifs having only been originated in 2002 [MSOI⁺02]. However, it is a prolific research field, with several proposed approaches.

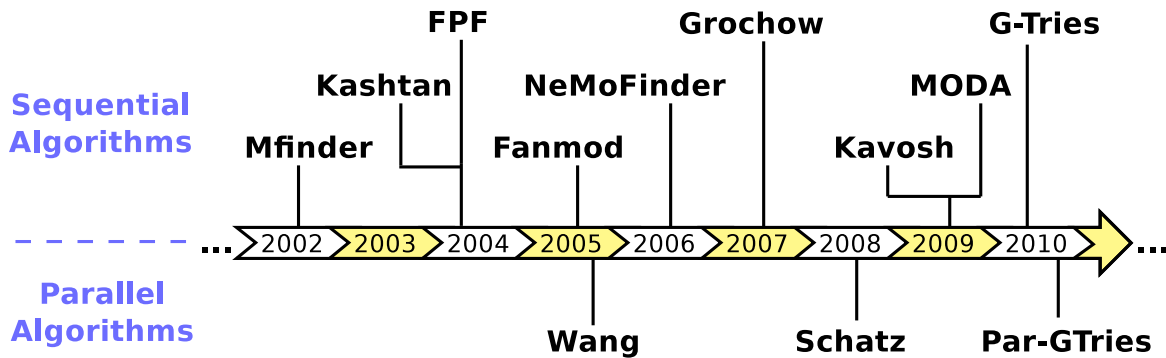


Figure 3.2 – Time line for motif discovery algorithms.

It is only natural that the motifs concept coincided with the first practical implementation of a sequential backtracking algorithm for finding motifs, in 2002, with **mfinder** [MSOI⁺02]. In 2004 the first improvements appeared with the possibility of trading accuracy for better execution times by sampling subgraphs (**Kashtan** [KIMA04a]). In the same year different frequency concepts were introduced and the algorithms were adapted accordingly (**FPF** [SS04]). In 2005 a breakthrough was reached, with the appearance of the first specialized algorithm that could avoid symmetries (**ESU** [Wer05, Wer06]), thus avoiding redundancy in computation. In 2006, the first algorithm able to reach subgraph sizes bigger than 10 appeared, although it succeeded in doing so by twisting a little bit the definition of motifs and only looking for a subset of all possible candidates (**NeMoFinder** [CHLN06]). In 2007, capability of searching and counting individual subgraphs was introduced, instead of being obliged to do a complete subgraph census (**Grochow** [GK07]). In 2009, two new algorithms appeared, similar in concept and asymptotical behavior to **ESU** (**Kavosh** [KAE⁺09]) and **Grochow** (**MODA** [OSMN09]). Finally, 2010 was the year in which the core work of this thesis was published, based around the notion of the new data-structure **g-tries** (refer to Chapter 4).

All the above referenced algorithms were sequential in nature, with no known practical and available parallel adaptations, with the exception of **g-tries**, which is part of the contribution of this thesis. Parallel algorithms were indeed very scarce and

3.1. ALGORITHMIC APPROACHES

only two approaches were published before parallel g-tries: in 2005 a single census was parallelized statically (*Wang* [WTZ⁺05]) and in 2008 an initial sketch of the parallelization of **Grochow** was provided (*Schatz* [SCBB08]).

3.1.3 Comparison of Existing Sequential Algorithms

In order to compare all the main present sequential algorithms, we now give a detailed comparison table, and create an associated taxonomy, in order to give the reader a better overall grasp of the whole set of algorithms available. This can be seen in Table 3.2, which is followed by a description of its column fields.

Method	Strategy Type	Symmetry Breaking	Allows Sampling	Sampling Bias	Graphical Tool	Public Source	Motif Size
Mfinder	Network-centric	no	no	—	yes	yes	small
Kashtan	Network-centric	no	yes	yes	yes	yes	large
FPP	Network-centric	no	no	—	yes	no	small
ESU	Network-centric	yes	yes	no	yes	yes	medium
NeMoFinder	Network-centric	yes	no	—	no	no	large
Grochow	Subgraph-centric	yes	no	—	no	no ¹	large
Kavosh	Network-centric	yes	no	—	no	yes	medium
MODA	Subgraph-centric	yes	yes	no	no	yes	large
G-Tries	Set-centric	yes	yes	no	no	no ²	large

Table 3.2 – Classification and comparison of sequential algorithmic strategies for motifs discovery.

Method indicates the name of the algorithm, associated tool or the first author name of the main reference. This will be the terminology used throughout this thesis when we need to refer to a specific algorithm.

Strategy Type indicates how the algorithm approaches motif discovery, and here we clearly divide and conceptually classify the algorithms in three large groups:

- **Network-centric** the method must be applied to the whole network, in order to find all motifs. It basically consists in enumerating all subgraphs and then finding which ones are isomorphic. The enumeration itself can be very time consuming

¹Source code is available upon request, but it is not made publicly available on the web.

²Currently no, but it will soon be made publicly available.

CHAPTER 3. ALGORITHMS FOR MOTIF DISCOVERY

and afterwards we basically have several instances of the Graph Isomorphism Problem, which is NP.

- **Subgraph-centric** the core of method is applicable to a single subgraph. It basically consists in generating or choosing the candidate subgraphs (for example, all subgraphs of a determined size) and then finding in turn all isomorphic matchings of each individual subgraph. Each of these instances is at least as hard as the NP-complete Subgraph Isomorphism Problem, since the subgraph may not appear at all, and in that case we will be computing the same problem. Moreover, if the graph exists, the given problem is even more difficult. It is not enough to know that the subgraph appears at least once, and we also compute its frequency.
- **Set-centric** the method is applied to a set of input subgraphs. It basically consists in generating or choosing the candidate subgraphs as before, but then it finds at the same time all isomorphic matchings of all the input subgraphs. This is in a way a mix of the two previous approaches.

Symmetry Breaking indicates if the method only finds once each occurrence of the motif. Since a subgraph can have several automorphisms, a careless approach could end up finding the same subgraph following a different search path. It is therefore desirable to find each instance only once, avoiding redundant calculations due to symmetry.

Allows Sampling indicates if the method allows subgraph sampling in order to trade accuracy for faster execution times. This consists in considering just a fraction of all the subgraph occurrences, which will result in approximate results.

Sampling Bias, for the methods with sampling, indicates if this sampling is biased. If it is the case, the method must provide a way to statically correct this bias at the end of the computation.

Graphical Tool indicates whether the method has available a graphical production software tool, ready to be used by anyone wishing to do so. By graphical we mean that it has a graphical user interface and/or that it produces a visual depiction of the results.

Public Source indicates if the source code of the respective method is publicly available, ready to be compiled and run at the command line. All available sources are written in C/C++, with the exception of **Grochow**, which is written in Java.

3.2. SEQUENTIAL EXACT CENSUS

Motif Size gives an indication of the size of the motifs that the respective algorithms can process in a reasonable amount of time. We considered **small** for subgraph sizes of around 4-7, **medium** for 6-8 and **large** for sizes >8 . Note that the exact maximum possible subgraph sizes depends on many factors, like the network, the method parameters, etc. This only intends to give an approximation and let the reader get a better feel of the algorithms. For a full evaluation of the performance of the algorithms refer to Chapter 6.

3.2 Sequential Exact Census

As described before, one very important part of the algorithmic flow for motif discovery is the computation of the subgraph census of a particular network. This section details how the main algorithms do this part in an exact manner, meaning that the results are completely accurate and not only an approximation.

We will only detail algorithms that are based upon the main standard definition of motifs and that present a significant innovation from previous methods. This leaves out FPF, because its focus is on using different frequency concepts, and NeMoFinder, because it twists a little bit the definition (merging it with the frequent subgraphs concept) in order to achieve larger subgraph sizes.

3.2.1 Original Algorithm - `mfinder`

The first algorithm dedicated to motif discovery appeared in 2002, and was described in the auxiliary notes to the original paper on network motifs [MSOI⁺02]. As it was shown before in Table 3.2, it is a network-centric algorithm, which means it works by enumerating all subgraphs of a determined size. Algorithm 3.2 describes this method using pseudo code.

The algorithm is basically a recursive backtracking search. **Mfinder** starts by choosing an edge (line 1) and constructs the subgraph starting with its two constituent nodes (line 2). Then it incrementally adds a new node that has an edge connected to the already partially constructed subgraph (lines 9 to 12). Whenever the desired subgraph size is achieved, the id of the corresponding isomorphism class is calculated and its frequency is updated in an hash table (line 6). To make the search more efficient, another hash table is maintained (in fact, one different hash table for each subgraph

CHAPTER 3. ALGORITHMS FOR MOTIF DISCOVERY

Algorithm 3.2 `mfindex` enumeration of subgraphs.

Require: Graph G and positive integer k

Ensure: k -subgraphs census of graph G

```

1: for all  $(i, j) \in E(G)$  do
2:   SEARCHSUBSET( $\{i, j\}$ )

3: procedure SEARCHSUBSET( $S$ )
4:   if  $|S| = k$  then
5:     if UNIQUE( $S$ ) then
6:       INCREMENTCOUNT(canonicalLabeling( $S$ ))
7:   else
8:     HASH.INSERT( $S$ )
9:     for all  $i \in S$  do
10:      for all  $(i, k) \in E(G)$  do
11:        if  $(k \notin S \text{ AND } \text{HASH.NOTFOUND}(S \cup \{k\}))$  then
12:          SEARCHSUBSET( $S \cup \{k\}$ )

```

size smaller than k) to determine all sets of vertices (subgraphs) already found. This is used to avoid expanding again from a vertex set already explored (lines 8 and 11). Even with these hash tables, the same subgraph can be found several times (due to symmetries) and a test is made to certify that this is indeed a new uncounted motif (line 5). `Mfinder` needs a lot of memory to maintain all the subgraphs explored (and the associated hash tables), which hinders its capabilities to deal with large motifs.

Figure 3.3 depicts the search tree of `mfindex` algorithm when enumerating all subgraphs of size 3 of a graph with 5 vertices. Each internal node indicates the set passed as a parameter to the `searchSubet()` recursive procedure. Note that several subsets are repeated because of symmetries, which wastes computation time, although they are not counted more than once because of the uniqueness test described above.

It should also be noted that `mfindex` uses a custom algorithm (not well documented) in order to find isomorphisms. This is done in `canonicalLabeling()`, that computes a canonical form of the subgraph, which is then comparable to already found occurrences, in the sense that if two graphs have the same canonical form, then it must mean that they are isomorphic.

3.2. SEQUENTIAL EXACT CENSUS

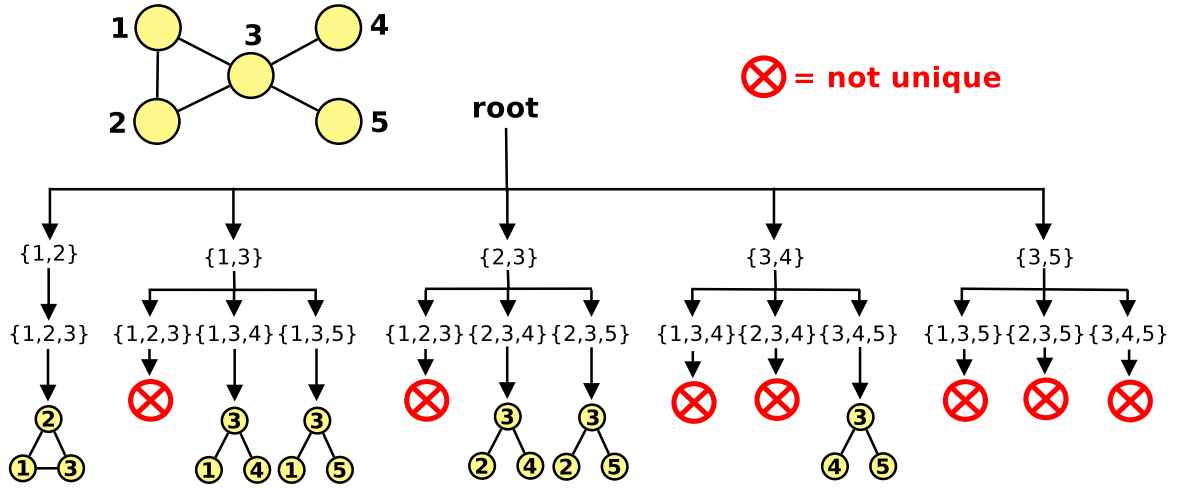


Figure 3.3 – Search tree of mfinder algorithm looking for 3-subgraphs.

3.2.2 ESU Algorithm

ESU appeared later in 2005, and was innovative over existing algorithms because it was able to avoid symmetries and to find all subgraphs only once. In this way, redundant computations were avoided. Algorithm 3.3 describes ESU in pseudo-code.

Algorithm 3.3 ESU enumeration of subgraphs.

Require: Graph G and positive integer k

Ensure: k -subgraphs census of graph G

```

1: for all  $v \in V(G)$  do
2:    $V_{Ext} := \{u \in N(v) : u > v\}$ 
3:   EXTENDSUBGRAPH( $\{v\}, V_{Ext}, v$ )

4: procedure EXTENDSUBGRAPH( $V_{Subg}, V_{Ext}, v$ )
5:   if  $|V_{Subg}| = k$  then
6:     INCREMENTCOUNT(canonicalLabeling( $V_{Subg}$ ))
7:   else
8:     while  $V_{Ext} \neq \emptyset$  do
9:       remove random chosen  $w \in V_{Ext}$ 
10:       $V'_{ext} := V_{ext} \cup \{u \in N_{excl}(w, V_{subg}) : u > v\}$ 
11:      EXTENDSUBGRAPH( $V_{subg} \cup \{w\}, V'_{ext}, v$ )

```

Instead of starting with an edge, this method starts with a single “root” node and expands from there. Its core idea is that when expanding the set of nodes, only the

CHAPTER 3. ALGORITHMS FOR MOTIF DISCOVERY

ones with an index greater than the initial spawning node are allowed (lines 2 and 10). A list of possible vertices for extension is maintained (lines 2 and 10) and each time a vertex is chosen for expansion it is removed from the possible extensions (line 9) and its exclusive neighbours are added to the new possible extensions. The fact that they are exclusive guarantees that each subgraph is enumerated exactly only once, because the ones which are not exclusive will be added on another instance of the recursion. In order to compute isomorphisms, ESU uses a highly efficient third-party algorithm (`nauty` [McK81]).

Figure 3.4 exemplifies in detail how the algorithm enumerates all 3-subgraphs of a graph with 5 vertices. Each internal search tree node indicates the sets passed as parameters to the procedure `extendSubgraph`, respectively V_{Subg} and V_{Ext} .

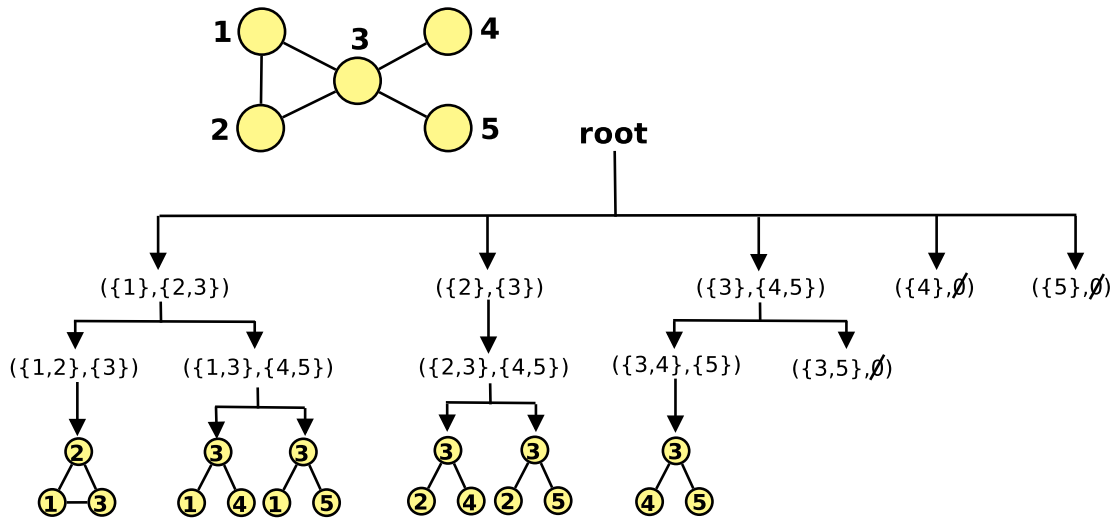


Figure 3.4 – Search tree of ESU algorithm looking for 3-subgraphs.

In 2007 another network-centric algorithm appeared claiming to be “optimal” and faster than ESU [RI07]. This claim was, however, rebutted, and the execution times were in fact worse than ESU [Wer10].

3.2.3 Grochow and Kellys’ Algorithm

The **Grochow** algorithm takes a very different approach. While the other methods are network-centric in the sense that they discover motifs for the whole network and only after do isomorphism tests, **Grochow** concentrates on counting the frequency of a specific isomorphic class. Algorithm 3.4 describes how it counts the number of occurrences of a single query graph.

3.2. SEQUENTIAL EXACT CENSUS

Algorithm 3.4 Grochow enumeration of an individual subgraph.

Require: Graph G , positive integer k and query graph H

Ensure: All occurrences of subgraph H in graph G

```

1:  $H_E := \text{EQUIVALENCE REPRESENTATIVES}(H)$ 
2:  $C := \text{SYMMETRY BREAKING CONDITIONS}(H)$ 
3: Sort all  $g \in V(G)$  by increasing degree and then by neighbour degree sequence
4: for all  $g \in V(G)$  do
5:   for all  $h \in H_E$  do
6:     if  $\text{SUPPORTS}(g, h)$  then
7:        $f :=$  partial map associating  $f(h) = g$ 
8:        $\text{ISOMORPHIC EXTENSIONS}(f, H, G, C)$ 
9:   Remove  $g$  from  $G$ .

10: procedure  $\text{ISOMORPHIC EXTENSIONS}(f, H, G, C)$ 
11:    $D :=$  domain of  $f$ 
12:   if  $D = H$  then
13:      $\text{FOUND OCCURRENCE}()$ 
14:   else
15:      $m :=$  most constrained neighbour of any  $d \in D$ 
16:     for all  $n \in N(f(D))$ , with  $d \in D$  do
17:       if  $f(m) = n$  does not violate  $C$  AND
          $\nexists d \in N(m) : n \notin N(f(d))$  AND
          $\nexists d \notin N(m) : n \in N(f(d))$  then
18:          $f' = f$  on  $D$  and  $f'(m) = n'$ 
19:          $\text{ISOMORPHIC EXTENSIONS}(f', H, G, C)$ 

```

The main idea is to progressively map the desired query subgraph H on the global graph G , instead of enumerating, and only after check for isomorphism. The algorithm starts by finding the equivalence classes of the query graph (line 1), in order to start the mapping in only one representative of each class, thus avoiding unnecessary and redundant searches. Then a series of symmetry conditions are found (line 2). The idea is to avoid symmetries by adding constraints on the labeling of the vertices. This is done by going through all equivalence classes and then imposing the condition that the label of one of its vertices is smaller than the minimum label of the others. This is done by successive calculations of the automorphisms of the graph (more details on [GK07]).

CHAPTER 3. ALGORITHMS FOR MOTIF DISCOVERY

Figure 3.5 gives an example of the conditions found in a graph with 6 vertices. Vertices in white are fixed by any automorphism preserving the indicated conditions. Other vertices are coloured accordingly to their equivalence class.

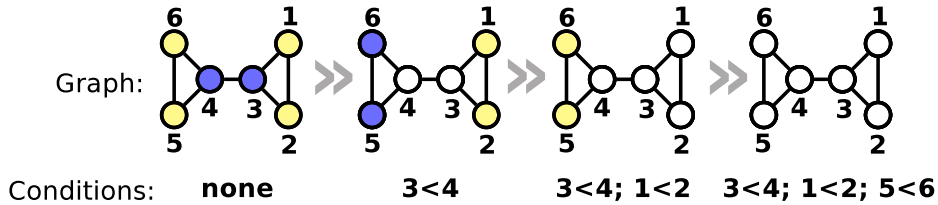


Figure 3.5 – Finding symmetry conditions on a graph with 6 vertices. Adapted from [GK07].

After doing all of this, the algorithm starts by trying to match every vertex g of the graph G into one of the vertices h representing each equivalence class of the query graph (lines 4 and 5). The vertices g are searched in order of their degree (line 3) to impose more constraints on the possible candidates. When one of the h vertices is a suitable candidate (line 6), that is, a possible match in what regards to its degree and neighbour degrees, the algorithm continues recursively mapping more vertices to see if the whole query graph can be mapped (line 8).

This recursion is handled by the `isomorphicExtensions()` procedure whose goal is to find all isomorphic extensions of a partial map $f:H \rightarrow G$, satisfying conditions C (line 10). In order to do this, the most constrained neighbour of g is tried (line 15), that is, the one with theoretically fewest possible candidates (by degree, mapped neighbours, etc). If a candidate vertex does not violate the already calculated symmetry conditions and does not induce a contradictory neighbourhood (line 17) then we add it to the mapping (line 18) and continue recursively (line 19) until the whole query graph is found (line 12 and 13). When we reach this stage, we have already addressed the isomorphism problem and we know that the found subgraph corresponds to the query subgraph.

Figure 3.6 exemplifies the search tree of `Grochow` when looking for a specific 3-subgraph in a network of size 3. The internal nodes indicate the current mapping of nodes to the query subgraph. $\{v_1, v_2, v_3\}$ indicates a mapping of vertices v_1, v_2, v_3 respectively to the vertices $\{A, B, C\}$. The underscore ($_$) identifies a subgraph node still not mapped.

In order to do an exhaustive census, `Grochow` uses McKay’s `gtools` package [McK98] to generate all possible subgraphs of a determined size and then runs the single query search to determine their frequency. Another way of using this algorithm is to include

3.2. SEQUENTIAL EXACT CENSUS

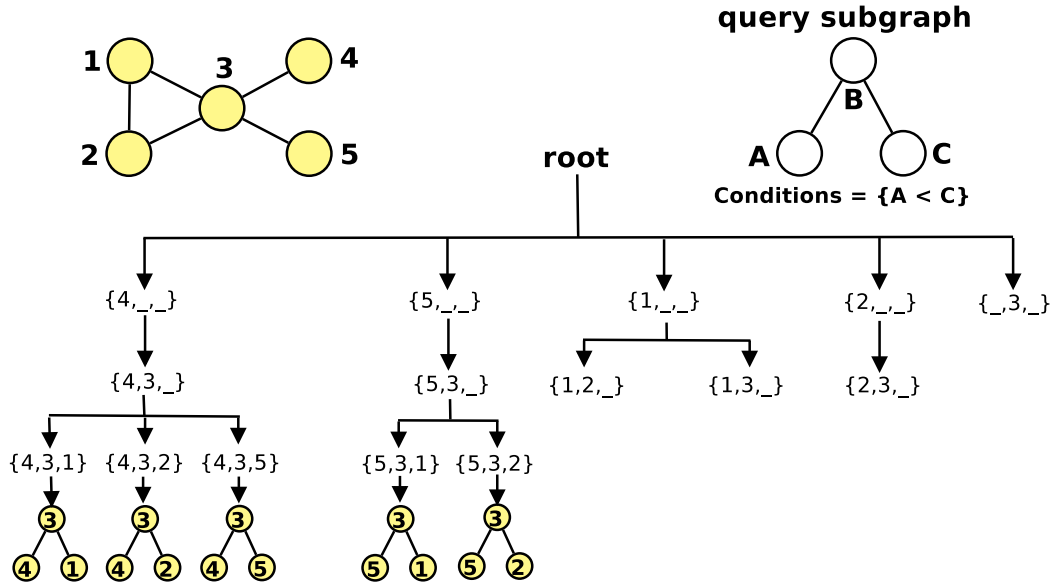


Figure 3.6 – Search tree of Grochow algorithm looking for a 3-subgraph.

a direct computation of whether a specific subgraph is a motif. This subgraph can be a larger than normal randomly sampled subgraph, which would have a prohibitive computational cost with the previous methods.

3.2.4 Kavosh Algorithm

Kavosh appeared in 2009 and, like some of the previous methods, it is network centric. Its core idea is to find all subgraphs that include a particular vertex, then remove that vertex and continue from there iteratively. It differs from other algorithms in that it builds an implicit tree rooted at the chosen vertex, and then generates all combinations with the desired number of nodes. Algorithm 3.5 details Kavosh.

Kavosh starts by choosing in turn all vertices as the root (line 3) and continues by enumerating all subgraphs containing that vertex (line 4). This is done using the recursive procedure `enumerateVertex()`, which starts by adding all unvisited neighbors of the partially constructed subgraph (function `neighbors`) to `list`. Note that if we are at the i -th call of `enumerateVertex()`, then this list contains vertices at distance i of the root. After this step, Kavosh iterates through all possible combinations of this list that can give origin to a subgraph of the chosen size (lines 10 to 14). For example, if there are 3 vertices in the list and the subgraph needs 3 more vertices, then one can choose 1 vertex (and let the other 2 come after), or choose 2 vertices (and let the other 1 come after) or choose all vertices (and the subgraph is complete).

CHAPTER 3. ALGORITHMS FOR MOTIF DISCOVERY

The combinations (functions `initialComb()` and `nextComb()`) are done using a “revolving door algorithm” [KS99]. As in ESU algorithm, the isomorphism detection is done using `nauty` [McK81].

Figure 3.7 illustrates this combinatorial search. It shows how **Kavosh** finds all 4-subgraphs with vertex 1 as root. The images show the constructed implicit tree and the possible combinations of nodes at every level.

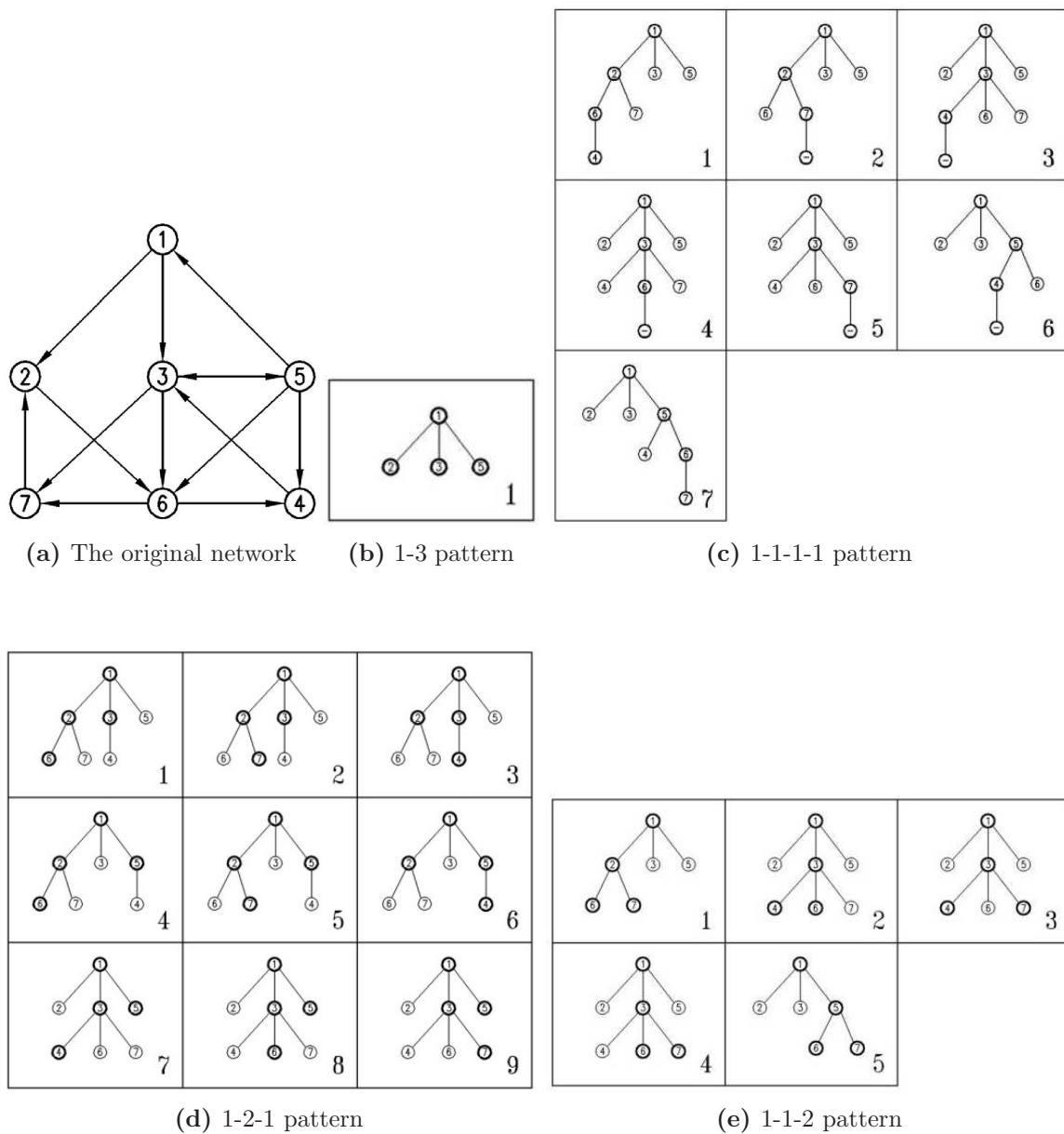


Figure 3.7 – Kavosh combinatorial search tree. Taken from [KAE⁺09].

Algorithm 3.5 Kavosh enumeration of subgraphs.

Require: Graph G and positive integer k

Ensure: k -subgraphs census of graph G

```

1: for all  $u \in V(G)$  do
2:   visited[ $u$ ] := true
3:    $S_0 := u$ 
4:   ENUMERATEVERTEX( $u, S, k - 1, 1$ )
5:   visited[ $u$ ] := false

6: procedure ENUMERATEVERTEX( $u, S, remainder, i$ )
7:   if  $remainder = 0$  then
8:     INCREMENTCOUNT(canonicalLabeling( $S$ ))
9:   else
10:    list := NEIGHBOURS( $S_{i-1}, u$ )
11:     $n_i := \text{MINIMUM}(|list|, remainder)$ 
12:    for  $k_i = 1$  to  $n_i$  do
13:       $C := \text{INITIALCOMB}(list, k_i)$ 
14:      repeat
15:         $S_i := C$ 
16:        ENUMERATEVERTEX( $u, S, remainder - k_i, i + 1$ )
17:        NEXTCOMB( $list, k_i$ )
18:      until  $C = \emptyset$ 
19:      for all  $v \in list$  do
20:        visited[ $v$ ] := false

21: function NEIGHBORS( $parents, u$ )
22:   list =  $\emptyset$ 
23:   for all  $v \in parents$  do
24:     for all  $w \in N(v)$  do
25:       if  $u < w$  AND visited[ $w$ ] = false then
26:         visited[ $w$ ] := true
27:         list := list +  $w$ 

```

3.2.5 MODA Algorithm

MODA appeared in 2009 and is a subgraph-centric approach based on a pattern growth methodology. Its core idea is to first query the frequency of subgraphs which are trees, store the respective mapping in memory and then use those mappings in order to speedup subsequent queries of non-tree subgraphs, by starting with the already computed mappings.

In order to do this, MODA introduces the concept of expansion trees for a determined graph size k , known as T_k . Each node of this tree is a graph that will be queried in the original network. In the first depth level there are all possible trees with k vertices. In the following level there are all possible expansions by adding one edge, that is, if you remove an edge, you would get a tree. This continues iteratively by adding one edge at a time for each increased tree depth, until a completely connected graph is obtained. Figure 3.8 illustrates T_4 , the expansion tree of 4-graphs.

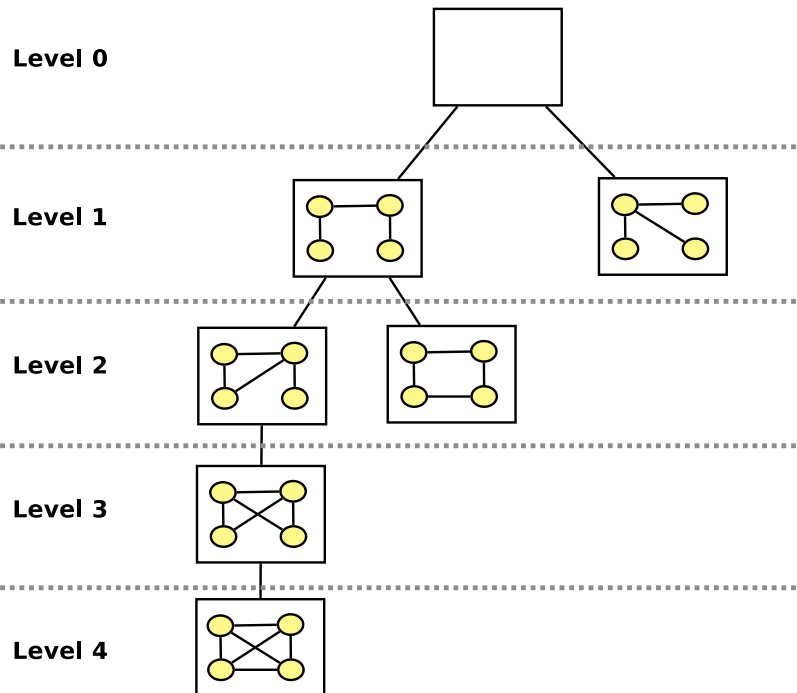


Figure 3.8 – The expansion tree of 4-graphs. Adapted from [OSMN09]

After computing this tree for the desired subgraph size (note that this is a statical data structure, that can be computed once and reused for all future queries of that size) MODA works by basically querying all subgraphs of level 1 using Grochow and Kellis' algorithm. Then, it follows by querying all subgraphs of level 2, using the previous results to accelerate the search. It continues by increasing one level at a

3.2. SEQUENTIAL EXACT CENSUS

time, and assumes that the mappings and frequencies of the previous level are already computed. This algorithm is detailed in Figure 3.6.

Algorithm 3.6 MODA enumeration of undirected subgraphs.

Require: Graph G and positive integer k

Ensure: k -subgraphs census of graph G

```

1:  $T_k :=$  expansion tree  $k$ -subgraphs
2: repeat
3:    $G' := \text{GETNEXTBFS}(T_k)$ 
4:   if ( then  $|E(G')| = (k - 1)$  ) ▷ Is it a tree?
5:      $\text{MAPPINGS}(G')$ 
6:   else
7:      $\text{ENUMERATE}(G')$ 
8: until  $|E(G')| = (k - 1)/2$  ▷ Is it a complete subgraph?

9: procedure  $\text{ENUMERATE}(G')$ 
10:   $H := \text{PARENT}(G', T_k)$ 
11:   $F_H :=$  from memory
12:   $(u, v) := E(G') - E(H)$ 
13:  for all  $f \in F_H$  do
14:    if  $(f(u), f(v)) \in G$  then
15:      Add  $f$  into  $F_{G'}$ 

16: procedure  $\text{MAPPINGS}(G')$ 
17:   $\text{GROCHOWKELLIS}(G', G)$ 
18:  add found mappings to  $F_{G'}$ 

```

The algorithm starts by constructing the corresponding expansion tree (line 1) and then traverses this tree in a breadth-first search way (line 3). At each step, if the selected node of the expansion tree is a tree graph, it calls the `grochowKellis` procedure (lines 4 and 5). If it is not, it calls a specialized procedure (`enumerate()`) that essentially loads all mappings of the parent node (lines 10 and 11) and then checks if the new edge of the expansion tree is also present in the original network for all possible mappings (lines 12 to 15)

One problem with this algorithm is the huge memory cost. Every possible instance found in the network of all the k -subgraphs will eventually end up being stored in memory. This algorithm could be adapted for directed graphs, which would exacerbate even more the memory usage.

3.3 Sequential Approximation Census

The problem with doing the complete census is that the number of existing subgraphs grows super exponentially as we increase the size of the network or the size of the subgraphs themselves. One way to cope with that growth is to sacrifice accuracy, using a probabilistic approximation algorithm: instead of fully enumerating all the subgraphs, we can sample a determined number of k -subgraphs in the original and in the random networks. We can then use their concentration to obtain an approximated z-score and therefore calculate an approximate significance, that will be more accurate as we increase the number of samples.

3.3.1 Sampling Subgraphs - Kashtan algorithm

The first time sampling was used in the motifs realm was in 2004 [KIMA04a]. The main idea is to have an algorithm for retrieving a sample subgraph. Then, we can repeat the procedure as many times as we need in order to obtain a sample of the whole set of existing subgraphs. The method for sampling one subgraph is detailed in Figure 3.7.

Algorithm 3.7 Kashtan method for sampling one subgraph

Require: Graph G and positive integer k

Ensure: One sample k -subgraph of G and probability of obtaining it

- 1: Pick random edge $(i, j) \in E(k)$
 - 2: $S := \{i, j\}$
 - 3: **while** $|S| \neq k$ **do**
 - 4: Pick random $v \in N(S) : v \notin S$
 - 5: $S := S \cup v$
 - 6: $P :=$ Calculate probability to sample S
 - 7: **return** S, P
-

To sample one subgraph, *Kashtan* chooses a random starting edge (line 1) and continues adding arbitrary new vertices that are on the neighbourhood of the partially constructed subgraph (lines 4 to 5) until the desired subgraph size is achieved (line 3). One problem is that this method is clearly biased because not all subgraphs have the same probability of being sampled [KIMA04a]. To account for that, the sampling also calculates the probability P of this graph being chosen and then assigns the sample a

3.3. SEQUENTIAL APPROXIMATION CENSUS

weight of $W = P^{-1}$. With all this, an approximate census is a matter of calling the algorithm the desired number of times.

3.3.2 Rand-ESU Algorithm

The ESU algorithm described in section 3.2.2 includes an option to take an uniform sample of the whole subgraph census, as described in Figure 3.8.

Algorithm 3.8 Rand-ESU sampling of subgraphs

Require: Graph G , positive integer k and set of probabilities P_d

Ensure: Uniformly sampled k -subgraphs

```

1: for all  $v \in V(G)$  do
2:    $V_{Ext} := \{u \in N(v) : u > v\}$ 
3:   with probability  $P_1$  EXTENDSUBGRAPH( $\{v\}, V_{Ext}, v$ ) ▷ NEW CODE
4: procedure EXTENDSUBGRAPH( $V_{Subg}, V_{Ext}, v$ )
5:   if  $|V_{Subg}| = k$  then
6:     INCREMENTCOUNT(canonicalLabeling( $V_{Subg}$ ))
7:   else
8:     while  $V_{Ext} \neq \emptyset$  do
9:       remove random chosen  $w \in V_{Ext}$ 
10:       $V'_{ext} := V_{ext} \cup \{u \in N_{excl}(w, V_{subg}) : u > v\}$ 
11:       $V'_{Subg} := V_{Subg} \cup \{w\}$  ▷ NEW CODE
12:      with prob.  $P_{|V'_{Subg}|}$  EXTENDSUBGRAPH( $V'_{Subg}, V'_{Ext}, v$ ) ▷ NEW CODE

```

This is essentially the same algorithm of ESU (Figure 3.3), with the exception of the lines indicated with the NEW comment. Therefore, the same base algorithm to enumerate all k -subgraphs is used, but each recursive call is made only with a certain probability P_d , associated to the depth of the enumeration. Since each subgraph appears once and only once in the search subtree, all the subgraphs have the exact same probability of being called. More than that, we know that all subgraph samples are different from each other, while in **Kashtan** there is no such guarantee. On the other hand, we cannot exactly generate a fixed number of samples.

Since we are dealing with probabilities, we can only choose the values of P_d in order to have an approximated number of sampled subgraphs: if we want to have a fraction $0 < q < 1$ of the subgraphs samples, then we must guarantee that $\prod P_d = q$. This still leaves the open question of how to choose the individual P_d values. The general

CHAPTER 3. ALGORITHMS FOR MOTIF DISCOVERY

advice given in [Wer06] is to have larger values for small d , since if we discard an entire search subtree near the root, a whole lot of subgraphs will not have the possibility to be sampled. However, it must be noted that the larger these values are, then the more time the sampling will take, since we will have to branch into more subtrees. Note that using the value of 1 for all probabilities will result in an exact census being computed.

An empirical comparison of the **Rand-ESU** and **Kashtan** methods can be seen in [JHQ09].

3.3.3 MODA Sampling

In the MODA algorithm (described in Section 3.2.5), the **grochowKellis()** procedure is used for a small portion of the subgraphs in the expansion tree (only level 1), but even so this part is the bottleneck of the computation and takes a considerable amount of time. In order to speed up this part of the algorithm, a sampling version of this particular procedure was added to MODA, which is described in figure 3.9.

Algorithm 3.9 MODA sampling version of mapping procedure

Require: Graph G , Query tree subgraph G'

Ensure: Approximate frequency and mapping of G'

```

1: procedure MAPPINGSSAMPLE( $G'$ )
2:   for  $i = 1$  to number_sample_vertices do
3:      $v :=$  select node from  $V(G)$  with probability proportional to  $\text{degree}(v)$ 
4:     for all  $u \in G'$  do
5:       if  $\text{degree}(v) \geq \text{degree}(u)$  then grochowKellis( $G'$ ) with  $f(u) = v$ 
6:     remove  $v$  from  $V(G)$ 
```

Instead of searching all possible mappings, this sampling procedure only finds those which are based on sampled root vertices. Note that if *number_sample_vertices* is equal to $|V(G)|$ it would mean that we would have a complete and exact search. Choosing vertices at random with a uniform distribution of $V(G)$ did not empirically produce reliable results, i.e., large fluctuations of the results were obtained. Therefore, MODA uses a probability distribution proportional to the degree of the nodes, based around the notion that subgraphs tend to aggregate around the higher degree nodes [VDS⁺04]. This lead to more stable results.

3.4 Determining the Significance

In order to determine the significance of a subgraph, an ensemble of networks similar to the original network is created and a subgraph census is made on each of them, as explained in Section 3.1. These networks should be simple and have the same degree sequence as the original one, that is, all nodes should maintain the same in and out degrees, as was exemplified in Figure 2.6.

The standard procedure is to use a Markov-Chain method, as detailed in Figure 3.10. Starting with the original graph (line 1), a pair of edges $a \rightarrow b$, $c \rightarrow d$ is repeatedly swapped by $a \rightarrow d$, $b \rightarrow c$ in order to preserve vertex in and out degree (lines 3 to 6). The degree of randomness (line 2) is controlled by the user. Care is taken so that no self-loops or multiple edges are introduced (line 4).

Algorithm 3.10 Markov-Chain for creating similar randomized graphs

Require: Graph G

Ensure: Random Graph similar to G

```

1:  $R := G$ 
2: while NOTWELLRANDOMIZED( $R$ ) do
3:   Choose arbitrary  $(a, b), (c, d) \in E(R)$ 
4:   if  $(a, d) \notin E(R) \wedge (c, b) \notin E(R) \wedge a \neq d \wedge b \neq c$  then
5:     Remove  $(a, b), (c, d)$  from  $E(R)$ 
6:     Add  $(a, d), (c, b)$  to  $E(R)$ 
return  $R$ 
```

Another not so common approach is to directly generate the random network from scratch, which Milo et al. [MSOI⁺02] adapted from Newman et al. [NSW01]. Figure 3.11 details this procedure. It starts with an empty graph, represented by its adjacency matrix (line 1) and repeatedly adds random connections with probabilities related to the number of connections that still must be made on each vertex (lines 4 to 8). This continues until all vertices have the desired number of connections (line 3).

It should also be noted that the ensemble of random networks could be avoided if analytical methods to derive the desired probability for a subgraph were available. Recent work pursues this line of research [MSB⁺06, Wer06, PDK⁺08, SLS08], but these theoretical statistical approaches still require further development to reach the accuracy and generality needed for a practical application. In any case, they would not avoid the need to compute the census in the original network.

CHAPTER 3. ALGORITHMS FOR MOTIF DISCOVERY

Algorithm 3.11 Direct method for creating similar randomized graphs

Require: Graph G

Ensure: Random Graph similar to G

```
1:  $R :=$  empty graph
2:  $R_i := \sum_j G_{Adj}[i][j]$ 
3:  $C_j := \sum_i G_{Adj}[i][j]$ 
4: while  $\exists R_i > 0$  do
5:    $m :=$  arbitrary row with probability  $R_m / \sum R_i$ 
6:    $n :=$  arbitrary col with probability  $C_n / \sum C_i$ 
7:   if  $R_{Adj}[m][n] \neq 0$  then
8:      $R_{Adj}[m][n] := 1$ 
9:      $R_i := R_i - 1$ 
10:     $C_j := C_j - 1$ 
```

3.5 Parallel Algorithms

Research work on parallel algorithms for motifs discovery is still very scarce. Specific to the motifs discovery problem and its associated subproblems, there are only two distinct implemented and studied parallel approaches available [WTZ⁺05, SCBB08]. Schreiber and Schwobbermeyer [SS04] do refer they were working on implementing a parallel version of their algorithm, but they defer the scalability analysis and further studies to possible future work.

These two parallel approaches were not subject to detailed and extensive scalability analysis and focus essentially on parallelizing a single complete census. Parallelism on the whole motifs discovery problem can then only be achieved by calculating consecutively the parallel census of the original and random networks, making it necessary to have synchronization after each census. The process of randomly generating a set of similar networks is also done sequentially.

3.5.1 Wang et al.

Wang et al. [WTZ⁺05] rely on finding neighborhood assignments for each vertex that avoid overlapping and redundancy on subgraph counts (as in the ESU algorithm), and try to statically balance the workload before the computation begins using mostly the node degrees. However, they do not detail the static scheduling process and they

do not study the scalability of their approach, limiting the empirical analysis to a single network (the *E. coli transcriptional* regulation network), and a fixed number of processors (32). The obtained speedup was not linear. Another characteristic of their approach is that they do not do isomorphism tests during the parallel computation, that is, they wait until the end to check all the subgraphs and compute the corresponding isomorphic classes.

3.5.2 Schatz et al.

Schatz et al. [SCBB08] focuses instead on parallelizing Grochow and Kellis [GK07] approach. They do different single subgraph queries at the same time on different processors in a master-worker strategy, experimenting with static pre-computed scheduling (the same number of queries for each processor) and first-fit scheduling (meaning that workers only process one query at a time and when they finish it they ask the master for more work). The later was shown to have almost linear speedup against the corresponding sequential algorithm up to 64 processors, although it should be noticed that they only used one experimental network and in terms of motif discovery, following the Grochow and Kellis [GK07] approach, they would have to query the entire set of possible k -subgraphs, even when some of those subgraphs may not even appear on the networks.

They have also tried to parallelize a single query using a network partition algorithm that creates small overlapping regions, adding some overhead for possibly repeated computations. This lead to some speedup, but again it was only tested on a single network and for eight different 7-subgraph queries.

3.6 Summary

This chapter has an extensive review of the state of the art in motif discovery. Being a relatively young field, a walk trough all algorithmic innovations was done, and a time line and comparison table was shown. Then, all main sequential algorithms were described in more detail, including pseudo-code. Care was taken to describe exact and approximate algorithms, as well as how similar random networks can be generated. A look at the existing parallel approaches was also included.

*Get your data structures correct first, and
the rest of the program will write itself.*

David Jones

4

The G-Trie Data Structure

The algorithms mentioned in the previous chapter still present serious limitations on the maximum feasible network and motif sizes. They follow one of two extreme approaches: either all subgraphs of a given size are enumerated and then checked for isomorphism, or individual subgraphs are queried on the desired network. There was no intermediate approach that would enable us to query sets of subgraphs: not necessarily all, but also not just one subgraph. In order to follow this path, a novel data structure was created, the g-tries, that is the subject of this chapter. G-tries enable new algorithms with great potential for efficiency gains. We start by presenting the inspiration and motivation behind it and follow with a more concise definition. We then describe in detail the associated algorithms and how they can be used to discover motifs.

4.1 Motivation and Prefix Trees

When analyzing the previous motif discovery methods, some common drawbacks were identified. All network-centric methods start by first computing a complete census of the respective network, regardless of it being the original or a similar random network. However, the random networks can contain more types of subgraphs than the original one. And we can know beforehand that specific subgraph classes can not possibly be motifs (for example because they do not present the minimum desired frequency). This means that by doing a complete census on the random networks, the operation

CHAPTER 4. THE G-TRIE DATA STRUCTURE

that is the bottleneck of motif computation, since this ensemble has at least dozens of networks, we will be doing a significant proportion of unnecessary work, by discovering the frequency of subgraphs that are not interesting from the point of view of motifs calculation.

Subgraph-centric methods appeared later and were able to give a workaround to this problem. However, they work by finding in turn the frequency of individual subgraphs. This means that there will be redundant work, in the sense that the same nodes will have to be traversed for every single interesting subgraph class. Two subgraphs that are exactly the same with the exception of a single node, will be computed extensively and separately without taking advantage of that similarity.

The main idea for a new methodology for discovering motifs is therefore to explore these aspects to gain computational efficiency. We wanted an algorithm that is specialized in finding a set of subgraphs, that is not necessarily all possible subgraphs, but also not just one single subgraph. This is the core of what motif algorithms are really doing on the random network census computation, which constitutes the bottleneck of the whole motif discovery process.

In many ways, what scientists are now doing with graphs is comparable to what was done in sequences. Indeed, the word motif is also recurrent in sequence analysis, particularly in DNA analysis, meaning a sequence that is interesting because it is statistically over-represented. Metaphorically, network-centric methods are the equivalent of searching all possible words of a determined size that exist in a word. In contrast, subgraph-centric methods are equivalent to searching one word individually, and then start over the search for each new word query, without reusing anything from past queries.

In sequences, if we want a data structure that can store a set of words, that is, a dictionary, one has many options. One of those is the trie data structure, also known as a prefix tree [Fre60]. Introduced in 1960 by Fredkin, tries make use of common prefixes of sequences. They are basically trees, where all descendants of a node have the same common prefix, as illustrated in Figure 4.1. Note how common prefixes are aggregated in the same nodes.

Algorithmically, tries can be considered an efficient structure. They provide linear execution time for verifying if a word of size n is in the set. Basically we can just descend the trie, one letter at a time. In memory terms they also provide big saves when comparing to actually storing all the words, because common prefixes are only stored once, avoiding redundancy.

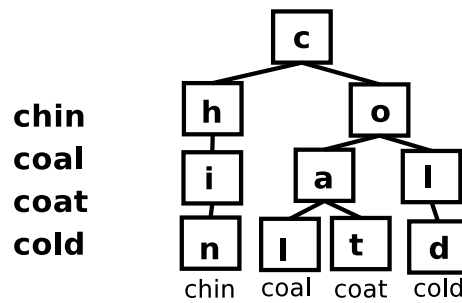


Figure 4.1 – A trie representing a set of four words.

One of the possible usages of a trie is to discover instances of all the words contained in the trie. When we have a partial match of a word, we know exactly which words can be formed from that particular subsequence. In that sense we do not need to do the redundant work of searching again for the same prefix if we would start to look for a new word disregarding previous searches. More than that, at a certain point, we could know that there is no possible word of a determined size starting with a certain prefix, since there are no descendant nodes, and we could stop the computation on that search branch.

The core idea for the new g-trie data structure is to take advantage of these conceptual advantages and apply them in the graphs realm, as we show in the next section.

4.2 G-Tries Definition

A trie takes advantage of common prefixes. By analogy, g-tries take advantage of common substructures in a collection of graphs. In the same way two or more strings can share the same prefix, two or more graphs can share a common smaller subgraph. Figure 4.2 exemplifies this. The five graphs all share a common strongly connected 3-subgraph indicated by the light vertices and thick edges.

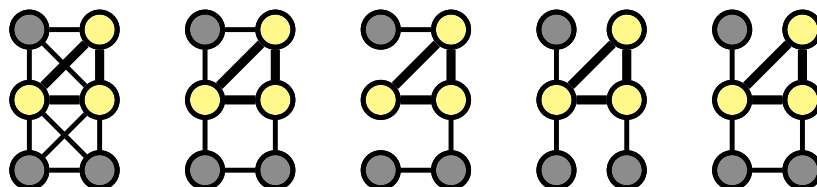


Figure 4.2 – Common substructures in graphs.

Like tries, g-tries are trees. Each trie node has a single letter and each g-trie node

CHAPTER 4. THE G-TRIE DATA STRUCTURE

will represent a single graph vertex. Each vertex is characterized by its connections to the respective ancestor nodes. This can be visualized in Figure 4.3. Each tree node adds a new vertex (in black) to the already existing ones in the ancestor nodes (light vertices).

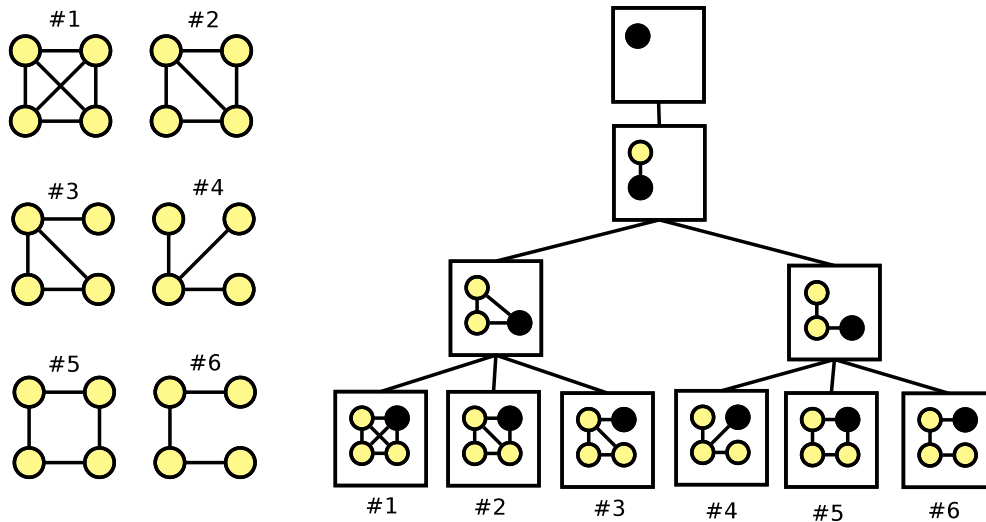


Figure 4.3 – A g-trie representing a set of 6 undirected graphs.

Note that all graphs with common ancestor tree nodes share common substructures that are characterized precisely by those ancestor nodes. A single path through the tree corresponds to a different single graph. Children of a node correspond to the different graph topologies that can emerge from the same subgraph. Graphs of different sizes can be stored in the same tree if each tree node also signals if it corresponds to the “end” of a graph. All of this is easily generalizable to directed subgraphs (see Figure 4.4), and also to colored graphs.

We call these kind of trees as **g-tries**, following the etymology “**G**raph **re****TRIE**val”. We now give an informal definition of this abstract data structure. Note that a multiway tree is a tree with a variable number of children per tree node.

Definition 4.1 (G-Trie) *A g-trie is a multiway tree that can store a collection of graphs. Each tree node contains information about a single graph vertex, its corresponding edges to ancestor nodes and a boolean flag indicating if that node is the last vertex of a graph. A path from the root to any g-trie node corresponds to one single distinct graph. Descendants of a g-trie node share a common subgraph.*

In order to avoid further ambiguities, throughout this chapter we will use the term *nodes* for the g-trie tree nodes, and *vertices* for the graph network nodes.

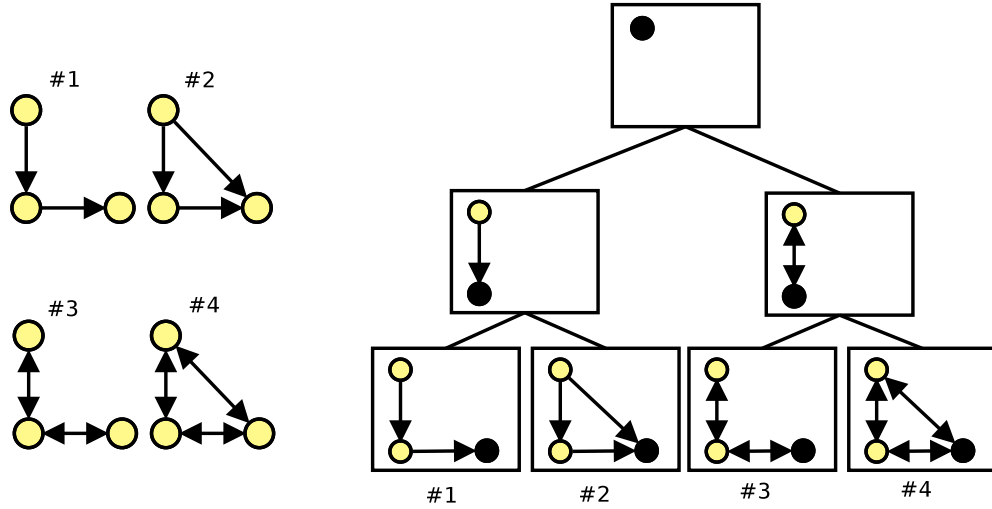


Figure 4.4 – A g-trie representing a set of 4 directed graphs.

4.3 A G-Trie Implementation

For the purposes of this work we need to implement g-tries efficiently and with the ability to support undirected and directed graphs. As we have seen, a g-trie is a tree where in each node we need to store two different types of data:

- The connections of the new vertex to the already existing vertices in the ancestor nodes. We will use two variables: `in` represents the ingoing edges and `out` the outgoing edges. In an undirected subgraph basically `in` is the same as `out` and can be disregarded if one wants to save memory.
- A flag indicating if the path from the root to that node corresponds to a graph that is stored on that g-trie. We will call `isGraph` to this g-trie node variable.

The flag is implemented as a boolean variable. There are however several options to store the connections. For the first practical implementation we chose the equivalent of the adjacency matrix, a simple yet effective option. For illustration purposes, we will use the standard 0-1 way of representing a cell of the adjacency matrix ('1' indicates a connection and '0' its absence).

Given this, on any node, `out` stores the adjacency matrix row up to that vertex and `in` stores the adjacency matrix column up to that vertex. In any case, given a path from the root to a node, we have a fully specified graph. Note that the g-trie root node must be empty since there are two possible direct child nodes: a vertex with

CHAPTER 4. THE G-TRIE DATA STRUCTURE

or without a connection to itself. In this way, g-tries are also able to accommodate self-loops.

Figure 4.5 exemplifies this implementation. White g-trie nodes have the boolean `isGraph` variable set to false and gray g-trie nodes have it set to true, the later meaning that a path to that node represents a graph in the set. Since we are dealing with undirected graphs, we only show the contents of the `out` variable, represented by a sequence of 0-1 numbers indicating the corresponding adjacency matrix row. Exemplifying, if the sequence starts by '1' it means that the new vertex is connected to the first vertex of the graph. If it is '0', it means it is not connected to that vertex. The adjacency matrix of the undirected graphs is a triangle matrix formed by the sequence of `out` variables of each node in the path from the root to its end node.

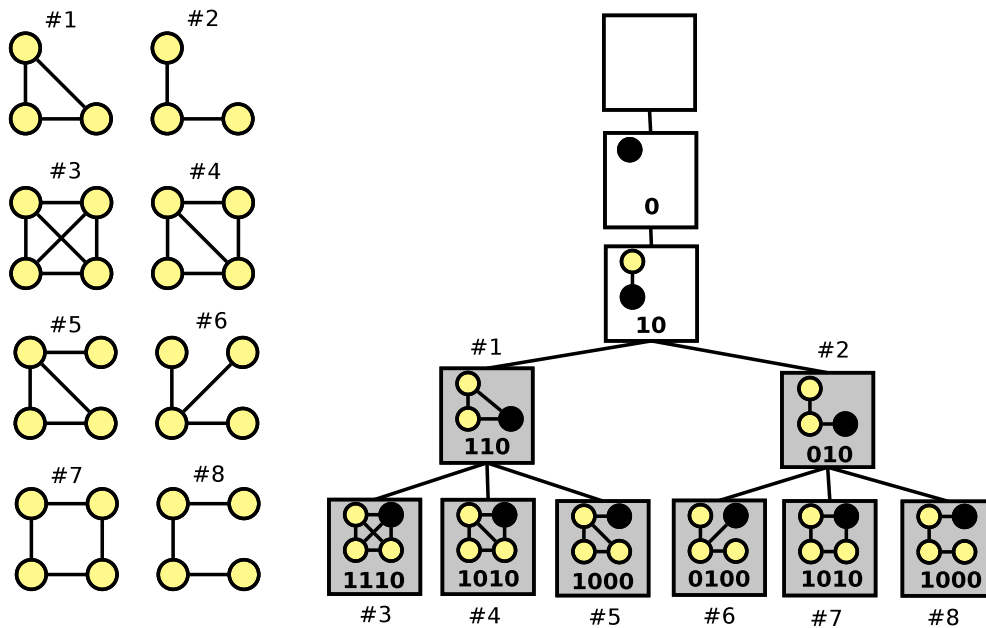


Figure 4.5 – An implementation of g-tries using adjacency matrices.

4.4 Creating a G-Trie

The first task one must be able to do in order to use g-tries is of course to be able to create one. The following sections will show how we do this.

4.4.1 Iterative Insertion

In order to construct a g-trie, we just repeatedly insert one subgraph at a time, starting with an empty tree (just a root node). Each time, we traverse the tree and verify if any of the children has the same connections to previous nodes as the graph we are inserting. With each increase in depth we also increase the index of the vertex we are considering.

Figure 4.6 exemplifies this process. The g-trie tree node squares in gray are the new ones after each insertion and node squares in white are the ones that remain from before. Squares with dashed lines represent the actual g-trie implementation with adjacency matrices, and squares with normal lines give the correspondent visual representation. Black vertices indicate new vertices, while the old ones are white. Nodes with the `isGraph` variable set to true are also indicated with the subtext “*isGraph*”.

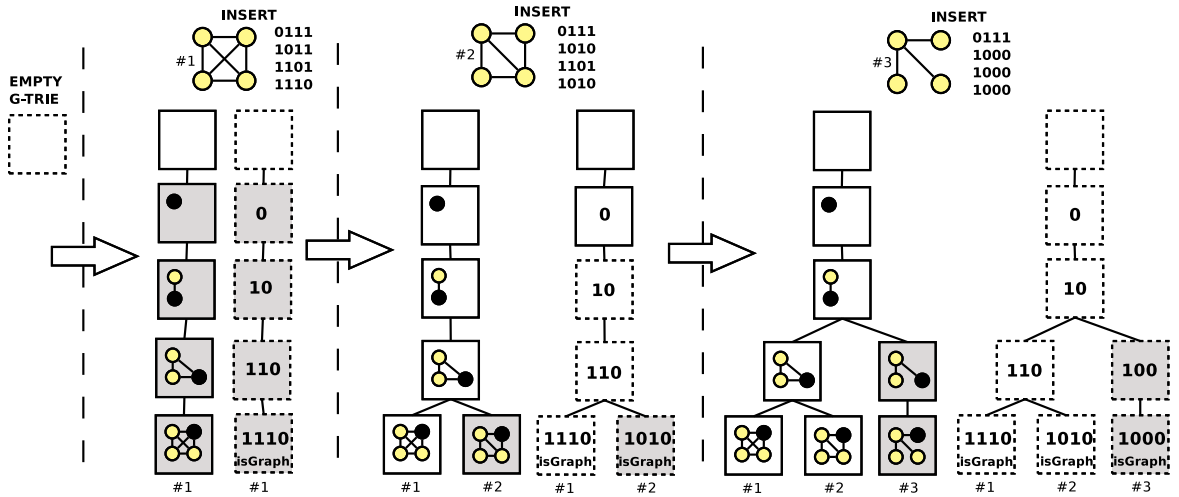


Figure 4.6 – Sequential insertion of 3 graphs on an initially empty g-trie.

CHAPTER 4. THE G-TRIE DATA STRUCTURE

4.4.2 Canonical Representation of Graphs

The same graph can be represented by different adjacency matrix representations. In this section we will first explain why we need to use a canonical representation and will then describe our custom canonical form.

4.4.2.1 The Need for a Canonical Form

Following the described insertion procedure, the insertion is completely defined by the adjacency matrix of the inserted graph. However, there are many different possible adjacency matrices representing the same class of isomorphic graphs. This is exemplified in Figure 4.7. Note how the labeling of the vertices affects the correspondent matrix.

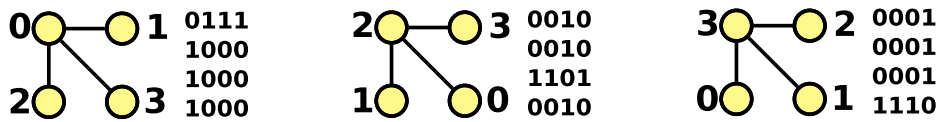


Figure 4.7 – Three different adjacency matrices representing the same graph.







The problem with this is that different matrices will give origin to different g-tries. We could even have two isomorphic graphs having different g-trie representations, leading to different branches of the tree representing the same graph, which would contradict the purpose of the g-trie. In order to cope with that we must use a canonical labeling, which guarantees that isomorphic graphs will always produce the same univocal adjacency matrix, and therefore the same set of subgraphs is guaranteed to produce the same g-trie.

4.4.2.2 Impact on the G-Trie Structure and Compression Ratio

There are many possible canonical representations, and the representation used directly and significantly impacts the g-trie structure. In order to illustrate this, consider the string formed by the concatenation of all adjacency matrix rows, and call it *adjacency string*. Any choice of canonical representation will give origin to different adjacency strings. Two possible options of forming a canonical adjacency string would be to consider the *lexicographically larger* or the *lexicographically smaller* one for each graph.

4.4. CREATING A G-TRIE

Figure 4.8 illustrates the g-tries generated for each of these possible choices for the same set of six 4-graphs. Note the contrast between these two choices, with completely different structures of the g-trie formed. One can clearly observe a variation on the number of g-tries nodes needed and a different balance on the nodes of each size of the g-trie.

Graph	#1	#2	#3	#4	#5	#6
						
lexicographically larger	0111 1011 1101 1110	0111 1011 1100 1100	0111 1010 1100 1000	0111 1000 1000 1000	0110 1001 1001 0110	0110 1001 1000 0100
lexicographically smaller	0111 1011 1101 1110	0011 0011 1101 1110	0001 0011 0101 1110	0001 0001 0001 1110	0011 0011 1100 1100	0001 0010 0101 1010

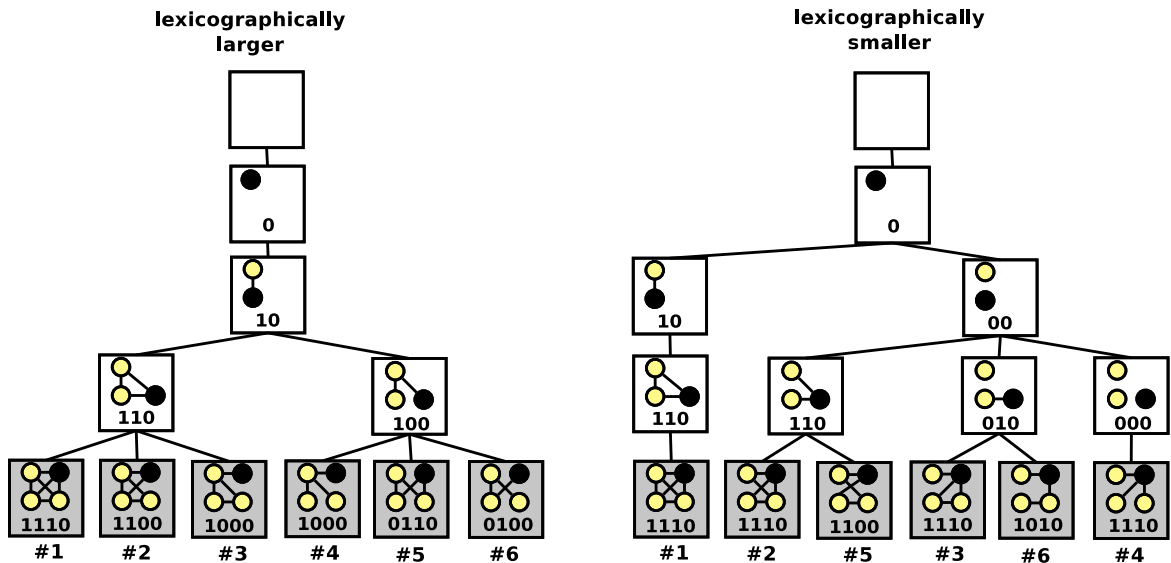


Figure 4.8 – Two different g-tries using lexicographically larger and smaller adjacency strings.

If we increase the amount of common ancestor topologies, we decrease the size of the tree and effectively we compress the representation, needing less memory to represent it than when we had the original set of subgraphs (represented by their adjacency matrices). We can measure the amount of compression if we take into account the number of nodes in the tree and the number of vertices in the subgraphs (equation (4.1)). By using a tree we do have to spend some auxiliary memory to

CHAPTER 4. THE G-TRIE DATA STRUCTURE

represent the tree edges, but the total memory needed for the tree structure is very small compared to the actual information stored in the nodes (the graph adjacency matrix) and loses relative weight as we increase the amount of subgraphs and their size. Hence, the real memory bottleneck is in the storage of node values, and equation (4.1) is a good indicator of how much space we save and how much common substructure we identified.

$$\text{compression ratio} = 1 - \frac{\text{nodes in tree}}{\sum \text{nodes of stored graphs}} \quad (4.1)$$

As an example, the two g-tries constructed in figure 4.8 have a compression ratio of respectively $58.34\% = 1 - 10/24$ (lexicographically larger) and $45.84\% = 1 - 13/24$ (lexicographically smaller). We can ignore the root, since it uses constant memory space and only exists as a placeholder for the initial children representing the first vertex. A tree with no common topologies would need a node for each graph vertex and would have a 0% compression ratio.

4.4.2.3 An Efficient Custom Built Canonical Form

The lexicographically largest adjacency string seems like a very good candidate for the canonical representation and in fact it was the first we experimented. However, it is not the only possible choice. In general, a canonical label suitable for g-tries use should observe the following properties:

- **Connectivity:** the path from the root to any given node always induces a connected subgraph.
- **Compressibility:** the g-trie should have the least possible number of nodes, that is, the one that identifies more common sub-structure and avoids redundant representations, and computations.
- **Constraining:** new subgraph vertices should have as many connections as possible to the ancestors, in order to highly constraint the choice of possible network vertices that will match with it. The worst case is a vertex not connected at all to previous vertices, which will allow any unconnected and unused network vertex to be candidate match for it.

Choosing the lexicographically largest string obeys to the first two properties but does not try to factor directly on the third. It is also very time consuming to compute.

4.4. CREATING A G-TRIE

Given this, we opted to create our own more efficient canonical representation, geared to being more efficient to compute and as constraining as possible for later use when matching the g-trie graphs as subgraphs of another larger network. Algorithm 4.1 describes our method for computing a canonical form, and we call it **GTCanon**.

Algorithm 4.1 Converting a graph to a canonical form

Require: Graph G

Ensure: Canonical Form of G

```

1: function GTCANON( $G$ )
2:    $G := \text{NAUTYLABELING}(G)$ 
3:   for all  $i \in V(G)$  do
4:      $\text{current\_degree}[i] := \text{nr ingoing+outgoing connections of } i$ 
5:      $\text{global\_degree}[i] := \text{last\_degree}[i] := \text{current\_degree}[i]$ 
6:   for  $pos : |V(G)|$  down to 1 do
7:     Choose  $u_{min}$  subject to:
8:       •  $u_{min}$  is still not labeled and is not an articulation point
9:       •  $u_{min}$  has minimum  $\text{current\_degree}$ 
10:      • In case of a tie,  $u_{min}$  has minimum  $\text{last\_degree}$ 
11:      • In case of a new tie,  $u_{min}$  has minimum  $\text{global\_degree}$ 
12:      $\text{label}[u_{min}] := pos$ 
13:      $\text{last\_degree}[] := \text{current\_degree}[]$ 
14:     update  $\text{current\_degree}[]$  removing  $u_{min}$  connections
15:   return  $\text{label}[]$ 

```

The first step of **GTCanon** is to apply any other canonical representation. In our case we use **nauty** [McK81], a proven and very efficient third-part algorithm (line 2). Then, several lookup tables are initialized with the degrees of every node of the graph. The core of the algorithm is iterative and in each step we select a new node for being labeled at the last available labeling position. The idea is to choose a node that has the minimum amount of connections as possible (lines 9 to 11), guaranteeing that it does not divide the graph in two (line 8) and then label it (line 12) and remove it from the graph. Before the next iteration, the lookup tables with degree information are updated (lines 13 and 14).

By removing a node not densely connected to the rest of the graph, we increase the number of connections in lower labeling positions, and therefore we increase constraining. By not choosing articulation points, we guarantee connectivity in the

CHAPTER 4. THE G-TRIE DATA STRUCTURE

subgraph. Finally, each time we remove a node, we get a smaller instance of the same problem, having to compute a canonical form of the graph with one less vertex. By using the same criteria on each phase for all graphs, we increase the compressibility. When our criteria does not suffice to choose a unique candidate, the fact that we first used another canonical form guarantees that **GTCanon** will also be canonical and always return the same labeling for isomorphic graphs.

Note that computing a canonical form is always a computational hard problem because solving it is at least as hard as the isomorphism problem (two graphs are isomorphic if they have the same canonical form). However, **GTCanon** takes advantage of an efficient third party algorithm, **nauty**, which is state-of-art, and uses an efficient algorithm after that computation. Computing the articulation points can be done in linear time $O(|V(g)| + |E(G)|)$ with a simple depth-first search [Tar71]. Computing and updating the three degree arrays can also be done in linear time.

Figure 4.9 illustrates **GTCanon** in action, showing a g-trie built with the algorithm and containing the 11 subgraphs of size 5 found in an electronic circuit network. The Figure was automatically created using our own g-trie drawing code.

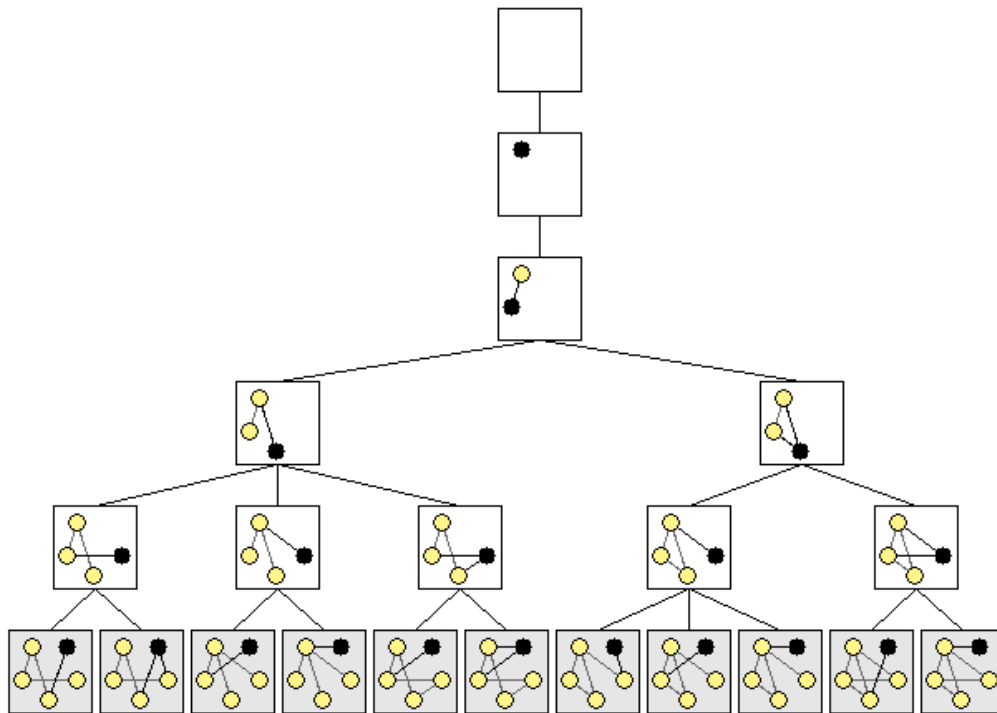


Figure 4.9 – A g-trie containing 11 undirected 5-subgraphs.

The results chapter gives more empirical verification that **GTCanon** is indeed a good choice for the labeling (see Section 6.2.2.1).

4.4.3 Insertion Algorithm

With the considerations made in the previous sections we are now ready to detail our algorithm for creating a g-trie. Algorithm 4.2 details a method to insert a single graph in a g-trie. As said, constructing a complete g-trie from a set of subgraphs can be done by inserting the graphs, one by one, into an initially empty tree.

Algorithm 4.2 Inserting a graph G in a g-trie T

Require: Graph G and G-Trie T

Ensure: Inserts graph G in G-Trie T

```

1: procedure GTRIEINSERT( $G, T$ )
2:    $M :=$  adjacency matrix of  $\text{GTCANON}(G)$ 
3:   INSERTRECURSIVE( $M, T, 0$ )
4: procedure INSERTRECURSIVE( $M, T, k$ )
5:   if  $k = \text{NUMBERROWS}(M)$  then
6:      $c.\text{isGraph} = \text{true}$ 
7:   else
8:     for all children  $c$  of  $T$  do
9:       if ( $c.\text{out} =$  first  $k + 1$  values of  $k$ -row of  $M$ ) AND
10:      ( $c.\text{in} =$  first  $k + 1$  values of  $k$ -column of  $M$ ) then
11:        INSERTRECURSIVE( $M, c, k + 1$ )
12:      return
13:    $nc :=$  new g-trie node
14:    $nc.\text{in} :=$  first  $k + 1$  values of  $k$ -row of  $M$ 
15:    $nc.\text{out} :=$  first  $k + 1$  values of  $k$ -column of  $M$ 
16:    $nc.\text{isGraph} := \text{false}$ 
17:    $T.\text{INSERTCHILD}(nc)$ 
18:   INSERTRECURSIVE( $M, nc, k + 1$ )

```

Explaining in more detail, we start by calculating the canonical adjacency matrix of the graph being inserted (line 2). Then we recursively traverse the tree, inserting new nodes when necessary, with the procedure `insertRecursive()`. This is done by going through all possible children of the current node (line 8) and checking if their stored value is equal to the correspondent part of the adjacency matrix (lines 9 and 10). If it is, we just continue recursively with the next vertex (line 11). If not, we create a new child (lines 13 to 16) and continue as before (line 18). When there are no more vertices

CHAPTER 4. THE G-TRIE DATA STRUCTURE

to process, we stop the recursion (line 5) and set the `isGraph` variable, indicating the end of the graph (line 6).

Regarding the complexity of the algorithm, `insertRecursive()` takes $O(|V(G)|^2)$, the size of the adjacency matrix. Besides this, the whole insertion needs to calculate the canonical labeling of the graph.

After constructing the g-trie, if we want to retrieve the initial set of graphs a simple depth-first search of the tree will suffice. A path from the root to any given g-trie node at depth k with `isGraph` set to true, represents a k -graph.

4.4.4 Reusing G-Tries

There are cases in which it would be very fruitful to reuse a g-trie previously created. For instance, we can pre-compute a g-trie containing all possible k -subgraphs, or we can store a g-trie containing only the subgraphs that are meaningful in the context of some subject we are studying.

For situations like these one, we created the option to serialize a g-trie, providing the ability to write and read a g-trie to the file system. For our initial implementation and in order to be able to use it in any computational environment, we used text files and compressed the information using only printable characters, that are constant for any encoding system.

The results chapter details the performance of reading and writing g-tries to the file system (see Section 6.2.1).

4.5 Computing subgraph frequencies

Once the g-trie is built, the next logical step in order to find motifs is to create a method for finding instances of the g-trie graphs as subgraphs of another larger network.

4.5.1 An Initial Approach

Algorithm 4.3 details a method for finding and counting all occurrences of the g-trie graphs as induced subgraphs of another larger graph. The main idea is to

4.5. COMPUTING SUBGRAPH FREQUENCIES

backtrack through all possible subgraphs, and at the same time do the isomorphism tests as we are constructing the candidate subgraphs. We take advantage of common substructures in the sense that at a given time we have a partial isomorphic match for several different candidate subgraphs (all the descendants).

Algorithm 4.3 Census of subgraphs of T in graph G

Require: Graph G and G-Trie T

Ensure: All occurrences of the graphs of T in G

```

1: procedure GTRIEMATCH( $T, G$ )
2:   for all children  $c$  of  $T.root$  do MATCH( $c, \emptyset$ )

3: procedure MATCH( $T, V_{used}$ )
4:    $V :=$  MATCHINGVERTICES( $T, V_{used}$ )
5:   for all node  $v$  of  $V$  do
6:     if  $T.isGraph$  then FOUNDMATCH( $V_{used} \cup \{v\}$ )
7:     for all children  $c$  of  $T$  do
8:       MATCH( $c, V_{used} \cup \{v\}$ )

9: function MATCHINGVERTICES( $T, V_{used}$ )
10:  if  $V_{used} = \emptyset$  then  $V_{cand} := V(G)$ 
11:  else
12:     $V_{conn} := \{v : v \in N(V_{used})\}$ 
13:     $m := m \in V_{conn} : \forall v \in V_{conn}, |N(m)| \leq |N(v)|$ 
14:     $V_{cand} := \{v \in N(m) : v \notin V_{used}\}$ 
15:   $Vertices := \emptyset$ 
16:  for all  $v \in V_{cand}$  do
17:    if  $\forall i \in [1..|V_{used}|]$ :
18:       $T.in[i] := G_{Adj}[V_{used}[i]][v] \wedge T.out[i] = G_{Adj}[v][V_{used}[i]]$  then
19:         $Vertices := Vertices \cup \{v\}$ 
20:  return  $Vertices$ 

```

At any stage, V_{used} represents the currently partial match of graph vertices to a g-trie path. We start with the g-trie root children nodes and call the recursive procedure `match()` with an initial empty matched set (line 2). The later procedure starts by creating a set of vertices that completely match the current g-trie node (line 4). We then traverse that set (line 5) and recursively try to expand it through all possible tree paths (lines 7 and 8). If the node corresponds to a full subgraph, then we have

CHAPTER 4. THE G-TRIE DATA STRUCTURE

found an occurrence of that subgraph (line 6). Note that at this time no isomorphic test is needed, since this was implicitly done as we were matching the vertices.

Generating the set of matching vertices is done in the `matchingVertices()` procedure. The efficiency of the algorithm heavily depends on the above mentioned constraints as they help in reducing the search space. To generate the matching set, we start by creating a set of candidates (V_{cand}). If we are at a root child, then all graph vertices are viable candidates (line 10). If not, we select from the already matched vertices that are connected to the new vertex (line 12), the one with the smallest neighborhood (line 13), thus reducing the possible candidates to the unused neighbors (line 14). Then, we traverse this set of candidates (line 16), and if one respects all connections to ancestors (lines 17 and 18) we add it to the set of matching vertices (line 19). Since we are using the lexicographically larger representation, the initial nodes will have the maximum possible number of connections. This also helps in constraining the search and reducing the possible matches.

Figure 4.10 exemplifies the flow of the previously described procedure, when searching for a 3-subgraph on a graph of 6 vertices. Note how the subgraph $\{0, 1, 4\}$ is found twice. This particular property of the procedure as it is now defined will be the subject of the next section.

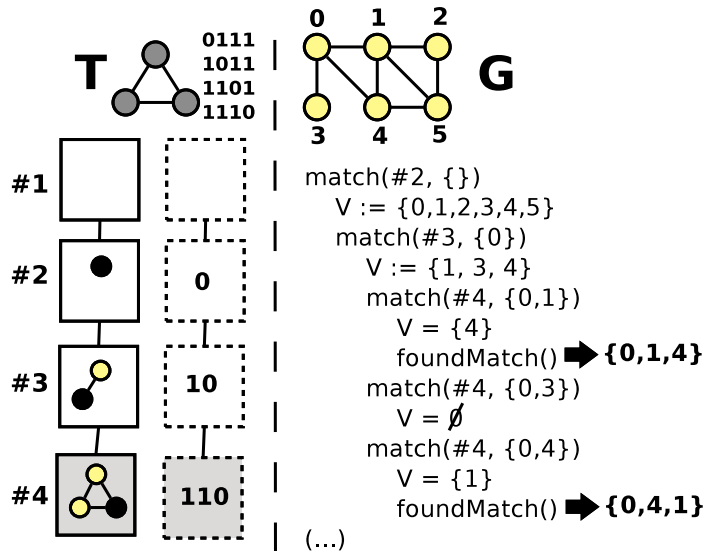


Figure 4.10 – An example of a partial program flow of the recursive g-trie `match()` procedure.

4.5. COMPUTING SUBGRAPH FREQUENCIES

4.5.2 Breaking Symmetries

One problem with the `gtrieMatch()` method described is that we do not avoid subgraph symmetries. If there are automorphisms on a subgraph, then that specific subgraph will be found multiple times. In the example of figure 4.10, we would not only find $\{0, 1, 4\}$ but also $\{0, 4, 1\}$, $\{1, 0, 4\}$, $\{1, 4, 0\}$, $\{4, 0, 1\}$ and $\{4, 1, 0\}$. At the end we can divide by the number of automorphisms to obtain the real frequency, but a lot of valuable computation time is wasted.

4.5.2.1 Creating a Set of Symmetry Breaking Conditions

G-tries need to avoid this kind of redundant computations and find each subgraph only once. In order to achieve that we generate a set of symmetry breaking conditions for each subgraph, similarly to what was done by Grochow and Kellis [GK07]. The main idea is to generate a set of conditions of the form $a < b$, indicating that the vertex in position a should have an index smaller than the vertex in position b .

Figure 4.11 shows an example of a graph and conditions of the type we described that break the symmetry. The conditions fix the position of the vertices indicated in white. Vertices in other colors have at least another equivalent vertex.

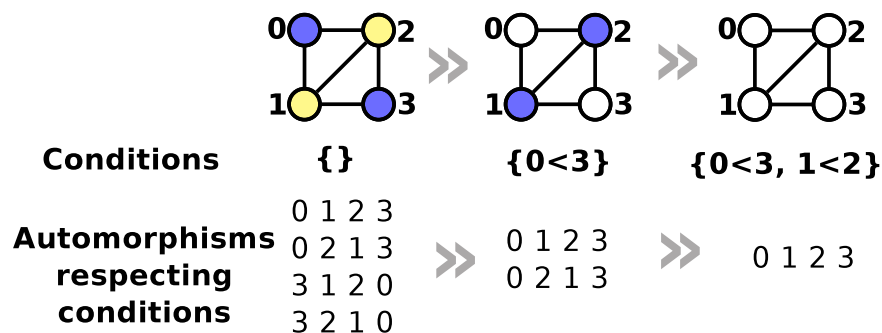


Figure 4.11 – Symmetry breaking conditions for an example 4-graph.

Although inspiration was taken from Grochow and Kellis [GK07], a different method for generating the conditions is used, which we detail in algorithm 4.4. Our algorithm differs from Grochow and Kellis [GK07] because of the method by which we choose the candidates for the conditions.

We start by emptying the set of conditions (line 2). We then calculate the set Aut of automorphisms of the graph (line 3), and start adding conditions that when respected will reduce the above mentioned set to the identity map. Note that although calcu-

CHAPTER 4. THE G-TRIE DATA STRUCTURE

Algorithm 4.4 Symmetry breaking conditions for graph G

Require: Graph G

Ensure: Symmetry breaking conditions of G

```

1: function GTRIECONDITIONS( $G$ )
2:    $Conditions := \emptyset$ 
3:    $Aut := \text{SETAUTOMORPHISMS}(G)$ 
4:   while  $|Aut| > 1$  do
5:      $m := \text{minimum } v : \exists map \in Aut, map[v] \neq v$ 
6:     for all  $v \neq m : \exists map \in Aut, map[m] = v$  do
7:       add  $m < v$  to  $Conditions$ 
8:      $Aut := \{map \in Aut : map[m] = m\}$ 
9:   return  $Conditions$ 

```

lating automorphisms is thought to be computationally expensive, in practice it was found to be almost instantaneous for the subgraph sizes used and with **nauty** [McK81] we were able to test much bigger subgraphs (with hundreds of nodes) and obtain their respective automorphisms very quickly, in less than 1 second. Thus, this calculation is very far from being a bottleneck in the whole process of generating and using g-tries.

In each iteration, to add a new condition, the algorithm finds the minimum index m corresponding to a vertex that still has at least another equivalent node (line 5). It then adds conditions stating that the vertex in position m should have an index lower than every other equivalent position (lines 6 and 7), which in fact fixes m in its position. We choose the minimum index vertex so that, when searching, we can know as soon as possible that a certain partial match is not a suitable candidate. Note that lower indexes mean lower depths in the g-trie.

After this, the algorithm reduces Aut by removing the mappings that do not respect the newly added connections, that is, the ones that do not fix m . It repeats this process until there is only the identity left Aut' (line 4), and finally returns all the generated conditions (line 9). In the case of the graph of figure 4.11, this algorithm would create the exact same set of conditions as depicted there. Figure 4.12 illustrates the symmetry conditions found for all 6 undirected graphs of size 4, with all graphs in the **GTCanon** canonical form.

4.5. COMPUTING SUBGRAPH FREQUENCIES

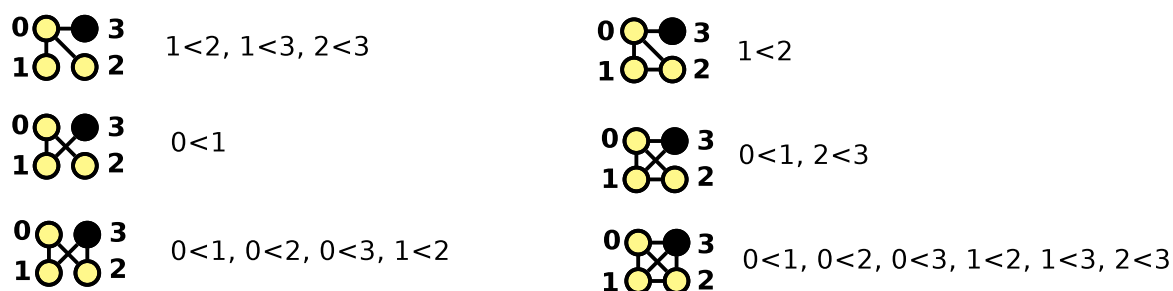


Figure 4.12 – Symmetry conditions computed for all undirected 4-subgraphs.

4.5.2.2 Using the Conditions to Constrain the Search

In order to use the symmetry breaking conditions in g-tries, we store the graph symmetry conditions in all the nodes corresponding to its g-trie path. The matching algorithm can then determine if the partial subgraph constructed respects the conditions of at least one possible descendant g-trie node, that is, there is at least one possible subgraph that can still be constructed by expanding the current partial match and still obeys to the symmetry conditions.

Algorithms 4.5 details how the insertion of the conditions is done. For the sake of understanding, we repeat several lines that are the same as the previous insertion algorithm, and indicate which lines are new. Basically, the only difference is that we now compute the symmetry breaking conditions (line 3) and then we store those conditions along the g-trie path that leads from the root to the final graph node (line 18).

With the symmetry breaking conditions placed in the g-trie nodes, we are now able to search more efficiently for subgraphs. Algorithm 4.6 details how a census with symmetry breaking is done. The same conventions of the insertion algorithm are followed, meaning that we repeat lines that were already on the previous census algorithm and we indicate the new lines of code.

The basic difference is that we now only accept matchings that respect at least one of the sets of conditions stored, that is, that can still correspond to a descendant graph (line 11). Moreover, we detect the minimum possible index for the current node being matched (line 12) and use it to further constraint the generation of candidates (lines 13 and 15). If the neighbours of each network node are sorted (which can be done only once before starting the census), we can use this minimum to discover that further smaller neighbours will never be suitable candidates. In the end we must verify that

CHAPTER 4. THE G-TRIE DATA STRUCTURE

Algorithm 4.5 Inserting a graph G in a g-trie T [with symmetry breaking]

Require: Graph G and G-Trie T

Ensure: Inserts graph G in G-Trie T

```

1: procedure GTRIEINSERT( $G, T$ )
2:    $M :=$  adjacency matrix of  $\text{GTCANON}(G)$ 
3:    $C := \text{SYMMETRYCONDITIONS}(G)$  ▷ NEW CODE
4:   INSERTCONDRECURSIVE( $M, T, 0, C$ ) ▷ NEW FUNCTION HEADER
5: procedure INSERTCONDRECURSIVE( $M, T, k, C$ ) ▷ NEW FUNCTION HEADER
6:   if  $k = \text{NUMBERROWS}(M)$  then
7:      $c.\text{isGraph} = \text{True}$ 
8:   else
9:     for all children  $c$  of  $T$  do
10:      if ( $c.\text{out} =$  first  $k + 1$  values of  $k$ -row of  $M$ ) AND
11:        ( $c.\text{in} =$  first  $k + 1$  values of  $k$ -column of  $M$ ) then
12:        INSERTCONDRECURSIVE( $M, c, k + 1, C$ ) ▷ NEW FUNCTION HEADER
13:      return
14:    $nc :=$  new g-trie node
15:    $nc.\text{in} :=$  first  $k + 1$  values of  $k$ -row of  $M$ 
16:    $nc.\text{out} :=$  first  $k + 1$  values of  $k$ -column of  $M$ 
17:    $nc.\text{isGraph} = \text{False}$ 
18:    $nc.\text{addConditions}(C)$  ▷ NEW CODE
19:    $T.\text{INSERTCHILD}(nc)$ 
20:   INSERTCONDRECURSIVE( $M, nc, k + 1, C$ ) ▷ NEW FUNCTION HEADER

```

a particular matching respects all symmetry breaking conditions for that subgraph. If the graph final vertex is in a g-trie leaf, this step can be skipped, since for sure the conditions are respected. However, if the g-trie node is not a leaf, the search might have arrived there because of the conditions of another descendant subgraph, and therefore the algorithm must assure that the conditions for that particular subgraph are respected.

The method to choose the minimum possible index for the current node that still respects the symmetry conditions ($label_{min}$ on line 12) consists in computing, for each set of conditions, the maximum already mapped node that must be smaller than the current node, and then we pick the minimum of these. Illustrating with an example, imagine that we are trying to match a node to position 2, the symmetry conditions

4.5. COMPUTING SUBGRAPH FREQUENCIES

Algorithm 4.6 Census of subgraphs of T in graph G [with symmetry breaking]

Require: Graph G and G-Trie T

Ensure: All occurrences of the graphs of T in G

```

1: procedure GTRIEMATCHCOND( $T, G$ )
2:   for all children  $c$  of  $T.root$  do MATCHCOND( $c, \emptyset$ ) NEW FUNCTION HEADER
3: procedure MATCHCOND( $T, V_{used}$ ) ▷ NEW FUNCTION HEADER
4:    $V :=$  MATCHINGVERTICESCOND( $T, V_{used}$ ) ▷ NEW FUNCTION HEADER
5:   for all node  $v$  of  $V$  do
6:     if  $T.isGraph \wedge T.GraphConditionsRespected()$  then ▷ UPDATED CODE
7:       FOUNDMATCH( $V_{used} \cup \{v\}$ ) ▷ UPDATED CODE
8:     for all children  $c$  of  $T$  do
9:       MATCHCOND( $c, V_{used} \cup \{v\}$ ) ▷ NEW FUNCTION HEADER
10: function MATCHINGVERTICESCOND( $T, V_{used}$ ) ▷ NEW FUNCTION HEADER
11:   if NOT  $\exists C \in T.cond : V_{used}$  respects  $C$  then return  $\emptyset$  ▷ NEW CODE
12:    $label_{min} :=$  minimum possible index for current position ▷ NEW CODE
13:   if  $V_{used} = \emptyset$  then  $V_{cand} := \{v \in V(G) : v \geq label_{min}\}$  ▷ UPDATED CODE
14:   else
15:      $V_{conn} := \{v : v \in N(V_{used}) \wedge v \geq label_{min}\}$  ▷ UPDATED CODE
16:      $m := m \in V_{conn} : \forall v \in V_{conn}, |N(m)| \leq |N(v)|$ 
17:      $V_{cand} := \{v \in N(m) : v \notin V_{used}\}$ 
18:    $Vertices := \emptyset$ 
19:   for all  $v \in V_{cand}$  do
20:     if  $\forall i \in [1..|V_{used}|]$ :
21:        $T.in[i] := G_{Adj}[V_{used}[i]][v] \wedge T.out[i] = G_{Adj}[v][V_{used}[i]]$  then
22:        $Vertices := Vertices \cup \{v\}$ 
23:   return  $Vertices$ 

```

are $\{\{0<1, 1<2\}, \{0<2, 1<2\}\}$, and the current matching is $V_{used} = \{34, 12\}$, which means that network node number 34 is matched to position 0, and node 12 is matched to position 1. Since we are matching position 2, only the conditions with 2 matter. For the first set this is $1<2$, which means that the current node must be larger than 12. For the second set, we must take into consideration $0<2$ and $1<2$, which means that the node must be simultaneously larger than 34 and 12. We take the maximum, which is 34. Afterwards, we know that the node must be larger than 12 (first set) or

CHAPTER 4. THE G-TRIE DATA STRUCTURE

larger than 34 (second set), and therefore we take the minimum, and we would have $label_{min} := 12$. If any set of conditions is empty, then $label_{min} := 0$, that is, there is no minimum for the index of the node.

With these two symmetry aware algorithms (insertion and census), a subgraph will only be found once. All other possible matchings of the same set of vertices will be broken somewhere in the recursive backtracking. Moreover, since the conditions generation algorithm always create conditions of the minimal indexes still not fixed (line 5 of algorithm 4.4), the census algorithm can discover early in the recursion that a condition is being broken, therefore cutting branches of the possible search tree as soon as possible.

4.5.2.3 Reducing the Number of Conditions

By using the last two algorithms, we may end up having a large number of symmetry conditions on a single g-trie node, since it can have a very large number of descendants. This can have a severe impact on memory costs and influence the performance, and we should reduce as much as possible this cost. With that in mind, we use four steps to filter and reduce the symmetry conditions.

Step #1 reduces the set of conditions by using the transitive property of the “less” relationship, and in the cases where $a < b$, $a < c$ and $b < c$ are in the set of conditions, we remove the condition $a < c$. This is illustrated in Figure 4.13, for all 6 undirected 4-graphs.

Step #2 reduces the conditions to the ones that matter to that particular node. This means that if we are at a certain g-trie depth, conditions in which one of the elements is bigger than the depth are discarded, that is, the conditions that are referring to descendant nodes that are still not matched. Figure 4.14 illustrates how a g-trie containing all 6 undirected 4-graphs would look like after this step.

Step #3 discards sets of conditions that are redundant. Since each g-trie node has a group of sets of conditions (one for each descendant graph) and it must assure that at least one of those sets is respected (meaning that that at least one descendant graph is achievable), we search the group for sets that are redundant, in the sense that they include another one, and we remove those sets. For instance, a descendant graph that imposes no conditions at all means that any partial match must be continued from there because of that graph. Therefore, no more sets of conditions are needed in that g-trie node, besides the empty one. Another example can be given using the sets of

4.5. COMPUTING SUBGRAPH FREQUENCIES

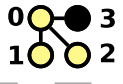

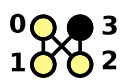



Symmetry Conditions		
	Original	After Filtering
	$1 < 2, 1 < 3, 2 < 3$	$1 < 2, 2 < 3$
	$0 < 1$	$0 < 1$
	$0 < 1, 0 < 2, 0 < 3, 1 < 2$	$0 < 1, 0 < 3, 1 < 2$
	$1 < 2$	$1 < 2$
	$0 < 1, 2 < 3$	$0 < 1, 2 < 3$
	$0 < 1, 0 < 2, 0 < 3, 1 < 2, 1 < 3, 2 < 3$	$0 < 1, 1 < 2, 2 < 3$

Figure 4.13 – Filtering symmetry conditions: step #1.

conditions $\{\{0 < 1\}, \{0 < 1, 1 < 2\}\}$. In this case we can discard the second set since the first one is included in it, that is, if a partial graph respects the second set, it would also respect the first set, and therefore the second set is redundant if the algorithm is trying to assure that at least one of the condition sets is respected. Figure 4.15 illustrates how the same g-trie containing all 6 undirected 4-graphs would look like after this third filtering step.

Step #4 is the final one and removes conditions that are already assured. It is applied after having all graphs already inserted in the g-trie, and all other filtering steps are already made. If at any g-trie node there is a specific condition $a < b$ that is included in all of the sets, we can be assured that this condition is certainly respected and all descendant nodes do not need to verify it again. As an example, consider the set $\{\{0 < 1\}, \{0 < 1, 1 < 2\}\}$. The condition $0 < 1$ is in every set and therefore we remove it from all descendant g-trie nodes. In the case of the g-trie with all undirected 4-subgraphs, this particular filtering step does not remove any conditions, but in larger g-tries it can eliminate a large number of conditions.

By following the 4 described filtering steps, the resulting g-trie has a much reduced number of stored conditions, which will not only save memory, but will also be more efficient for census computation, as there are less conditions to be verified.

CHAPTER 4. THE G-TRIE DATA STRUCTURE

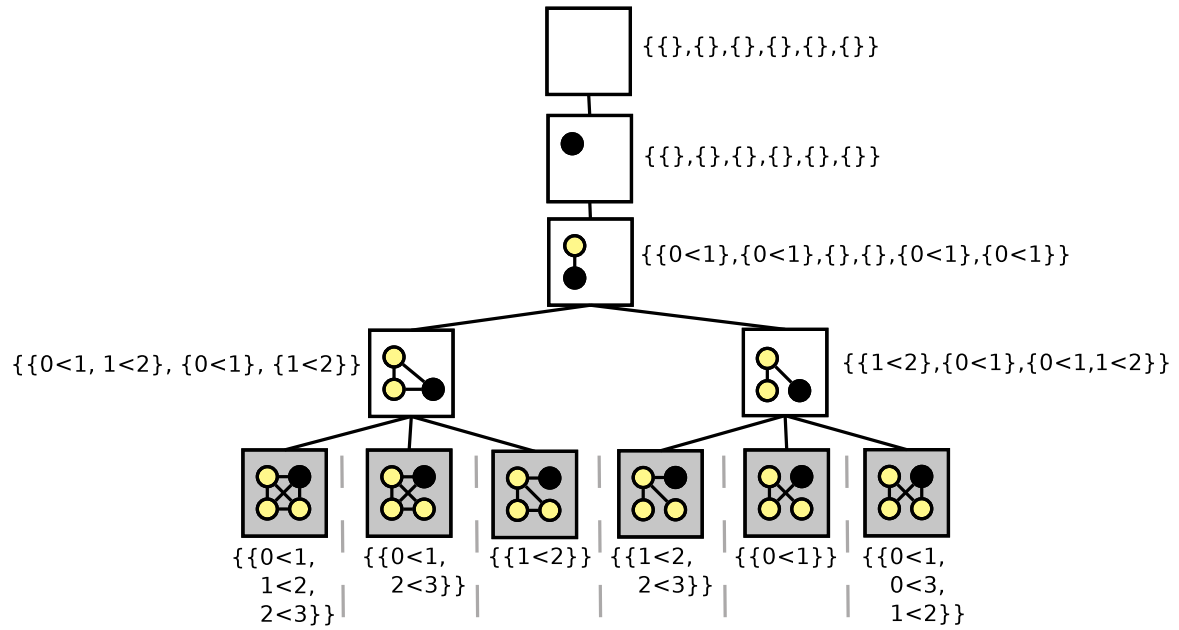


Figure 4.14 – Filtering symmetry conditions: step #2.

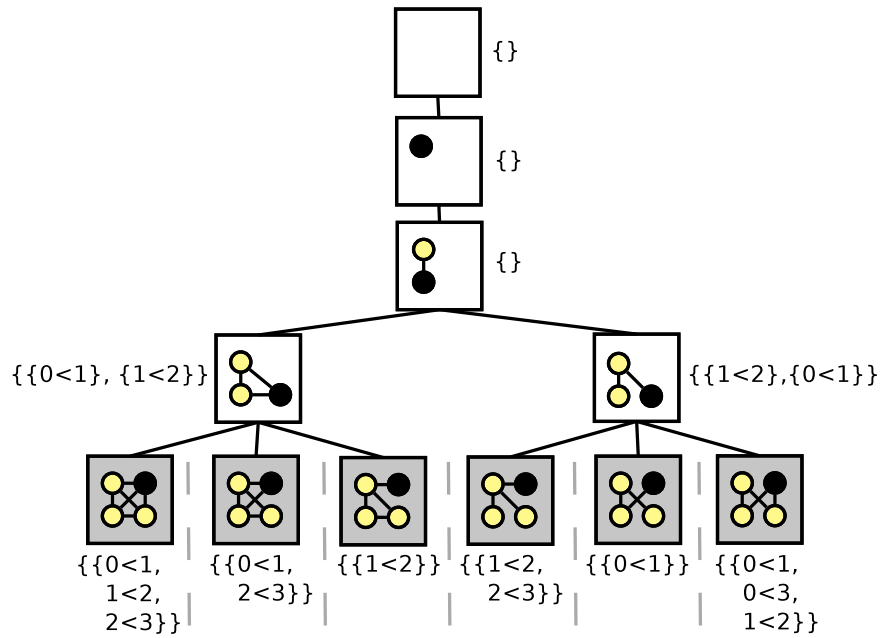


Figure 4.15 – Filtering symmetry conditions: step #3.

4.6 Sampling Subgraphs

The census algorithm given before creates an exhaustive and complete enumeration of all subgraph occurrences. One way to accelerate its execution is to approximate the exact value by only sampling a fraction of all the occurrences, trading accuracy for execution speed. This section details how this can be done.

4.6.1 Uniform Sampling

The described recursive matching algorithm induces a search tree in which the search nodes in the last depth level correspond to the g-trie leaf nodes, as exemplified in Figure 4.16. Squares correspond to trying a g-trie node (a call to `matchCond`) and hexagons correspond to trying all candidate nodes for a certain node (line 5 of Algorithm 4.6). The dashed boxes with '...' indicate search branches that could continue and the vertex numbers are only shown for illustration purposes.

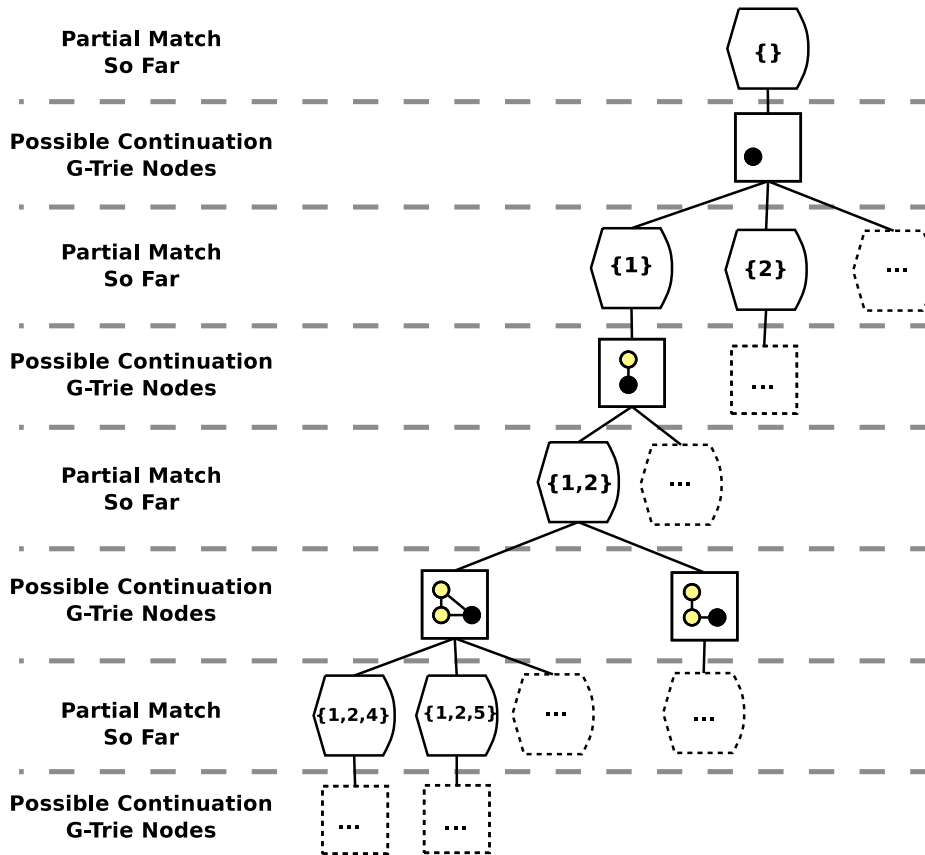


Figure 4.16 – An example g-trie matching search tree.

CHAPTER 4. THE G-TRIE DATA STRUCTURE

Computing the exact census corresponds to traversing this entire search tree. In order to sample, our main conceptual idea is to only explore each g-trie search branch (the squares of the figure) with a certain probability. This is similar to what was done in [Wer06]. Algorithm 4.7 details our approach. Note that it is exactly the same as the previous algorithm with the exception of the indicated lines. Probability of reaching a leaf is P , with $P = \prod P_d$, where P_d is probability of depth d .

Algorithm 4.7 Sample subgraphs of g-trie T in graph G .

Require: Graph G , G-Trie T and set of probability values P

Ensure: Sample of occurrences of the graphs of T in G

```

1: procedure GTRIESAMPLEALL( $T, G, P$ )
2:   for all children  $c$  of  $T.root$  do
3:     With probability  $P_0$  do MATCHSAMPLE( $c, \emptyset$ ) ▷ NEW CODE
4: procedure MATCHSAMPLE( $T, V_{used}$ ) ▷ NEW FUNCTION HEADER
5:    $V = \text{MATCHINGVERTICESCOND}(T, V_{used})$ 
6:   for all node  $v$  of  $V$  do
7:     if  $T.isGraph \wedge T.GraphConditionsRespected()$  then
8:       FOUNDMATCH( $V_{used} \cup \{v\}$ )
9:     for all children  $c$  of  $T$  do
10:      With probability  $P_{T.depth}$  MATCHSAMPLE( $c, V_{used} \cup \{v\}$ ) ▷ NEW CODE

```

In order to follow a probabilistic approach, the algorithm uses a set of probabilities associated to each g-trie depth: $\{P_0, P_1, \dots, P_{gtrie_max_depth}\}$ where $0 \leq P_i \leq 1$. Any given node of depth d will therefore only be reached with probability $P_0 \times \dots \times P_{d-1}$. With this, we can produce an unbiased estimator of the frequency count of a single subgraph. Let P_i be the probability associated with depth i and $F_{sample}(G_k, G)$ be the number of occurrences of the k -subgraph G_k found in G by the `gtrieSampleAll()` procedure of Algorithm 4.7. Then, an unbiased estimator $\hat{F}(G_k, G)$ of the total number of occurrences of G_k in G is given by the following equation:

$$\hat{F}(G_k, G) = \frac{F_{sample}(G_k, G)}{P_0 \times P_1 \times \dots \times P_{k-1}} \quad (4.2)$$

We say that the estimator is unbiased because any occurrence of G_k can be found with equal probability, and as we increase the set of probabilities, the estimator gets closer to the real value. In fact, if we choose $P_i = 1$ for all i , then the result is the same as the original complete algorithm.

4.6. SAMPLING SUBGRAPHS

Figure 4.17 exemplifies how the probability values can affect which search tree branches are followed. Assuming the probabilities given in the figure, hypothetically the colored nodes would be explored and the white nodes would not be.

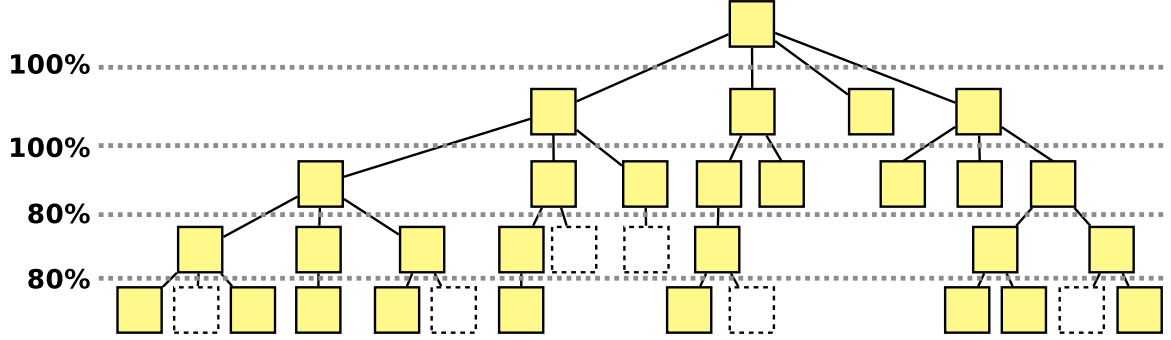


Figure 4.17 – Associating a probability with each search tree depth.

4.6.2 Sampling Parameters

The parameters P_i control the search. Regarding the accuracy, we should avoid small values of probability for lower depths, closer to the root. Its effect is to increase the variance of the result because any disregarded branch in lower depths may correspond to entire parts of the graph, and therefore may correspond to a higher number of subgraph occurrences not found. In terms of the execution time, the opposite happens. Very high probabilities in the lower depths will increase the execution time, since more parts of the search tree will have to be computed. For example, in the extreme case of having all probabilities equal to one except the last one, in the higher possible depth d , means that in practice we will explore all possible subgraphs of depth $d - 1$.

Picking the parameters is therefore a delicate choice that will influence both the accuracy and speed of our method. The results chapter gives more detail on possible choices for the parameters (see Section 6.2.4).

The main benefit of our sampling algorithm when compared to other proposals, is that it is able to sample only the desired set of subgraphs (**mfinder** and **ESU** can only sample the entire set of possible k -subgraphs and **MODA** can only sample the occurrences of a particular single subgraph).

The quality of the estimation depends on many factors. A fully fledged analytical determination of tight bounds on error margins is very complicated since we do not know beforehand the distribution of the subgraphs that we are looking for. For

CHAPTER 4. THE G-TRIE DATA STRUCTURE

example, if the subgraph is very well spread in the entire subgraph, we will have less variance than if all occurrences are clustered in a small number of nodes. This is because in the later case a search branch not followed may imply a significant percentage of occurrences not found.

4.7 Motif Discovery with G-Tries

With the g-tries algorithms described in this chapter it is now possible to discover motifs. As shown before in Algorithm 3.1, the main flow of all exact network motifs algorithms is to calculate a census of subgraphs of a determined size k in the original network, then generate a set of similar random networks, followed by the calculation of the census on all of those, in order to assess the significance of the subgraphs present in the original network.

The generation of the random networks themselves (normally done by a Markov chain process [MSOI⁺02]) is just a very small fraction of the time that the census takes. Computing the census on all random networks is therefore the main bottleneck of the whole process (there can be hundreds of random networks) and **g-tries** can help precisely in this phase. In order to use g-tries we propose two possible approaches:

- **G-Trie use only** - generate all possible graphs of a determined size (for example using McKay's **gtools** package [McK98]), insert all these in a g-trie and then apply g-trie matching to the original network. Create a new g-trie only with the subgraphs found, and then apply g-trie matching to the random networks, with this new g-trie.
- **Hybrid approach** - use another network-centric method to enumerate the subgraphs in the original network, like the efficient **ESU** algorithm. Create a g-trie only with the subgraphs found, and then apply g-trie matching to the random networks.

In both cases we will only be trying to discover in the random networks the subgraphs that appear in the original network, and not spending execution time trying to find subgraphs that are not interesting from the motifs problem point of view.

Note that in some cases one may also be interested in *anti-motifs*, which as the name suggests are patterns under-represented. In this case, a complete census must also

be performed in the random networks, either with other method, or with the g-trie containing all possible subgraphs.

4.8 Summary

In this chapter we described a novel data-structure, the g-tries. It allows the representation of a set of graphs by using a tree that identifies common substructure, thus avoiding redundant representation. This characteristic of g-tries can be used for generating a new efficient methodology for discovering motifs. Algorithms for creating and using g-tries for this purpose were described. The possibility of trading accuracy for faster execution times was also explored and the associated algorithm was described.

*When you are stuck in a traffic jam with a
Porsche, all you do is burn more gas in idle.
Scalability is about building wider roads,
not about building faster cars.*

Steve Swartz

5

Parallel Network Motif Discovery

Almost all the algorithms presented in the previous chapters are sequential in their nature. Considering the computational tractability of the problem, resorting to scalable parallelism for speeding up the computation is a research path capable of pushing the limits and having impact in the feasible sizes of motifs and networks. The purpose of this chapter is therefore to identify opportunities for parallelism in the motif discovery process and to present real and practical algorithms for parallelizing all aspects of it. We start by analyzing the algorithmic flow and we create a taxonomy of possible parallel approaches. We then show that the whole computation can be expressed as a tree based search with non-overlapping nodes, thus exposing its inherent parallel nature. Furthermore, the computation can be divided into several steps for which we present different strategies, with dynamic load balancing during the parallel processing. We then describe in more detail how these techniques can be specifically applied by using both g-tries and ESU algorithms in order to efficiently discover motifs.

5.1 Opportunities for Parallelism

Parallelism in the motif discovery realm has been scarcely used, as was shown in Section 3.5. Its usage could however lead to significant efficiency improvements, which could in turn lead to bigger achievable sizes both in motifs and networks. It seems only natural then to explore this possibility.

CHAPTER 5. PARALLEL NETWORK MOTIF DISCOVERY

Chapter 3 detailed the sequential program flow of the motif discovery process, presenting the respective pseudo-code in Algorithm 3.1. As Figure 5.1 shows more graphically, the basic methodology starts by computing a subgraph census on the original network. This is followed by the generation of an ensemble of similar random networks, the respective census computation for each of them, and finally with the analysis of the significance of each isomorphic class of subgraphs, knowing their frequency on both the original network and the random networks.

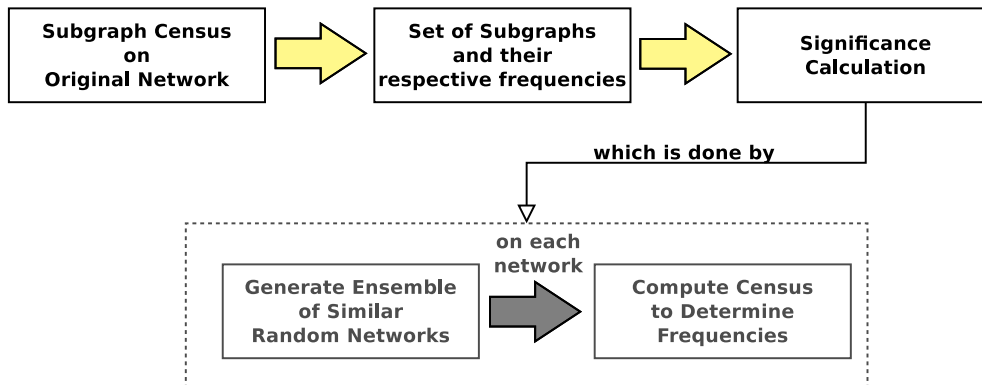


Figure 5.1 – Motif discovery algorithmic flow.

This process offers several opportunities for parallelism that we now identify by creating a taxonomy that we later use in order to classify parallel algorithms for motif discovery, namely:

- **Census Parallelization:** compute in parallel the census of subgraphs in a single network. This in turn can be done with three different methods:
 - **Partition:** the network is pre-divided in several (possibly overlapping) partitions/regions and different processors analyze different partitions.
 - **Tree:** a recursive search procedure is executed in parallel, with different search tree branches being searched at the same time in different processors.
 - **Query:** in subgraph-centric algorithms, each individual subgraph query is done separately by different processors.
- **Random Networks Parallelization:** distribute the random networks between the processors. For example, if we have to generate 100 random networks and have access to 100 processors, then each processor could compute its own random network and its corresponding census.

5.2. MOTIF DISCOVERY AS A TREE SHAPED COMPUTATION

- **Significance Parallelization:** distribute the significance calculation after the census is computed.

With this nomenclature at hand, we can now classify the strategies described in Section 3.5, namely Wang et al. [WTZ⁺05] and Schatz et al. [SCBB08], and both rely only on single census parallelization. More specifically, Wang et al. use partition parallelization and Schatz et al. use separately partition and query parallelization.

If we profile the sequential algorithm during real computations, we find that the main bottleneck are the census computations (both on the original and the random networks), taking on average more than 95% of the whole execution time. Therefore, census parallelization is really a key issue. If we can do a single census in parallel, one way of doing the whole computation is to do exactly as the sequential algorithm, except that individual census calls are done in parallel. On its own this strategy presents two main drawbacks:

- Synchronization is necessary after each census to ensure that all processors have completed their computation before the next census begins. Since typically we generate at least dozens of random networks, this can provoke a significant amount of unwanted idle time on processors.
- The other steps of the network motif discovery must be done sequentially. In particular, repeating the process of generating a similar network for every new random network can be time consuming.

We could therefore do better if we parallelized at the same time all the steps needed to discover motifs (not just the census) and this is precisely the aim of this chapter. The goal is to use at the same time census and random network parallelization, something which was not done before. The significance computation takes on average less than 0.01% of the total time and therefore even if done sequentially will not detract good scalability.

5.2 Motif Discovery as a Tree Shaped Computation

In the previous chapter we have shown how to use g-tries in order to discover motifs. More specifically, in Section 4.7, we have provided two possibilities: g-tries only (GO)

CHAPTER 5. PARALLEL NETWORK MOTIF DISCOVERY

and an hybrid approach (HA). Both approaches start by doing a subgraph census on the original network, either by generating all possible subgraphs and inserting them into a g-trie (GO) or by using any other method other than g-tries (HA). The next phase uses the computed census to feed a g-trie only with the subgraphs that present the desired minimum frequency and then that g-trie is used to count the occurrences of those subgraphs in the ensemble of random networks.

For the purposes of this work, in the hybrid approach, we will use exclusively the ESU algorithm as the basis for the initial step. This is due to several reasons: (a) it is one of the most efficient among the already existing algorithms (Chapter 6 shows this more empirically, with experimental results); (b) it allows for sampling, like g-tries, in a way that we can trade accuracy for speed in the whole motif discovery; (c) it uses as a basis a recursive search procedure that implicitly builds a search tree like in the case of g-tries, a common property that we will explore to use the same generalizable parallel approaches for both methods.

With all of this in mind, the whole motif discovery process using g-tries and R random networks can be depicted as a very large search tree, as exemplified in Figure 5.2. The search trees of individual census are only given as examples, and we can imagine that they could present completely different layouts. Note that there are two synchronization points: (a) computing the census of the random networks can only be started after the census on the original network, since we will be using a g-trie containing only the relevant subgraphs found on that original census; (b) computing the significance can only be started after all the census are done, because it requires both the frequency in the original and in the random networks. In the figure, G_1 to G_N are the subgraph isomorphic classes for which we have to compute its respective significance.

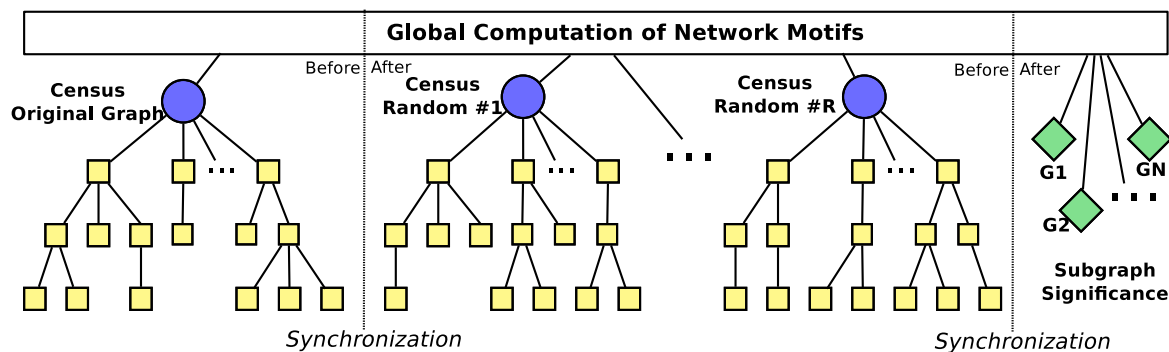


Figure 5.2 – Motif discovery as a tree shaped computation.

5.2. MOTIF DISCOVERY AS A TREE SHAPED COMPUTATION

Thus, parallelizing motif computation consists now, in its essence, in parallelizing the traversal of this global search tree. Let us now analyze in more detail the search trees of ESU and g-tries census, the basic building blocks for each individual census, highlighting a very important common characteristic: the independence of subtrees.

5.2.1 The Search Tree of ESU Census

When performing a subgraph census with the ESU algorithm, an implicit search tree is created, with each internal tree node being a call to the `extendSubgraph(V_{Subg}, V_{Ext}, v)` recursive procedure (line 4 of algorithm 3.3), where v is the root vertex, V_{Subg} is the partially constructed subgraph and V_{Ext} are the possible nodes for extending the subgraph. Figure 5.3 exemplifies such a search tree, with each internal node indicating the parameters passed as $\{V_{Subg}, V_{Ext}\}$.

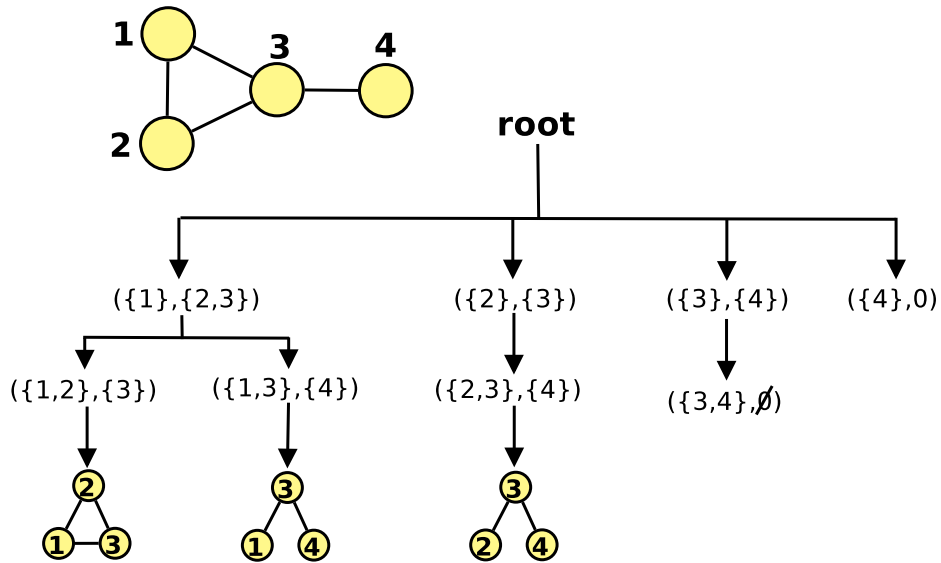


Figure 5.3 – Search tree of ESU algorithm (revisited).

Note that the root vertex v is always the first element of V_{Subg} , which makes this information redundant. Moreover, a crucial aspect is that all the subtrees (different calls to the recursive procedure) are independent from each other. Therefore, a pair (V_{Subg}, V_{Ext}) uniquely identifies where we are in the search and we can continue from that point without knowledge of what may have been computed before. This is vital and will be put to use in the proposed parallel algorithms.

5.2.2 The Search Tree of G-Trie Census

When doing a subgraph census with the g-tries matching algorithm, an implicit search tree is also created, with each internal node being a call to the recursive procedure $\text{MatchCond}(T, V_{used})$ (line 3 of Algorithm 4.6), where T is a g-trie node and V_{used} are the already used network vertices that match the subgraph of the g-trie node. Figure 5.4 exemplifies this, with each internal node representing the respective g-trie node and the used vertices. Nodes with '...' are search branches that could continue and the vertex numbers are only shown for illustration purposes.

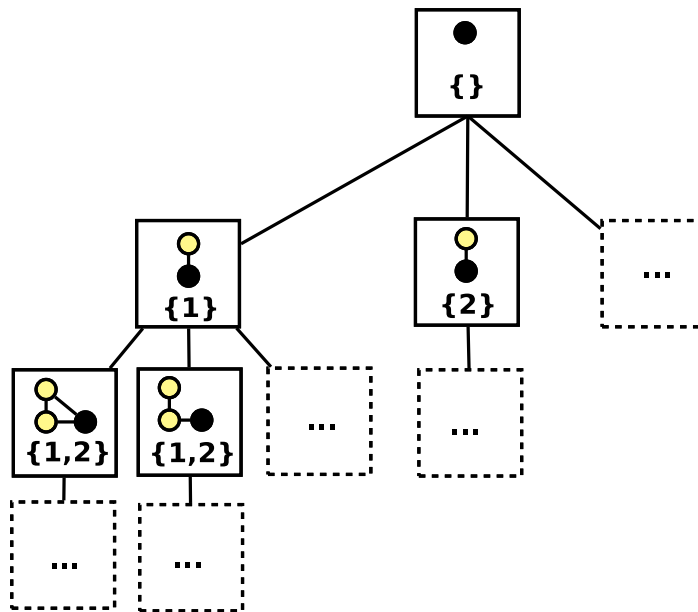


Figure 5.4 – An example g-trie matching search tree (revisited).

As in the case of ESU, the crucial aspect is that all subtrees (different calls to the recursive procedure) are independent from each other. Therefore, a pair (T, V_{used}) uniquely identifies where we are in the search and we can continue from that point without the need to know anything else.

5.3 A General Parallel Approach

We have seen in the previous sections that the whole motif discovery can be seen as large tree shaped computation and that different search branches exhibit a crucial property: they are independent. In this section we will describe general parallel strategies that are able to exploit this property to efficiently search this tree.

5.3. A GENERAL PARALLEL APPROACH

5.3.1 Terminology

In order to better express all the concepts used in this section and to use a coherent terminology, the following notation will be used:

- P - the number of processors, or cores. If the processor is actively participating in the motif computation (and not just used for scheduling purposes) then it is also called a *worker*.
- R - the number of similar random networks.
- N - the number of meaningful subgraphs found in the original network and that will be fed to the g-trie used for matching the random networks.
- id - a number representing a network, where 0 (zero) means it is the original subgraph and any $0 < id \leq R$ means it is the id -th random network.

5.3.2 Work Units

We call each node of the search tree a *work unit*. Each of these corresponds to actual work that needs to be done, and there are three possible types of work units:

- \bigcirc_{id} - the census of the single network id .
- $\square_{(id,s)}$ - an internal node of the single census of network id , with a partially constructed subgraph of size s . Note that this node can be either from ESU or g-trie algorithms, but in both cases we have a partial subgraph already constructed. If we want to specify the contents of a work unit of this type we will use the notation $\square_{(id,s)}(contents)$, that is, $\square_{(id,s)}(V_{Subg}, V_{Ext})$ for ESU and $\square_{(id,s)}(T, V_{used})$ for g-tries.
- \diamond_{sg} - computation of the significance of the subgraph sg . Note that $1 \leq sg \leq N$.

This terminology (\bigcirc , \square and \diamond) was already used in Figure 5.2.

With all of this at hand, solving the network motifs problem can from now on be expressed using our work unit symbols. We need to solve in order the **work queues** $Q_{original}$, Q_{random} and $Q_{significance}$, defined in Equations 5.1, 5.2 and 5.3, which correspond respectively to the census of the original network, the census of the random networks and the significance computation.

CHAPTER 5. PARALLEL NETWORK MOTIF DISCOVERY

$$Q_{original} = \{\bigcirc_0\} \quad (5.1)$$

$$Q_{random} = \{\bigcirc_1, \bigcirc_2, \dots, \bigcirc_R\} \quad (5.2)$$

$$Q_{significance} = \{\diamond_1, \diamond_2, \dots, \diamond_N\} \quad (5.3)$$

Note that in practice, computing a \bigcirc_{id} can be decomposed in several smaller $\square_{(id,i)}$. So, for example, \bigcirc_0 can be decomposed in several $\square_{(0,1)}$, that is, computing the census of the original network can be decomposed in all the work units that have a partial subgraph of size 1. Exemplifying with the ESU algorithm, this means computing the census consists in trying all different network vertices as possible root nodes.

Every $\square_{(0,i)}$ will then be decomposable in several $\square_{(0,i+1)}$ until the motif size is reached. This means that every work unit with a partial subgraph of size n will generate several possible work units with partial subgraphs of size $n + 1$.

Note that different $\square_{(id,i)}$ will generate a different number of smaller work units, corresponding to different topological parts of the network. This means that the search tree is completely unbalanced.

The actual computation of a work unit is detailed in Algorithms 5.1, 5.2 and 5.3. Essentially, according to the type of the work unit and the current method being applied, the respective function is called. Algorithm 5.1 is just the dispatcher code, together with the significance calculation if the work unit is of the type \diamond , according to Equation 2.1. Algorithm 5.2 mimics the behavior of the ESU algorithm (previously shown in Algorithm 3.3), and Algorithm 5.3 mimics the behavior of the g-tries matching algorithm with conditions (previously shown in Algorithm 4.6), whose details were already explained. The main difference is that each recursive call of the algorithms is here transformed into a new work unit, ready to be computed at a later stage.

Note that for efficiency reasons, our actual implementations of the algorithms just described are not exactly like the given pseudo-code, but their functionality is the same. In particular, the dispatcher function `expand()` is specialized for each method and work unit type, avoiding constant comparisons for knowing which subgraph census method is being used and for knowing the unit work type. Furthermore, when expanding a work unit, the behavior of the queue is simulated by using the recursive stack. The insertion of the unexplored units into a queue is delayed until the program has to stop computing and must communicate part of the work units to another processor.

5.3. A GENERAL PARALLEL APPROACH

Algorithm 5.1 Expanding or computing a Work Unit (main part)

Require: Work Unit U and method M

Ensure: New “smaller” unprocessed work units or storage of results in memory

```

1: function EXPAND( $U, M$ )
2:   if TYPE( $u$ ) =  $\bigcirc$  then
3:     if  $M = \text{ESU}$  then return ESUMAIN( $U$ )
4:     else if  $M = \text{GTries}$  then return GTRIESMAIN( $U$ )
5:   else if TYPE( $u$ ) =  $\square$  then
6:     if  $M = \text{ESU}$  then return ESUEXPAND( $U$ )
7:     else if  $M = \text{GTries}$  then return GTRIESEXPAND( $U$ )
8:   else if TYPE( $u$ ) =  $\diamond(sg)$  then
9:      $z\text{-score}(sg) := (f_{\text{original}} - \bar{f}_{\text{random}}) / \text{std}(f_{\text{random}})$ 
10:  return  $\emptyset$ 

```

Algorithm 5.2 Expanding or computing a Work Unit (ESU part)

```

11: function ESUMAIN( $\bigcirc_{id}$ )
12:   $Q := \emptyset$ 
13:  for all  $v \in V(G_{id})$  do
14:     $Q.\text{ADD}(\square_{(id,1)}(\{v\}, \{u \in N(v) : u > v\}))$ 
15:  return  $Q$ 

16: function ESUEXPAND( $\square_{(id,size)}(V_{Subg}, V_{Ext})$ )
17:   $Q := \emptyset$ 
18:  if  $|V_{Subg}| = k$  then
19:    INCREMENTCOUNT(canonicalLabeling( $V_{Subg}$ ))
20:  else
21:    while  $V_{Ext} \neq \emptyset$  do
22:      remove random chosen  $w \in V_{Ext}$ 
23:       $V'_{ext} := V_{ext} \cup \{u \in N_{excl}(w, V_{subg}) : u > \text{firstVertex}(V_{Subg})\}$ 
24:       $Q.\text{ADD}(\square_{(id,size+1)}(V_{subg} \cup \{w\}, V'_{ext}))$ 
25:  return  $Q$ 

```

5.3.3 Parallel Strategies

Since our work units are independent, we do not really have the need to use shared memory. We target the distributed memory model, which offers scalability to a large

CHAPTER 5. PARALLEL NETWORK MOTIF DISCOVERY

Algorithm 5.3 Expanding or computing a Work Unit (g-tries part)

```

26: function GTRIESMAIN( $\bigcirc_{id}$ )
27:    $Q := \emptyset$ 
28:    $T :=$  g-trie of interesting subgraphs
29:   for all children  $c$  of  $T.root$  do
30:      $Q.ADD(\square_{(id,1)}(c, \emptyset))$ 
31:   Return  $Q$ 

32: function GTRIESEXPAND( $\square_{(id,size)}(T, V_{used})$ )
33:    $V =$  MATCHINGVERTICESCOND( $T, V_{used}$ )
34:   for all node  $v$  of  $V$  do
35:     if  $T.isGraph \wedge T.GraphConditionsRespected()$  then FOUNDMATCH()
36:     for all children  $c$  of  $T$  do
37:        $Q.ADD(\square_{(id,1)}(c, V_{used} \cup \{v\}))$ 

```

number of processors, thus allowing massive parallelization of motif discovery. We use the message passing interface (MPI) as our communication model.

In order to parallelize our search we have to distribute the work-units among all worker processors. One problem is that, as seen, the search tree is highly unbalanced and the execution time of each work-unit varies significantly. Figures 5.5 and 5.6 better illustrate this by showing the relative weight of all $\square_{(0,1)}$ in the total execution time when applying ESU and g-tries to compute the census of 8-subgraphs in a social network [LSB⁺03, New09], that is, the percentage of time spent using as a starting partial subgraph each of the 62 single nodes of the network. For better legibility, work units are sorted decreasingly by their weight.

Note the high variability, with some units taking around 20% of the total time, with others being almost insignificant in terms of execution time. This makes it very hard to use a pre-determined static allocation scheme, and approximating the execution time cost of a work-unit can be as hard as computing the work-unit itself. Therefore, we opted for a dynamic load balancing strategy, that redistributes work among the worker processors during the execution.

Computing a work queue in parallel defines a *parallel job*. As seen before, the motif discovery problem consists in computing three different queues in order: $Q_{original}$, Q_{random} and $Q_{significance}$, defined in Equations 5.1, 5.2 and 5.3.

Each parallel job is made of three main phases, that must also be done in order:

5.3. A GENERAL PARALLEL APPROACH

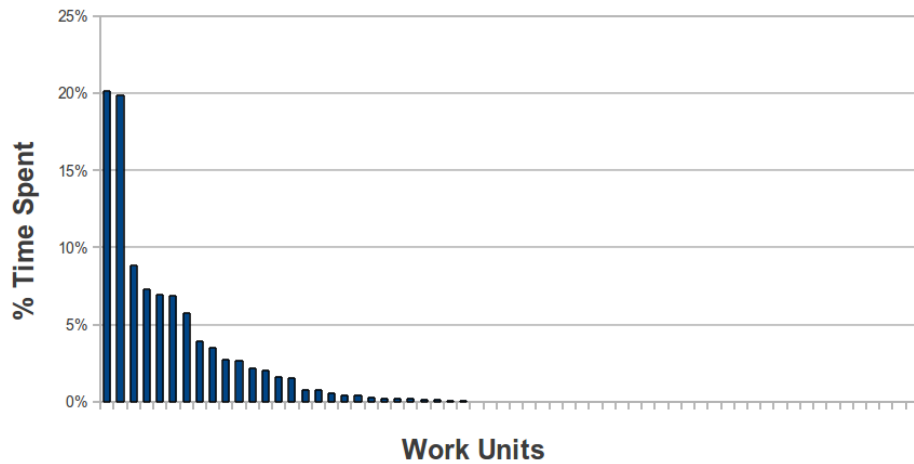


Figure 5.5 – Relative execution time of work units applying ESU to a social network. Network data from [LSB⁺03, New09].

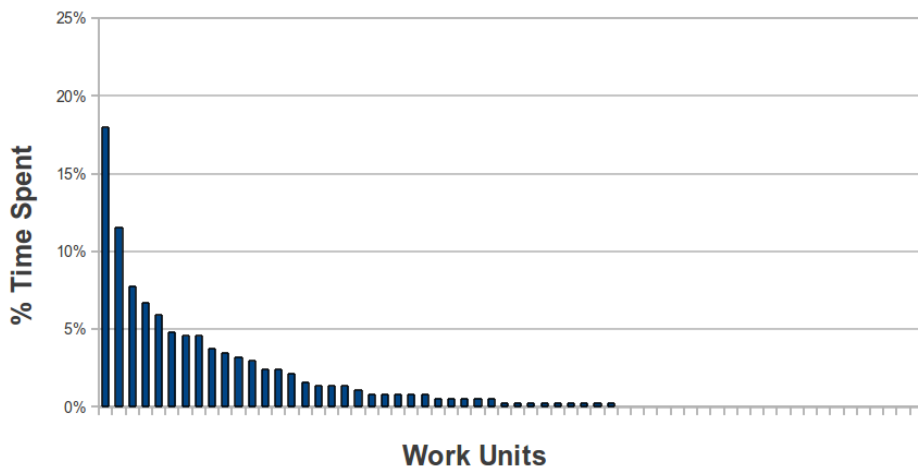


Figure 5.6 – Relative weight of work units applying g-tries to a social network. Network data from [LSB⁺03, New09].

1. **Pre-Processing Phase.** Performs all the computations required to start the job, providing an initial work queue for each worker.
2. **Work Phase.** Performs the bulk of the job, analyzing subgraphs and discovering their frequency. It consists of workers computing the current work queue and asking for more work when their queues become empty.
3. **Aggregation Phase.** In this phase the subgraph frequencies found on each worker are aggregated on a single processor.

CHAPTER 5. PARALLEL NETWORK MOTIF DISCOVERY

For each phase we developed several possible strategies, that are enumerated in Table 5.1. In the next sections we will detail exactly how each of these phases and respective strategies are implemented. Any of these strategies is general enough to be applied both to ESU and g-tries and they can be combined at will, because using a determined strategy on a phase does not hinder the applicability of a different strategy on another phase.

Phase	Possible Strategies
Pre-Processing	<ul style="list-style-type: none"> • all_in_one • static_partition
Work	<ul style="list-style-type: none"> • master-worker • distributed queues • distributed snapshot
Aggregation	<ul style="list-style-type: none"> • naive • hierarchical • collective

Table 5.1 – Strategies for different phases of a parallel job

5.3.4 Pre-Processing Phase

Each worker processor has its own work queue, with Q_i being the work queue of worker i ($1 \leq i \leq P$). We need to make sure that all computational work units are in some work queue. So, for example, when computing the census on all random networks, we must guarantee that $\cup Q_i = Q_{random}$.

We identify and propose two different strategies for this phase:

- **all_in_one:** the entire work queue is put on a single worker, and all other workers start with an empty queue.
- **static_partition:** Q_{census} is statically divided among all workers.

This last option seems intuitively better, since all processors can start working on their work queues, without the need for initial communication and distribution of work units. Note that as said before, this static division cannot provide a guaranteed balanced partition of the work, but it allows a speedier start up of the computation.

5.3. A GENERAL PARALLEL APPROACH

Algorithm 5.4 details our implementation of a static division using a round robin scheme. If the work queue has less work units than the number of workers (line 2) then the first work unit is removed from the global queue (line 3), it is expanded (line 4), that is, subdivided in more work units (for example a \bigcirc is expanded into possibly several \square) and the results of that expansion are added to the global queue. After this is done, we can start allocating the existing work queues until no more work units are left to allocate (line 7). In order to do that we remove the first work unit (line 8), allocate it to the “current” worker (line 9) and select a new next “current” worker, by going through each one in turn and returning to the first worker after the last one (line 10).

Algorithm 5.4 Static division of work queue using round-robin scheme

Require: Work Queue Q and P processors

Ensure: Partition Q into P different Q_i work queues

```

1: procedure PARTITION( $Q$ )
2:   while  $|Q| < P$  AND  $Q \neq \emptyset$  do
3:      $WU := Q.POP()$ 
4:      $NQ := EXPAND(WU)$ 
5:      $Q.PUSHLAST(NQ)$ 
6:    $i := 1$ 
7:   while  $Q \neq \emptyset$  do
8:      $WU := Q.POP()$ 
9:      $Q_i.PUSHLAST(WU)$ 
10:     $i := 1 + (i \bmod P)$ 

```

Consider an example with 4 workers ($P = 4$) and 6 random networks ($R = 6$) for a $Q = Q_{random} = \{\bigcirc_1, \bigcirc_2, \bigcirc_3, \bigcirc_4, \bigcirc_5, \bigcirc_6\}$. Then the above algorithm would result in the following partition: $Q_1 = \{\bigcirc_1, \bigcirc_5\}$, $Q_2 = \{\bigcirc_2, \bigcirc_6\}$, $Q_3 = \{\bigcirc_3\}$ and $Q_4 = \{\bigcirc_4\}$.

For an example in which the number of initial work units is less than the number of processors, consider a case with 3 workers ($P = 4$) and $Q = Q_{original} = \{\bigcirc_0\}$. Consider also that the original network has 10 vertices. Then, the original work queue will be expanded into $Q = \{\square_{(0,1)}^0, \square_{(0,1)}^1, \square_{(0,1)}^2, \square_{(0,1)}^3, \square_{(0,1)}^4, \square_{(0,1)}^5, \square_{(0,1)}^6, \square_{(0,1)}^7, \square_{(0,1)}^8, \square_{(0,1)}^9\}$ and the algorithm would produce the following partition: $Q_1 = \{\square_{(0,1)}^0, \square_{(0,1)}^3, \square_{(0,1)}^6, \square_{(0,1)}^9\}$, $Q_2 = \{\square_{(0,1)}^1, \square_{(0,1)}^4, \square_{(0,1)}^7\}$ and $Q_3 = \{\square_{(0,1)}^2, \square_{(0,1)}^5, \square_{(0,1)}^8\}$,

CHAPTER 5. PARALLEL NETWORK MOTIF DISCOVERY

No matter which strategy is chosen for the initial division of work, whenever a processor empties its work queue, it will immediately try to obtain more work. What is important here is to give some initial reasonably balanced work to all processors, in order to avoid unnecessary communication in the beginning of the computation.

5.3.5 Work Phase

This phase is the core of the work and where the main computations are made. Its main goal is to ensure that workers are kept busy computing work units, minimizing idle time. We will now describe our three proposed strategies for this phase.

5.3.5.1 Master-Worker

In this case there is a core dedicated exclusively for the load balancing and distribution of work units (the master) and all the other processors (workers) do work and communicate only with the master [HSL⁺00]. This means that in practice if we have P processors, the maximum possible theoretical linear speedup is $P - 1$, the number of workers.

For this strategy it is vital that the master is able to maintain an updated list of unprocessed work units, so that it can serve work requests as soon as possible, thus keeping workers busy. Figure 5.7 overviews this approach.

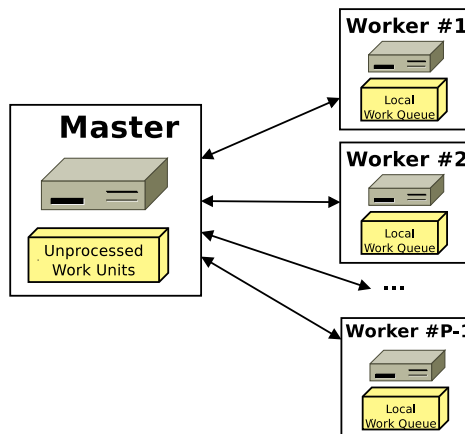


Figure 5.7 – Master-Worker load balancing strategy.

Algorithms 5.5 and 5.6 detail our master-worker strategy. The master (Algorithm 5.5) keeps receiving messages from the workers (line 4) and acts accordingly. If the message

5.3. A GENERAL PARALLEL APPROACH

is a request for more work (line 5) then it can either send the first unprocessed work unit (lines 7 and 8) or, if its work queue Q is empty, add the worker which sent the message to the list of workers that are in need of work (*IdleWorkers*). If the message contains a new unprocessed work unit (line 11), then it either adds it to the work queue Q for future processing (line 16) or sends it directly to an idle worker, in case there is one (lines 12 to 14). If all workers are idle, it means that there are no more unprocessed work units and therefore we can end the work phase (line 3) and broadcast a termination message to all workers (line 17).

Algorithm 5.5 Master procedure for master-worker load balancing

Require: Work Queue Q and P processors

Ensure: Computation of all work units of Q

```

1: procedure MASTER( $Q$ )
2:   IdleWorkers :=  $\emptyset$ 
3:   while (not all workers are idle) do
4:      $msg := \text{RECEIVEMESSAGE}(\text{AnyWorker})$ 
5:     if  $msg.type = \text{RequestForWork}$  then
6:       if  $Q \neq \emptyset$  then
7:          $WU := Q.\text{POPFront}()$ 
8:          $\text{SENDMESSAGE}(msg.Sender, WU)$ 
9:       else
10:        IdleWorkers.PUSHBACK( $msg.Sender$ )
11:      else if  $msg.type = \text{NewWorkUnit}$  then
12:        if IdleWorkers  $\neq \emptyset$  then
13:           $worker := \text{IDLEWORKER}.\text{POPFront}()$ 
14:           $\text{SENDMESSAGE}(worker, msg.WU)$ 
15:        else
16:           $Q.\text{PUSHBACK}(msg.WU)$ 
17:    BROADCASTMESSAGE(Terminate);
```

Regarding the worker execution (Algorithm 5.6), while there is no termination message (lines 2 and 6) it keeps processing its own work queue Q in depth-first manner (line 8). If this queue is empty, then it asks the master for new work (line 4) and waits until a message is received (line 5), adding the newly received work unit to the local queue. Whenever **masterThreshold** is reached (line 9), the worker gives all but one of its unprocessed work units to the master, so that they are distributed among workers

CHAPTER 5. PARALLEL NETWORK MOTIF DISCOVERY

Algorithm 5.6 Worker procedure for master-worker load balancing

Require: Work Queue Q and master processor

Ensure: Computation of all work units

```
1: procedure WORKER( $Q$ )
2:   while NOTFINISHED() do
3:     if  $Q = \emptyset$  then
4:       SENDMESSAGE( $master$ , RequestForWork)
5:        $msg :=$  RECEIVEMESSAGE( $master$ )
6:       if  $msg.type = Terminate$  then EXITWHILE
7:        $Q.PUSHBACK(msg.WU);$ 
8:        $Q.PUSHFRONT(\text{expand}(Q.popFront()))$ 
9:       if CHECKMASTERTHRESHOLD() then
10:        while  $Q.hasMoreThanOneElement()$  do
11:          SENDMESSAGE( $master$ ,  $Q.popBack()$ )
```

that are or will become idle. Note that this threshold is very important. If it is set too high, the work units will not be sufficiently divided in order to adequately balance the work among all workers. If it is too low, work will be divided too soon and the communication costs will increase. We tried two different options for the threshold: either a time limit or a number of work units processed limit. Chapter 6 gives more details on actual parameters chosen.

5.3.5.2 Distributed Queues

In this case all processors are responsible both for the work itself and the load balancing. At any time they can communicate with any other processor and dynamically try to redistribute the work. The maximum possible theoretical linear speedup is P , the number of processors. Figure 5.8 overviews what is done on a distributed strategy.

The **distributed queue** approach follows a receiver-initiated scheme [ELZ85], and all processors run Algorithm 5.7. The basic idea is simple: while a worker still has work units in its queue, it keeps computing them (line 4). If its queue becomes empty, then it asks for work from other processor (line 3) and continues processing the new work units. If at a point in time a consensus is reached that the computation is over, the worker stops (line 2). The other key component is serving work requests from

5.3. A GENERAL PARALLEL APPROACH

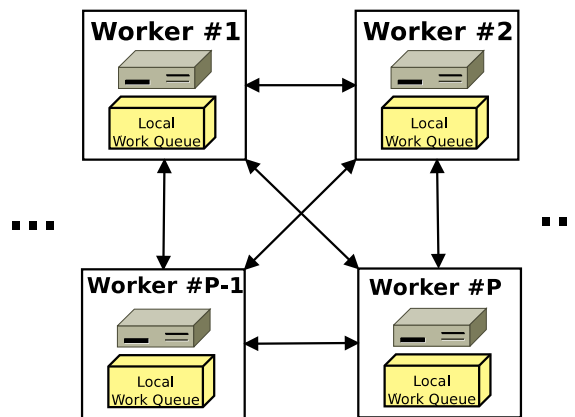


Figure 5.8 – Distributed load balancing strategy.

other workers (line 5).

Algorithm 5.7 Distributed queue main worker procedure.

Require: Work Queue Q

Ensure: Computation of all work units

```

1: procedure DISTRIBUTEDQUEUEWORKER
2:   while NOTFINISHED() do
3:     if  $Q = \emptyset$  then ASKFORWORK()
4:      $Q.PUSHFRONT(\text{expand}(Q.popFront()))$ 
5:     if CHECKMESSAGESTHRESHOLD() then SERVEWORKREQUESTS()

```

We will now explain in more detail the work request mechanism. The first thing to notice is that due to the nature of our desired environment (distributed memory with message passing) there is no way to steal work from another processor without intervention from it. We must send a message and wait for an answer. All processors have a polling mechanism and from time to time (line 5, `checkMessagesThreshold()` function) they will check if there are any incoming requests. This threshold is important and can have an impact on performance. If it is set too low, the receiver worker will be checking for messages too often and will spend valuable execution time trying to serve nonexistent requests. If it is set too high, the sender worker will have to wait for new work while remaining idle, because the receiver will take some time to check for messages.

We tried two different options for the `messagesThreshold` value, as in `masterThreshold` of the master-worker: either a time limit or a number of work units processed.

CHAPTER 5. PARALLEL NETWORK MOTIF DISCOVERY

Chapter 6 gives more details on actual parameters chosen.

Regarding which worker should we try to steal work from, ideally each processor would know the processor that still has more work to do. In a completely distributed environment, that is not possible without introducing major computation overheads. Moreover, since search trees are unbalanced, a worker cannot even have a precise prediction for the amount of computation in its own work queue. Therefore we opted to always choose a random worker to ask work from, which was established as an adequate heuristic [San94, San99].

The third aspect to detail is our strategy for dynamically sharing work in a distributed setting. The main question here is to decide exactly which work units from our work queue should we share whenever a work request is received. The ideal option is to divide as equally as possible the work, in order to maximize the time in which both processors will not need to ask for work again. In order to do that we opted for a diagonal work-queue splitting scheme [RSM03]. Basically we distribute work units in round-robin fashion: one for the sender, one for the receiver, one for sender, and so on. As we are exploiting the search tree in a depth-first order (remember that new expanded work units are pushed to the front of the queue), this will distribute as evenly as possible the work units, taking into account that work units of the same search depth will more likely have similar computational costs. However, since the tree is really unbalanced, this cannot promise equal execution time, but it constitutes our best prediction implemented by a simple yet elegant solution, diagonally distributing work along the search tree.

Figure 5.9 exemplifies our splitting scheme. Dashed work units are yet to be explored. The shaded area corresponds to the work units that will remain in the receiver of the request. The other nodes go to the requester.

We also use a splitting threshold T_{split} , a way of knowing when not to share our work. If it is in fact too small, we may spend more time preparing and sending the work units than really just computing them. We based our threshold on the distance to the g-trie leaf node: as we get closer, our work-unit will take less time. So, if all remaining unexplored vertices are closer to a leaf than T_{split} , we do not divide work and instead send a “no work available” message to the requesting processor. This threshold could even be dynamically adapted as we discover how much time an average work-unit is taking, but in practice a constant value was enough. Chapter 6 describes practical choices for this parameter.

Finally, how do we detect termination? Whenever someone asks for work and receives

5.3. A GENERAL PARALLEL APPROACH

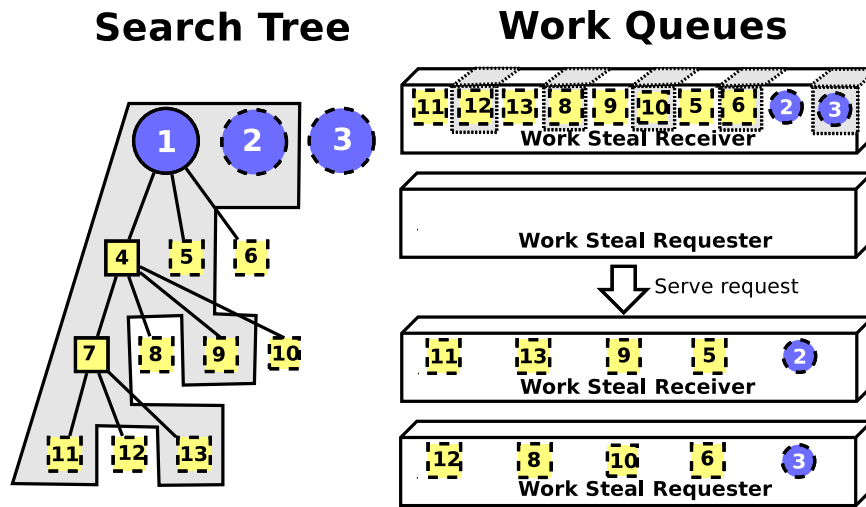


Figure 5.9 – Diagonal work-queue splitting scheme example.

as response that the receiver’s work queue is empty, it will ask another processor. If it happens that everyone answers that it has no more work, the worker will be able to conclude that indeed there are no more work units in the global work queue and it will broadcast a “termination” message to everyone, ending this phase of the parallel job.

5.3.5.3 Distributed Snapshot

When we compare the **master-worker** and **distributed queue** parallel strategies to the original sequential recursive procedures, we can observe that a small execution time overhead is introduced. This is because almost every work unit must be added to the queue and then removed, in order for it to be processed. By contrast, in the original recursive procedure this was taken care by the natural procedural cycles and recursive calls.

Furthermore, if we stop the computation at a given time, we can observe that a memory overhead is introduced. Figure 5.10 shows a possible expansion of a work queue with ESU that starts with only one unit and two steps after already has three units. Note that all of the three units of the step #2 naturally have '1' as their first node of the partial constructed subgraph, this is because they all derive from the same first unit. Similarly, the first two units of step #2 have '1' and '2' as the first vertices of the partial subgraph, since they all derive from the same step #1 unit. When the sequential recursive procedure is being run, this redundant information is taken care of naturally by the call-stack. Note also that as we go further, the work queue tends to

CHAPTER 5. PARALLEL NETWORK MOTIF DISCOVERY

grow exponentially with the branching factor, that is, the number of children of each work unit. This can potentially hinder the parallel computation, not only because of available memory for each processor, but also in terms of communication costs, because work units are being exchanged between processors.

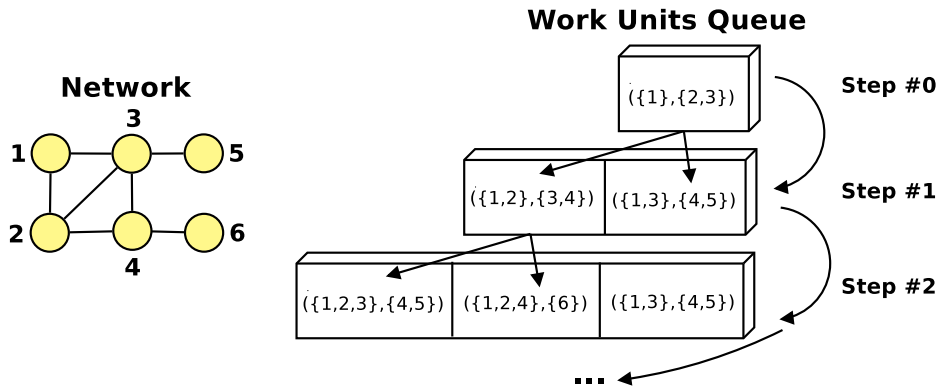


Figure 5.10 – An example expansion of a work queue using ESU algorithm.

We devised a scheme that is able to avoid these undesirable time and memory overheads. First, if the worker still has entire networks unexplored, it will give half of these to the requesting processor, keeping its current network that is being explored. When only one network is being explored, the basic idea is to follow the recursive procedure as in the sequential algorithms and add two main capabilities: the ability to stop the computation at will, efficiently storing the stack search space in what we call a **snapshot** of the computation and the ability to rollback and continue work from a given **snapshot**. This is similar to the concept of checkpointing [WHV⁺95]. The main difference and novelty is however a capability of being able to divide a **snapshot** in two halves that can be continued on two different processors. With this, we can present a new parallel strategy, that we call **distributed snapshot**, as is overviewed in Algorithm 5.8,

Basically, each processor starts by creating a snapshot from its initial share of work (line 2) and then keeps processing it (line 4) until the whole global computation is completed (line 3). While doing this, after a determined threshold is reached (line 11), it checks for incoming messages from other processors (12). If a work request message was received (line 13), the recursive computation is stopped and the search state stored (line 14). Then it uses this state to divide the current search into two different sets of work units (line 6), sending one to the requesting processor (line 7) and keeping one for itself, in order to continue the computation. If there were no work request messages, the recursive search is completed, ending in a situation with no more work-units to

5.3. A GENERAL PARALLEL APPROACH

Algorithm 5.8 Distributed snapshot main worker procedure.

Require: Work Queue Q

Ensure: Computation of all work units

```

1: procedure DISTRIBUTEDSNAPSHOTWORKER( $Q$ )
2:    $S = \text{MAKE\_SNAPSHOT}(Q)$ 
3:   while NOTFINISHED() do
4:     RECURSIVEPROCESS( $S$ )
5:     if RECEIVEDWORKREQUEST() then
6:        $(S, S_2) = \text{DIVIDEWORK}(S)$ 
7:       SENDWORK(requester,  $S_2$ )
8:     if  $S = \emptyset$  then
9:        $W = \text{ASKFORMOREWORK}()$ 
10: procedure RECURSIVEPROCESS( $S$ )
11:   if CHECKMESSAGESTHRESHOLD() then
12:     CHECKMESSAGES()
13:   if RECEIVEDWORKREQUEST() then
14:     stop and store recursive computation
15:   else
16:     keep doing recursive search

```

compute, and therefore the processor starts looking for unprocessed work-units from another processor (line 9).

The main core of the search is the recursive procedure, which will correspond to the respective recursive procedure of the sequential algorithm. As said, a crucial extension is that we must be able to stop and store the state of that recursive computation. For this we must capture the stack contents and we want to do it in an efficient way. We will now detail how we do this both for g-tries and ESU.

G-Trie Snapshots

Figure 5.11 depicts the recursive state of the g-tries matching computation at any given time. Note that the original recursive procedure `matchCond()` (see Algorithm 4.6) is based on two cycles: one enumerates all possible matching vertices and the other enumerates all possible children of the corresponding matching g-trie node. If we freeze time, the search position will therefore be defined by knowing the position where we

CHAPTER 5. PARALLEL NETWORK MOTIF DISCOVERY

are at each of these two cycles in all depths. In the figure, the exact subgraph being matched corresponds to the current position in the two main cycles of the recursive procedure.

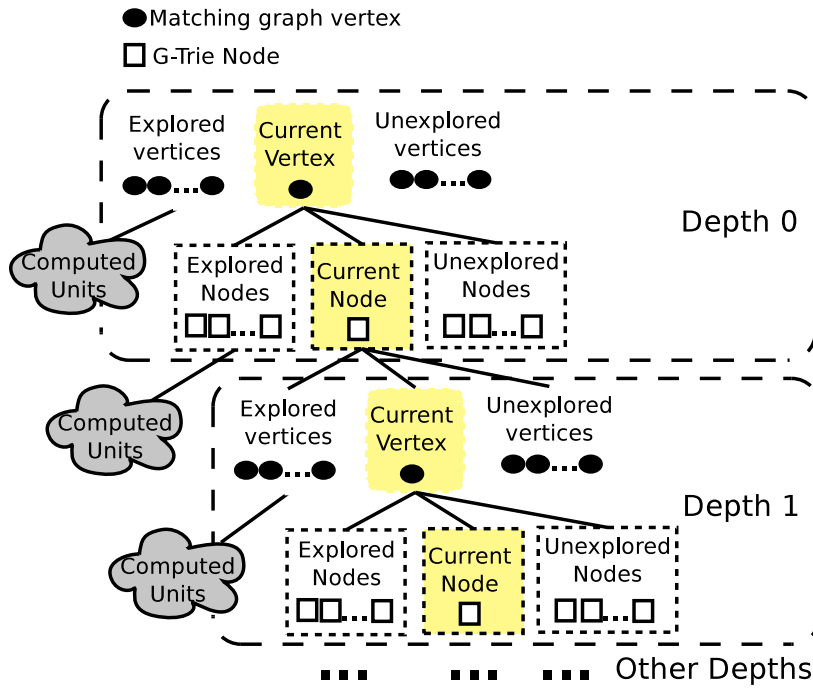


Figure 5.11 – G-Trie recursive procedure frozen at a given time.

In order to store this state, we need to save the cycle position (that is, the current node and vertex) for all depths of the recursive procedure. In the case of a g-trie node, by knowing the current node one can instantly know the nodes that still remain to be explored, since the g-trie is fixed. In the case of the vertices, we must explicitly store the unexplored vertices, because they are dynamic and correspond to real computation work done in the `matchingVertices()` procedure. We can unequivocally identify each g-trie node by a single integer number, and the same can be done with each graph vertex. With this in mind, we create a compact array structure that encapsulates everything that we need in order to later resume the search. This snapshot array is depicted in Figure 5.12.

Algorithm 5.9 details our modified version the `matchCond()` procedure so that it is able to stop and store the recursion whenever a request for work arrives. In this case, it builds the correspondent snapshot and stops the search.

New lines are indicated in the code. What changes is that now we keep checking for messages (lines 2 and 3), and when a request for work is received, we stop making

5.3. A GENERAL PARALLEL APPROACH

D - Larger recursive depth

V_i - current graph vertex being explored at depth i

N_i - current g-trie being explored depth i

U_i - number of unexplored vertices in depth i

UV_i^j - j -th unexplored vertices in depth i

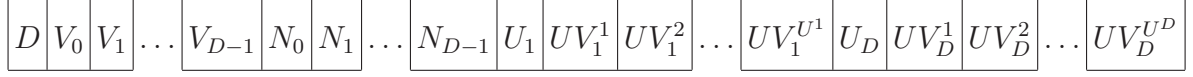


Figure 5.12 – An array structure representing a g-trie snapshot.

Algorithm 5.9 Distributed snapshot version of g-trie recursive procedure.

```

1: procedure SNAPSHOTMATCH( $T, V_{used}$ )
2:   if CHECKMESSAGESTHRESHOLD() then                                ▷ NEW CODE
3:     CHECKMESSAGES()                                                ▷ NEW CODE
4:    $V = \text{MATCHINGVERTICESCOND}(T, V_{used})$ 
5:   for all node  $v$  of  $V$  do
6:     if RECEIVEDWORKREQUEST() then                                ▷ NEW CODE
7:        $snapshot.ADD(\text{remaining nodes of } V)$ ; break                ▷ NEW CODE
8:     if  $T.isGraph$  then FOUNDMATCH()
9:     for all children  $c$  of  $T$  do
10:      if RECEIVEDWORKREQUEST() then                                ▷ NEW CODE
11:         $snapshot.ADD(\text{current children } c)$ ; break                ▷ NEW CODE
12:      SNAPSHOTMATCH( $c, V_{used} \cup \{v\}$ )

```

recursive calls, break all cycles, and build the snapshot array (lines 6, 7, 10 and 11).

Algorithm 5.10 shows how to resume a given snapshot S , assuming the notation given in Figure 5.12. We also assume that the child nodes of a g-trie node are ordered and that by *bigger siblings* we mean the g-tries nodes that share the same direct ancestor node, that is, the same parent, and that are bigger in that order context.

We start by creating the set of used vertices, V_{used} as in `snapshotMatch()` (line 2). We then traverse all recursive depths, from the highest to the lowest (line 3), since it would be in that order that they would be computed in the original recursive procedure. Then, we traverse all remaining g-trie nodes (line 5), as it would happen in the inner cycle of `snapshotMatch()`, in order to simulate the continuation of the cycle that we stopped. We then follow with all remaining unexplored nodes of that depth (line 9),

CHAPTER 5. PARALLEL NETWORK MOTIF DISCOVERY

Algorithm 5.10 Resuming a g-trie snapshot.

```

1: procedure RESUMEGTRIESNAPSHOT( $S$ )
2:    $V_{used} = \{V_0, \dots, V_{D-1}\}$ 
3:   for  $i$ :  $D - 1$  down to 0 do
4:      $N_{remaining} = N_i \cup \text{BIGGERSIBBLINGS}(N_i)$ 
5:     for all  $C$  in  $N_{remaining}$  do
6:       if RECEIVEDWORKREQUEST() then
7:          $state.ADD(\text{remaining children})$ ; break;
8:       SNAPSHOTMATCH( $C, V_{used}$ )
9:     for  $j$ : 1 to  $U^i$  do
10:      if RECEIVEDWORKREQUEST() then
11:         $state.ADD(\text{remaining nodes})$ ; break;
12:      remove last element from  $V_{used}$ 
13:      add  $UV_i^j$  to end of  $V_{used}$ 
14:      if  $N_i.isGraph$  then FOUNDMATCH()
15:      for all children  $C$  of FATHER( $N_i$ ) do
16:        if RECEIVEDWORKREQUEST() then
17:           $state.ADD(\text{remaining children})$ ; break;
18:        SNAPSHOTMATCH( $C, V_{used}$ )
19:      remove last element from  $V_{used}$ 

```

proceed by updating V_{used} accordingly (line 12 and 13), and then traverse all possible g-trie nodes from that search position (lines 15). In all of these cases, continuation of computation itself is done by calling the original `snapshotMatch()` procedure (lines 8 and 18). As in the original procedure, if the computation has to stop, we update the respective snapshot array (lines 6, 7, 16 and 17).

The whole `resumeGTrieSnapshot()` is done in order to simulate what would happen if the original recursive procedure kept computing. In fact, if we would artificially stop `snapshotMatch()`, and then resume work with `resumeGTrieSnapshot()`, we would obtain the exact same results with almost no performance loss. This is due to our efficient snapshot array structure, that minimizes the information needed to continue, restricting it to the bare essential. In fact, the maximum theoretical size of the snapshot array is $O(\max_depth_gtrie \times |V(G)|)$, since we can only go as far as the maximum g-trie depth, and each depth always has at maximum all nodes as unexplored. In

5.3. A GENERAL PARALLEL APPROACH

practice, the work array will be kept, with very high probability, much lower than this maximum, since all the constraints will reduce the possible candidates.

The fact that our array is small sized is also beneficial because it means we can easily communicate the array to another processor that issued a request for work. The idea is that, upon receiving such a request, the current processor partitions its work array in two valid pieces, continues to compute with one of them and dispatches the other for the requesting processor. In order to maintain the computation balanced, it is crucial that a processor divides his work array as equally as possible. To achieve this goal, we partition the work array in two halves, by equally dividing all unexplored vertices in the following way, where the number of a vertex is its position on the array structure:

- **1st Half** maintains currently explored nodes and vertices; gets even numbered unexplored vertices.
- **2nd Half** gets odd numbered unexplored vertices.

This configures a *diagonal* split, meaning that we traverse our tree search space diagonally, as was the case in the `distributed queue` parallel strategy. This always provides two equally sized halves in terms of the number of unexplored vertices.

ESU Snapshots

Figure 5.13 depicts the recursive state of the ESU matching computation at any given time. It is essentially a simpler version of a g-trie snapshot, since the original recursive procedure `extendSubgraph()` (see Algorithm 3.3) is based on just one cycle, that traverses a list of possible expansion vertices. By freezing time, the search position is defined by the position on this cycle on all depths. The exact subgraph being matched corresponds to the set formed by the current vertex of each cycle.

In order to store this state we need only to save the cycle position for all depths of the recursive procedure. We must also store the unexplored vertices, that is, the vertices of the corresponding expansion list that were still not traversed. This is encapsulated in the compact array structure of Figure 5.14. The maximum theoretical size of this snapshot array is $O(max_subgraph_size \times |V(G)|)$, since we can only go as far as the maximum subgraph size, and each depth always has at maximum all nodes as unexplored. Like in g-tries snapshots, in practice, the actual contents will be much lower than this maximum due to the constraints applied on the expansion list of vertices.

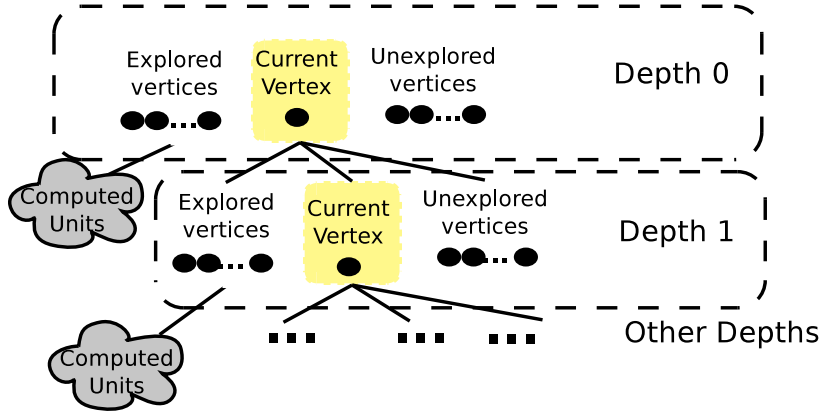


Figure 5.13 – ESU recursive procedure frozen at a given time.

D - Larger recursive depth

V_i - currently graph vertex being explored at depth i

U_i - number of unexplored vertices in depth i

UV_i^j - j -th unexplored vertices in depth i

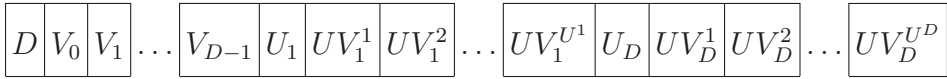


Figure 5.14 – An array structure representing an ESU snapshot.

Algorithm 5.11 details our adaptation of the `ESU extendSubgraph()` recursive procedure so that it is able to stop and store the recursion whenever a request for work arrives, building an ESU snapshot. It is the same as the initial sequential algorithm with the exception of the marked lines, with the already explained checking of new messages (lines 2 and 3) and the storage of the current vertex (line 12) and unexplored vertices (line 13).

Algorithm 5.12 shows how to resume a given ESU snapshot S , assuming the notation given in Figure 5.14. It starts by creating the partially extended subgraph V_{subg} (line 2). Then it traverses all recursive depths, from the highest to the lowest (line 3), since it would be in that order that they would be computed in the original recursive procedure. After this, it builds the correspondent expansion list (line 7) and does exactly as we would do in the `snapshotExtendSubgraph()` recursive algorithm (lines 10 to 16). If the computation has to stop again, the respective snapshot array is updated (lines 4 to 5 and 13 to 16).

5.3. A GENERAL PARALLEL APPROACH

Algorithm 5.11 Distributed snapshot version of ESU enumeration of subgraphs

```

1: procedure SNAPSHOTEXTENDSUBGRAPH( $V_{Subg}, V_{Ext}, v$ )
2:   if CHECKMESSAGESTHRESHOLD() then ▷ NEW CODE
3:     CHECKMESSAGES() ▷ NEW CODE
4:   if  $|V_{Subg}| = k$  then
5:     INCREMENTCOUNT(canonicalLabeling( $V_{Subg}$ ))
6:   else
7:     while  $V_{Ext} \neq \emptyset$  do
8:       remove random chosen  $w \in V_{Ext}$ 
9:        $V'_{ext} := V_{ext} \cup \{u \in N_{excl}(w, V_{subg}) : u > firstVertex(V_{Subg})\}$ 
10:      SNAPSHOTEXTENDSUBGRAPH( $V_{subg} \cup \{w\}, V'_{ext}, v$ )
11:      if RECEIVEDWORKREQUEST() then ▷ NEW CODE
12:         $snapshot.ADD(\text{current vertex } w)$  ▷ NEW CODE
13:         $snapshot.ADD(\text{remaining nodes of } V_{Ext})$  ▷ NEW CODE
14:        break ▷ NEW CODE

```

Algorithm 5.12 Resuming an ESU snapshot.

```

1: procedure RESUMEESUSNAPSHOT( $S$ )
2:    $V_{subg} := \{V_0, \dots, V_{D-1}\}$ 
3:   for  $i$ :  $D - 1$  down to 0 do
4:     if RECEIVEDWORKREQUEST() then
5:        $snapshot.ADD(\text{depth } i \text{ of } S)$ 
6:     else
7:        $V_{ext} := \{UV_i^0, \dots, UV_i^{U_i}\}$ 
8:       remove last vertex from  $V_{subg}$ 
9:       while  $V_{Ext} \neq \emptyset$  do
10:        remove random chosen  $w \in V_{Ext}$ 
11:         $V'_{ext} := V_{ext} \cup \{u \in N_{excl}(w, V_{subg}) : u > firstVertex(V_{Subg})\}$ 
12:        SNAPSHOTEXTENDSUBGRAPH( $V_{subg} \cup \{w\}, V'_{ext}, v$ )
13:        if RECEIVEDWORKREQUEST() then
14:           $snapshot.ADD(\text{current vertex } w)$ 
15:           $snapshot.ADD(\text{remaining nodes of } V_{Ext})$ 
16:          break

```

CHAPTER 5. PARALLEL NETWORK MOTIF DISCOVERY

This whole `resumeESUSnapshot` is able to resume work exactly as it would have originally run, as in the case of g-tries snapshots. The partition is also similar, with the list of unexplored vertices being divided in two for each depth: current and even unexplored vertices one one half and odd unexplored vertices on the other half. This configures again a *diagonal* split, with two equally sized halves in terms of the number of unexplored vertices.

5.3.6 Aggregation Phase

After the work-phase has ended, every worker will have its own dictionary of frequencies for every network analyzed (the original one and the random ones) and subgraphs discovered. There will probably exist many zeroes, meaning that a particular network was not analyzed at all by that particular worker, but potentially there can be valuable information for every worker, for any subgraph, on any network. This is a huge amount of data that we need to aggregate in order to calculate the subgraph significance. Note that the number of possible k -subgraphs grows super-exponentially as k increases.

For each class of isomorphic subgraphs, we need to know the frequency in the original network and its average frequency and standard deviation in the set of random networks. We choose to have a “root” worker, responsible for storing the global results. After gathering all needed frequencies, this worker can calculate the necessary subgraph significance in a sequential manner (note that, as said, the majority of the work is done during the computation of the census and that at this phase, since a computation of a single subgraph significance is done in almost constant time, it would not improve if it was sent to another worker - on the contrary it would take more time).

A simple primitive approach for this would be for each worker to communicate in turn with the root worker, sending its own results. In order to do that, it would have to send pairs of subgraph descriptions (for example using the canonical labeling) and their respective frequencies. We call this strategy **naive**.

This naive approach is not enough and can take a huge amount of time. Instead, our strategy is to first agree on the list of subgraphs that are being computed. If all workers have that list before communicating its own frequencies found, we could avoid the need to communicate graph identifications, because we can induce a fixed order of relevant subgraph types. Therefore, if we communicate a vector of frequencies, the position in the vector will determine which subgraph we are referring too.

5.3. A GENERAL PARALLEL APPROACH

In order to create that list of relevant subgraph types, depending on the subgraph frequency discovery method being used, we do the following:

- **g-tries:** in this case the list of relevant subgraphs is already at our disposal in the g-trie, and we assume that the subgraphs come in the depth-first order of traversing that g-trie.
- **ESU:** the root worker will start by advertising a set of T concrete subgraphs that it knows are being computed (that is, the ones whose frequency it has found to be greater than zero), by broadcasting a message. Then, all workers, organized in a binary tree (see an example in Figure 5.15), communicate to tree ancestors their list of subgraph types not found by the root worker. After this process is complete, the root will have a list of new subgraphs that it will broadcast to all workers.

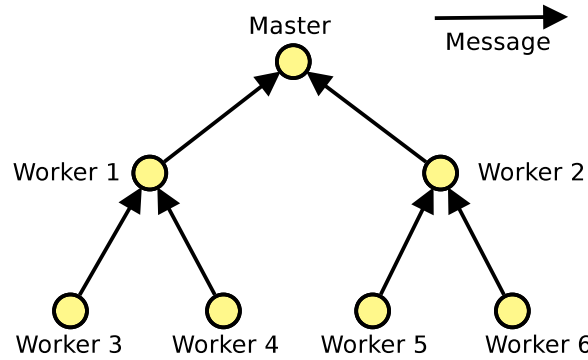


Figure 5.15 – 7 processors organized in binary search tree.

Having the pre-defined list of subgraphs at hand, the workers can just communicate frequencies (and not graph descriptions). For the this step we propose two strategies:

- **hierachical:** like in the agreement of subgraph types being computed, we organize the workers in a binary tree. Each worker receives an array of frequencies from its descendants, adds it to its own array and in turn communicates it to the ancestor worker. Note that this has the potential to logarithmically cut the needed time when compared to a naive sequential communication of frequencies.
- **collective:** instead of using point-to-point messages (from one worker to another worker), we use the specialized MPI collective communications facilities. We use `MPI_Reduce` to gather and sum all frequency values in a vector. The position in

CHAPTER 5. PARALLEL NETWORK MOTIF DISCOVERY

the vector denotes which network and subgraph type it refers to. The specific algorithm used by `MPI_Reduce` is an implementation defined issue and can vary even for two different versions of the same MPI suite. Traditionally it also uses a tree [Hus99] but it can also incorporate more advanced features such as taking advantage of node locality. The main aspect to notice is that we delegate to the MPI implementation the gathering of information, by explicitly denoting that we want to aggregate and sum the frequencies.

5.3.7 Parallel Sampling

The g-trie and ESU algorithm also allow for sampling and we have implemented that option, providing a quicker but only approximate parallel calculation of motifs. The basic idea is to only follow each search branch with a probability P_i related to the depth i as described before in Sections 3.3.2 and 4.6.

Essentially the algorithms remain the same in all the cases with the following exceptions:

- **master-worker and distributed queue:** an unit $\square_{(id,s)}$ is only expanded with a probability associated to its size s , which corresponds to the respective recursive depth.
- **distributed snapshot:** the `snapshotMatch()` and `snapshotExtendSubgraph()` are only called with a probability associated to the recursive depth, which is as before equal to the size of the partially constructed subgraph.

With this in position, when looking for k -subgraphs, each possible occurrence will only be found with probability $P_0 \times P_1 \times \dots \times P_{(k-1)}$. This leads to an even more unpredictable search tree topology, since a branch can sometimes be completely eliminated because it was not selected in the respective probabilistic test. The dynamic load balancing scheme will then ensure that this workload is distributed among all the processors.

5.3.8 Motif Discovery

With all the algorithms described before in this chapter it is now possible to discover motifs in parallel, in three synchronization steps, as was depicted in Figure 5.2.

1. Compute the frequency of subgraphs of a given size k in the original network, using either parallel ESU or parallel gtries, with an initial g-trie containing all the possible k -subgraphs. This will generate a list L of interesting subgraphs.
2. Compute the frequency of the subgraphs of L in all the random network using parallel g-tries, with L being used to create the g-trie. This will originate a list of frequencies in each network.
3. Compute the significance of each subgraph in serial, since each significance computation takes only a very small fraction of time and doing it in parallel would be detrimental.

This configures a direct parallel equivalent of the g-trie motif discovery methods given in Section 4.7. The general applicability of our parallelization methodology makes it also possible another parallel alternative, which is to compute the frequency of all k -subgraphs at the same time in the original and random networks, using only parallel ESU. This is the parallel equivalent to discovering motifs using only ESU.

5.4 Summary

In this chapter we first identified interesting opportunities for parallelism in the sequential motif discovery program flow and created an associated taxonomy. We then abstracted the whole process as a tree shaped computation with independent branches. Using that as a basis we presented a general parallel methodology with dynamic load balancing capabilities, with both centralized and distributed control. We have also added the capability to freeze the computation, creating an efficient way of storing, dividing and resuming a snapshot of the search state. Finally we have added the option to use sampling to further reduce execution time and we have shown how all of this could be used to efficiently discover motifs in parallel.

*The difference between theory and practice
is that in theory, there is no difference
between theory and practice.*

Richard Moore

6

Experimental Evaluation

In this chapter we present empirical data obtained by running our sequential and parallel methods on a large and representative set of synthetic and complex networks. We first detail the computational environment and the networks used, and follow with a characterization of the g-tries sequential method. We then compare the behaviour of g-tries with the other state-of-art motif discovery algorithms, in both exhaustive and approximate (with sampling) scenarios. Finally, we do a performance evaluation and scalability study of the parallel algorithms and conclude with an overview of the main findings.

6.1 Common Materials

We organized our experimental evaluation in two major parts: one deals with sequential algorithms, the other deals with parallel algorithms. However there are aspects, such as the computing the environment and the datasets used, that are common to both parts. These are described next.

6.1.1 Computational Environment

All experimental results were obtained on a dedicated cluster with 12 SuperMicro Twinview Servers for a total of 24 nodes. Each node has 2 quad-core Xeon 5335

CHAPTER 6. EXPERIMENTAL EVALUATION

processors and 12 GB of RAM, totaling 192 cores, 288 GB of RAM, and 3.8TB of disk space, using Infiniband interconnect.

All code was developed in C++ and compiled with gcc 4.1.2. For message passing we used OpenMPI 1.2.7 [GWS06]. Execution times were measured using the `gettimeofday()` function. They are wall clock times, that is, they measure real time elapsed from the start to the end of the respective computation. The time unit used is the second.

6.1.2 Complex Networks

There is no generally accepted set of benchmark networks for motif discovery experimentation. This is mainly because motifs are so ubiquitous that they can be applied to any system that can be modeled as a graph and therefore the possible applications areas are almost infinite and from many scientific fields. Much research work has been done already related to application-specific synthetic networks [WP08], such as community detection algorithms [LFR08a], however, with motifs, we are aiming for the most general applicability.

Our option was thus to use a large set of real representative networks from several domains, focusing on getting diversity in topological features and scientific fields. We divided the real networks that will be used in four big groups. Next, we describe each of these networks, provide a name for future reference and indicate where the data set was obtained.

- **Biological networks:** model biological processes and complex systems and are gaining increasing attention [Alo03].
 - **ppi:** an undirected budding yeast (*S. cerevisiae*) protein-protein interaction (PPI) network [BZC⁺03]. PPI networks are one of the most intensely studied, and model how proteins bind together to carry out many important biological functions. Source: [BM06].
 - **neural:** a directed neural network of the small nematode roundworm *C. elegans*, describing its nervous system [WSTB86, WS98]. Source: [New09].
 - **metabolic:** a directed metabolic network of the small nematode roundworm *C. elegans* [DA05]. Source: [Are11].

6.1. COMMON MATERIALS

- **Social networks:** describe interactions between individuals or entities, such as friendships, common interests or financial exchanges. Social network analysis is nowadays a key technique in sociology [WFI94], with several scientific conferences exclusively devoted to it.
 - **coauthors:** an undirected network describing coauthorship between scientists working on network theory [New06]. Source: [New09].
 - **dolphins:** an undirected social network of frequent associations between 62 dolphins in a community living near New Zealand [LSB⁺03]. Source: [New09].
 - **links:** a directed network of hyperlinks between weblogs on US politics [AG05]. Source: [New09].
 - **company:** a directed ownership network of companies in the telecommunications and media industries [NLGC02]. Source: [BM06].
- **Physical networks:** describe networks which have a physical real-world representation, such as computer or transportation networks, where each node's geographical location may influence the topology of the network.
 - **circuit:** an undirected network representing an electronic circuit. Source: auxiliary material of [MSOI⁺02].
 - **power:** an undirected network representing the topology of the Western States Power Grid of the United States of America [WS98]. Source: [New09].
 - **internet:** a symmetrized snapshot of the structure of the Internet at the level of autonomous systems, reconstructed from BGP tables posted by the University of Oregon Route Views Project. This data was compiled by Mark Newman in July, 2006. Source: [New09].
- **Semantic networks:** represent connections between concepts.
 - **foldoc:** Foldoc is an online dictionary of computing terms [How10]. This is a directed network where an edge (X, Y) means that term Y is used to describe the meaning of term X . Source: [BM06].
 - **oldis:** Oldis is the Online Dictionary of Library and Information Science [Rei02]. It is a directed network built in the same way as **foldoc**. Source: [BM06].

CHAPTER 6. EXPERIMENTAL EVALUATION

Table 6.1 summarizes the topological features of all these networks. They are shown ordered by increasing number of nodes. All networks are simple unweighted graphs, in accordance with the formalization of the motif discovery problem we use in this thesis. Some simplifications on the original networks were done as necessary. Self-loops were discarded, multiple edges were transformed in a simple unique edge between the two nodes, and all weights were ignored.

Network	Group	Directed	$ V(G) $	$ E(G) $	Nr. of Neighbours	
					Average	Max
dolphins	social	no	62	159	5.1	12
circuit	physical	no	252	399	3.2	14
neural	biological	yes	297	2,345	14.5	134
metabolic	biological	yes	453	2,025	8.9	237
links	social	yes	1,490	19,022	22.4	351
coauthors	social	no	1,589	2,742	3.5	34
ppi	biological	no	2,361	6,646	5.6	64
odlis	semantic	yes	2,909	18,241	11.3	592
power	physical	no	4,941	6,594	2.7	19
company	social	yes	8,497	6,724	1.6	552
foldoc	semantic	yes	13,356	120,238	13.7	728
internet	physical	no	22,963	48,436	4.2	2,390

Table 6.1 – Topological features of the networks used to test the algorithms.

For each network we also provide the average and the maximum number of neighbours per node, since this can provide an insight on how the number of subgraph occurrences grows as its size increases. Generally speaking, the more neighbours a single node has, the more subgraph occurrences it will participate in. Therefore, a larger number of neighbours implies a larger growth ratio on the number of subgraph occurrences.

In order to perform some systematic tests on our algorithms, we also use synthetic networks, artificially generated to present some topological constraints. In particular we will use a benchmark social network proposed by Lancichinetti et al. [LFR08b], that was originally created with the purpose of evaluating community detection algorithms. It provides undirected or directed networks with features close to a real social network and is fully customizable.

Since we are using a very large set of possible networks, we will not use all of them for every aspect tested. For the sake of simplicity and ease of reading, some sections will only present some representative cases.

6.1.3 Other Competing Algorithms

We will compare the performance of our algorithm against the main state-of-art algorithms that were designed to compute the same task. These were described in detail in Chapter 3 and, in particular, Section 3.2 gives the pseudo-code for how they compute the exact subgraph census of a network.

`mfinder` is not used since it is significantly slower, by several orders of magnitude, than all other major algorithms, as it is shown for example in [Wer06]. For instance, we empirically verified that it was more than 1000x slower than `g-tries` when computing the 5-subgraph census of the undirected network `ppi` (an undirected network) and that it was more than 2000x slower than `g-tries` when computing the 4-subgraph census of the directed network `company`.

In order to compare with the network-centric approach, we will use both the `ESU` and `Kavosh` algorithms. We use our own implementation of the algorithms, so that the whole code infra-structure is the same with the exception of the algorithm itself, allowing for better and fairer comparison. For example, both `ESU` and `Kavosh` use the `nauty` third-part algorithm for isomorphism calculation, but the author's implementation use different versions of it. In our implementation, we can make sure that this is not the case and the same `nauty` version is used in both algorithms.

Great care was taken in the implementation of these algorithms, with special emphasis on guaranteeing the best performance possible. We also downloaded, compiled and used as a guide the original source code provided by the authors of `ESU` (fanmod tool¹) and `Kavosh`², and compared the execution times with our own implementation, to certify that we were not slowing down the algorithms. Both original implementations use C++.

Table 6.2 shows this comparison. The Fanmod tool has a mandatory graphical user interface that is not available at the dedicated cluster we used. Therefore, for this experiment, we used a personal computer with an Intel Core i5-450M processor running at 2.4GHz with 4GB of DDR3 memory. We show the execution times (measured in seconds) for a complete k -subgraph census on four networks, two undirected and two directed, with increasing k .

We can observe that our `Kavosh` implementation is very similar to the original implementation, with a slight gain. With respect to `ESU`, our implementation is faster than

¹Fanmod is available at <http://theinf1.informatik.uni-jena.de/~wernicke/motifs/>

²Kavosh source code is available in <http://lbb.ut.ac.ir/Download/LBBsoft/Kavosh/>

CHAPTER 6. EXPERIMENTAL EVALUATION

network	neural			metabolic			coauthors			power		
k	3	4	5	3	4	5	3	4	5	3	4	5
Original Kavosh	0.03	1.29	70.84	0.05	3.22	283.55	0.01	0.06	0.46	0.01	0.07	0.35
Our Kavosh	0.03	1.26	63.12	0.05	3.35	272.67	0.01	0.05	0.41	0.01	0.07	0.31
Original ESU	0.08	2.09	76.64	0.12	5.66	386.52	0.02	0.17	1.16	0.04	0.18	0.81
Our ESU	0.03	1.29	69.27	0.04	3.15	270.4	0.01	0.05	0.41	0.02	0.07	0.31

Table 6.2 – Execution time of our ESU and Kavosh implementations.

the original, with an average speedup close to 2x. This gain seems to be even more pronounced in the undirected networks (**coauthors** and **power**).

Moreover, our own implementation of ESU does not impose strict limits on the motif size, while the original source code is limited to 8 nodes. We also implemented **RAND-ESU**, the sampling version of this algorithm.

Regarding the subgraph-centric approach we will use **Grochow**. However, given that **Grochow** is implemented in Java, we do not show a comparison with our implementation. Generally speaking, Java is considered to be slower than C++ [K08], although each case may have its particular behaviour. Nevertheless, our own implementation provides a much fairer comparison to the others, in the sense that it is written in the same language and with the same code-infrastructure.

MODA is not used for comparison since it has a very high memory cost since it must store subgraph occurrences, as these are needed for all possible k -subgraphs, in order to build the complete expansion tree for the desired size. This hinders its applicability, specially when the number of subgraphs found becomes large. Instead, we just compare with algorithms that base their computation only on storing subgraph counts, as is the case with **g-tries**, **ESU**, **Kavosh** and **Grochow**.

6.2 Sequential Algorithms

This section shows results obtained with sequential algorithms for motif discovery, starting with a more detailed analysis of **g-tries** and then with a comparison with the main competitor algorithms.

6.2. SEQUENTIAL ALGORITHMS

6.2.1 G-Tries Creation

We will start by evaluating the time it takes to construct a g-trie and its ability to compress the topological information of the original set of graphs.

We first generated all possible simple undirected k -graphs, increasing k , using the `nauty` tools. We then experimented reading all those graphs from a file and inserting them in an initially empty g-trie. No step was bypassed, such as computing the symmetry breaking conditions, and filtering these conditions in the g-trie. Note that on a real usage case, this could be avoided by pre-computing these conditions, but the intention of this experiment is to show how much time it takes to create a g-trie on the fly.

Table 6.3 shows the results obtained. We took note on the number of graphs, the compression ratio (see Equation 4.1), the time it takes to create the g-tries and the average number of subgraphs stored per second. We stop at the first k where the number of different graphs is greater than one million. The smallest k used throughout this chapter is 3, since those are the simplest subgraphs and considering less would correspond to look for single vertices ($k = 1$) or for edges ($k = 2$).

k	Nr. Graphs	Compression	Time (s)	Graphs/sec
3	2	33.33%	< 0.001	37,736
4	6	58.33%	< 0.001	35,088
5	21	70.48%	< 0.001	29,494
6	112	77.98%	0.004	25,524
7	853	82.21%	0.040	21,514
8	11,117	85.01%	0.588	18,917
9	261,080	87.12%	15.766	16,559
10	11,716,571	88.78%	917.379	12,772

Table 6.3 – Execution time for inserting all undirected k -graphs in a g-trie.

Table 6.3 shows that the number of k -graphs grows exponentially and, at the same time, the compression also increases, meaning that more common substructure is identified. This is mainly because there are more graphs and therefore more potential for overlapping of structure. For example, all subgraphs will naturally share the same root g-trie ancestor, containing a single node without a connection to itself. Regarding the execution time, we can see that is initially almost instantaneous, but it grows exponentially with the number of subgraphs. The results show that this approach

CHAPTER 6. EXPERIMENTAL EVALUATION

could become prohibitive as k grows, but it should be noted that the number of graphs would become so large that even the g-trie itself containing all k -graphs would be too big to fit in main memory.

We now show the results for creating a g-trie with all directed k -graphs in the same way, stopping again as soon as the number of graphs exceeds one million. The results can be seen in Table 6.4.

k	Nr. Graphs	Compression	Time (s)	Graphs/sec
3	13	56.41%	< 0.001	156,627
4	199	71.98%	0.002	83,789
5	9,364	79.07%	0.150	62,535
6	1,530,843	83.03%	50.254	30,462

Table 6.4 – Execution time for inserting all directed k -graphs in a g-trie.

The results are very similar to what happens with undirected networks, with growing compression and again an exponential growth in number of graphs and execution time, exposing the same virtues and limitations.

Remember that these complete g-tries containing all possible k -graphs can be reused and they could be stored in the file system ready to be uploaded to main memory without really computing them, as was described in Section 4.4.4. Tables 6.5 and 6.6 show execution times when reading the files of g-tries containing all possible k -graphs. Again we show the number of graphs and the ratio of graphs per second. In this experiment we also take note of the size of the file (in bytes) containing the respective g-trie, and the number of bytes per graph.

We can observe that reading the previously computed g-trie from a file is obviously much faster than creating the g-trie on the fly, and the main bottleneck is the file size itself. The number of necessary bytes per graph decreases as k increases since the compression ratio also increases.

For motif computation, as discussed before, we will typically find all k -subgraphs of the original network and then populate a g-trie only with those that both appear in the network and are meaningful for motif computation, that is, that appear at least a given minimum of times. This means that in practice we will be inserting a dynamic set of graphs, which is generally just a small percentage of the whole set of possible subgraphs of a determined size, which would mean much smaller creation times. Note that some of the computational costs could be further reduced, namely

6.2. SEQUENTIAL ALGORITHMS

k	Nr. Graphs	Time (s)	Graphs/sec	File Size (bytes)	Bytes/graph
3	2	< 0.001	83,333	42	21.0
4	6	< 0.001	193,548	97	16.2
5	21	< 0.001	437,500	281	13.4
6	112	< 0.001	783,217	1,287	11.5
7	853	< 0.001	1,055,693	8,935	10.5
8	11,117	0.010	1,059,670	100,182	9.0
9	261,080	0.209	1,249,288	2,020,172	7.7
10	11,716,571	8.277	1,415,603	79,571,853	6.8

Table 6.5 – Execution time for reading a g-trie with all undirected k -graphs.

k	Nr. Graphs	Time (s)	Graphs/sec	File Size (bytes)	Bytes/graph
3	13	< 0.001	351,351	126	9.7
4	199	< 0.001	1,463,235	1,398	7.0
5	9,364	0.004	2,084,131	54,354	5.8
6	1,530,843	1.384	1,106,481	8,113,436	5.3

Table 6.6 – Execution time for reading a g-trie with all directed k -graphs.

be pre-computing the symmetry conditions, avoiding the necessity of computing the automorphisms.

In practice, this means that we can achieve a very fast g-trie creation, with the g-trie creation step not being the bottleneck in the motif computation. In fact, this time will only start to be more meaningful as the number of graphs reaches a number so high that their g-trie representation cannot be stored in main memory. This is the main limitation of the g-trie creation process as it is right now, since all algorithms assume that the g-trie in itself fits in memory.

Given this, for the last experiment on this section, we took note of the total memory spent by a g-trie residing in memory during motif computation, using the `valgrind` tool [NS07]. Tables 6.8 and 6.7 show the amount of memory needed for a g-trie containing all possible k -graphs, as well as the number of stored graphs and the average memory per graph.

We can observe that, as expected, the needed memory grows exponentially with the number of stored graphs. The average memory per graph decreases as k grows because of larger compression ratios. When comparing this with the number of bytes per graph

CHAPTER 6. EXPERIMENTAL EVALUATION

k	Nr. Graphs	Memory (bytes)	Bytes/graph
3	2	814	407.0
4	6	2,294	382.3
5	21	7,172	341.5
6	112	34,852	311.2
7	853	241,294	282.9
8	11,117	2,899,538	260.8
9	261,080	63,504,120	243.2
10	11,716,571	2,680,803,240	228.8

Table 6.7 – Memory needed for a g-trie containing all undirected k -graphs.

k	Nr. Graphs	Memory (bytes)	Bytes/graph
3	13	3,004	231.1
4	199	38,210	192.0
5	9,364	1,635,190	174.6
6	1,530,843	260,796,274	170.4

Table 6.8 – Memory needed for a g-trie containing all directed k -graphs.

needed to store the g-trie in a file, we can see that the actual memory needed for the computation itself is much larger. Several factors contribute for this, such as the overhead introduced by using C++ objects or the added extra information, like the actual frequencies found.

6.2.2 G-Tries Census

This section will focus on examining the algorithm that computes the census of the subgraphs stored in a g-trie.

6.2.2.1 Effect of Canonical Labeling

As explained in detail in Section 4.4.2, the canonical representation will influence the topology of the g-trie and therefore will have a great impact in the efficiency of the census algorithm. In order to empirically justify our option for the canonical labeling, we show the execution time of a complete k -subgraph census using a g-trie created

6.2. SEQUENTIAL ALGORITHMS

with four different methods:

- **GTCanon:** our custom built **GTCanon** labeling algorithm, as described, that favors in lower levels nodes with more connections to ancestor nodes in the g-trie.
- **Larger:** the lexicographically largest possible adjacency matrix, that will induce high compression in the g-trie.
- **Random:** a deterministic pseudo-random labeling (obtained by fixing the seed) applied after the **nauty** canonization. We create it by a sequential random choice of a node that is connected to an ancestor node (when possible). This is to ensure minimum efficiency, because a purely random labeling would create many unconnected nodes, for which any unused network node would be a suitable candidate. This would exponentially increase the execution time needed for computing a census, as was explained in Section 4.4.2.
- **Opposite:** a labeling algorithm expressing the opposite of **GTCanon**, that chooses the nodes with less connections for the lower levels of the g-trie. As in the **Random** labeling case, we ensure connectivity in order to guarantee minimum efficiency.

Table 6.9 shows the results obtained for two networks, one directed and one undirected, for a representative set of sizes k so that the computation is not instantaneous but at the same time small enough so that even the slowest labeling method takes only a few minutes. We computed the census by using a g-trie with all possible k -subgraphs.

network		circuit			metabolic	
k		7	8	9	4	5
Execution time (s)	GTCanon	0.037	0.202	1.230	0.178	10.611
	Larger	0.075	0.554	4.202	0.224	19.108
	Random	0.251	2.846	45.814	0.292	40.319
	Opposite	0.628	8.723	145.116	0.351	55.927
Compression Ratio	GTCanon	82.21%	86.33%	87.12%	71.98%	79.07%
	Larger	83.44%	86.33%	88.32%	72.86%	79.48%
	Random	67.43%	68.01%	69.00%	68.84%	74.15%
	Opposite	78.80%	81.87%	84.28%	72.61%	79.18%

Table 6.9 – Effect of canonical labelings on k -census computation.

CHAPTER 6. EXPERIMENTAL EVALUATION

We can observe that our chosen strategy has the best behaviour (smaller execution times) for every pair of network and subgraph size, both in undirected and directed cases.

The lexicographically largest adjacency matrix (**larger**), which is more costly to compute, produces higher compression in the g-tries, but this does not have a proportional impact in lowering the execution time. Indeed, a greater compression ratio is desirable, but in itself it does not guarantee better performance and the **GTCanon** labeling takes advantage of the fact that we know the census algorithm and therefore can optimize it for discovering, as soon as possible, that a set of nodes will not match to a subgraph.

The **random** labeling shows that the compression ratio and census efficiency cannot be assumed to happen, and thus efficiency must be obtained by choosing an appropriate labeling. Finally, **opposite**, the reverse of our chosen strategy, is even worst than the random case, further substantiating the claim that our labeling choice has a positive effect on the efficiency of the census.

6.2.2.2 Effect of Symmetry Breaking Conditions

If we did not use the symmetry breaking conditions detailed in Section 4.5.2, all automorphisms of each subgraph isomorphic class would be found and the census would be substantially slower. Our filtering of symmetry conditions, as previously explained, reduces the memory needed for storing the the g-trie, and improves the execution times.

Table 6.10 shows execution times for a k -census on the same two networks used in the previous section, with a g-trie with all k -subgraphs, varying the symmetry breaking conditions used:

- **Normal:** our standard use of symmetry breaking conditions.
- **No filtering:** symmetry breaking conditions are used, but we do not minimize their number by following the 4 filtering steps described on section 4.5.2.3.
- **No conditions:** No symmetry breaking conditions are used at all.

As expected, not using any conditions slows the algorithm. Not filtering also slows down, albeit by a smaller margin. However, for bigger g-tries, the amount of unfiltered symmetry conditions starts to be so large that the algorithm becomes even slower than

6.2. SEQUENTIAL ALGORITHMS

network		circuit			metabolic	
k		7	8	9	4	5
Execution time (s)	Normal	0.037	0.202	1.230	0.178	10.611
	No filtering	0.062	0.590	38.562	0.238	15.638
	No conditions	0.205	1.592	13.155	0.460	56.414

Table 6.10 – Effect of symmetry breaking conditions on k -census computation.

the g-trie without any conditions (see $k = 9$ for **circuit**). This showcases the need for reducing the number of conditions and the validity of our filtering process.

6.2.2.3 Asymptotic Behavior

We will now have a look at the empirical asymptotic behavior of the g-trie census algorithm as the size of both the networks and subgraphs grows.

For testing network growth, we will use the synthetic social benchmark network described in Section 6.1.2, that allows us to slowly grow network size n while preserving the same topological characteristics. As before, we will be computing k -census with g-tries containing all possible subgraphs. The network is customized with the following parameters: average degree 20; minimum community size 20, maximum community size 50, mixing parameter 0.1 (for more information on these see [LFR08a] and the networks generation source code **Readme** file³).

Figure 6.1 shows the execution time for computing a full k -subgraph census by using a g-trie with all possible k -subgraphs. We vary the size from 500 to 10,000 nodes, with increments of 500 nodes and we use both undirected and directed versions of the network. We opted for choosing k , the subgraph size, so that the execution times are within a single minute. For better legibility, both graphs use the same scale, but one should note that in the case of the undirected network we are computing subgraphs of size 5, and in the directed network subgraphs of size 4.

We can see that the execution time appears to have a linear relation to the network size, for this particular network. Remember that g-tries (and for that matter all other current major motif detection algorithms) must explicitly pass in every subgraph occurrence in order to have an exhaustive perfect count. Taking this into account, a more telling statistic is to check the subgraph discovery ratio, that is, the number of

³The source code is available at <http://sites.google.com/site/andrealancichinetti/files>

CHAPTER 6. EXPERIMENTAL EVALUATION

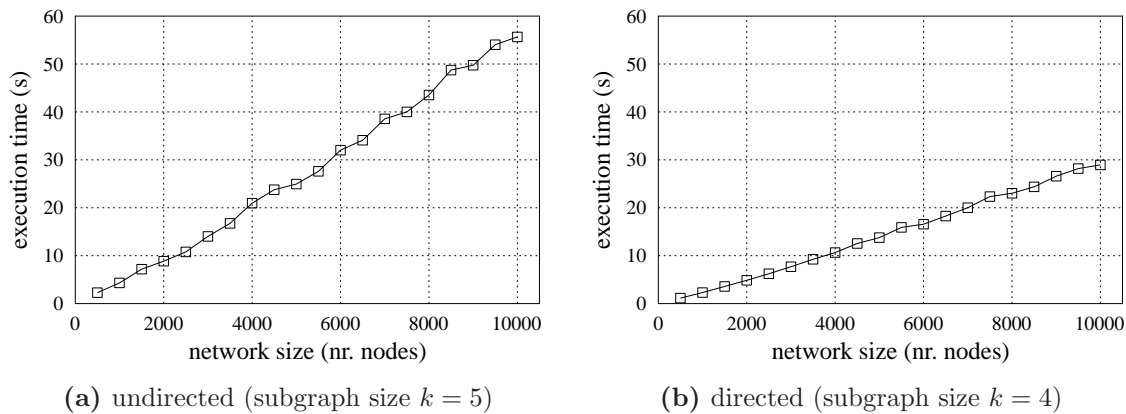


Figure 6.1 – Execution time for a census on a social network as the number of nodes increases.

occurrences found per second. A constant value would be the desirable situation, since the algorithm (such as the others) is by now limited by design to those occurrences. Figure 6.2 illustrates the subgraph discovery ratio for the same experiment.

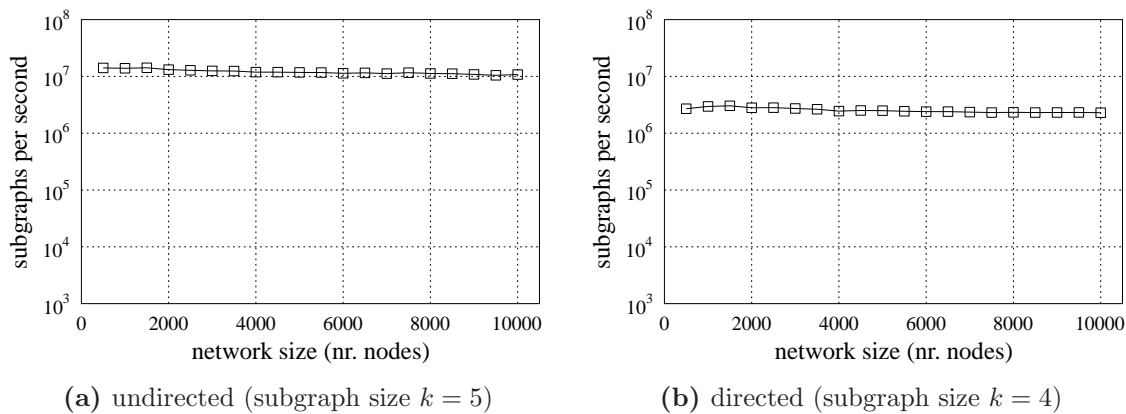


Figure 6.2 – Subgraph discovery ratio for a social network as the number of nodes increases.

We can see that our algorithm is able to maintain a steady flow on the number of subgraphs found per second, without losing much performance as the network size grows. Note however that the ratio is different in the two cases, since it is quite different to look for directed or undirected subgraphs, and their size is different.

In general, two different networks will induce different discovery ratios, as this depends heavily on the topology of the network itself. For example, a dense network would be more prone to a bigger ratio, with many subgraphs packed together. However, at the same time, it would have more subgraph occurrences.

6.2. SEQUENTIAL ALGORITHMS

Table 6.11 and Table 6.12 give a more practical view on this effect, by showing the subgraph ratio for different networks on subgraphs of the same size, when again computing a census with a complete g-trie. Note the differences, but also note the magnitude of the ratio, that stays between 10^6 and 10^7 . This does not mean however that, for every network, g-tries subgraph discovery ratio will stay within this margin.

Network	Execution Time	Subgraph Occurrences	Subgraphs/sec
dolphins	3.027	12,495,833	4,128,240
circuit	1.230	13,512,688	10,985,192
coauthors	448.071	886,423,840	1,978,312
power	18.444	183,453,978	9,946,388

Table 6.11 – Subgraph discovery ratio for 9-census on undirected networks.

Network	Execution Time	Subgraph Occurrences	Subgraphs/sec
neural	3.778	43,256,069	11,448,911
metabolic	10.611	195,573,511	18,431,064
links	971.269	7,347,672,714	7,562,022
odlis	713.630	8,655,784,561	12,129,235

Table 6.12 – Subgraph discovery ratio for 5-census on directed networks.

In what concerns the execution time behaviour when we increase the size of the motifs for the same network, what generally happens is that we have a subgraph discovery ratio that very slowly starts degrading as the size increases. Figure 6.3 shows this behaviour and how this ratio changes as we increment the subgraph size for two networks, one undirected, and one directed, as we compute a k -census using a complete g-trie.

6.2.3 G-Tries Comparison with Other Algorithms

In this section, we provide a thorough comparison of g-tries, with other competing algorithms, namely ESU, Grochow and Kavosh, as explained in Section 6.1.3.

For the first set of tests we will fully enumerate all k -subgraphs present in the original network, doing the equivalent to the first step of any motif discovery process. In the case of network-centric algorithms (ESU and Kavosh), this is simply running them.

CHAPTER 6. EXPERIMENTAL EVALUATION

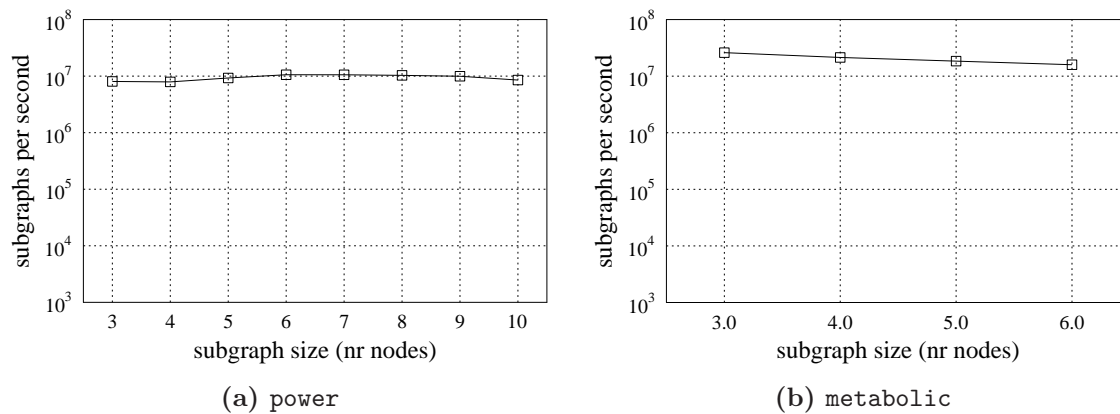


Figure 6.3 – Subgraph discovery ratio for `power` and `metabolic` as subgraph size increases.

In the case of the subgraph-centric algorithms (`Grochow`), we queried all possible k -subgraphs, in the same way we used g-tries with all those subgraphs inserted.

In order to give a broad overview of how g-tries consistently outperform every other major motif algorithm we used for comparison the entire set of 12 real complex networks. For each one of them we did a k -census, increasing k one by one starting with 3. In order to produce readable tables, we stopped when the slowest algorithm would take more than 5 hours to run and took note of the execution times and of the relative speedup of g-tries versus the other algorithms.

Tables 6.13 and 6.14 present the results obtained for the undirected and directed networks, respectively. All methods are identified by the first three letters of their name (`GTR` for g-tries, `ESU`, `KAV` for Kavosh and `GRO` for `Grochow`). Every experiment was executed at least three times and average execution times were recorded.

The major fact to notice is that there is not a single case where g-tries performs worst than other method, which showcases the efficiency of our algorithm. G-Tries clearly outperform the other algorithms in all the complex networks used.

6.2. SEQUENTIAL ALGORITHMS

Network	k	k -census execution time (s)				Speedup of g-tries vs		
		GTR	ESU	KAV	GRO	ESU	KAV	GRO
dolphins	3	< 0.001	< 0.001	< 0.001	< 0.001	14.9x	14.3x	9.8x
	4	< 0.001	0.006	0.006	0.003	15.5x	15.6x	8.3x
	5	0.002	0.036	0.036	0.032	14.7x	14.5x	12.8x
	6	0.014	0.251	0.241	0.324	17.7x	17.0x	22.8x
	7	0.085	1.499	1.405	3.727	17.7x	16.6x	43.9x
	8	0.483	9.182	8.468	55.093	19.0x	17.5x	114.0x
	9	3.027	52.431	46.033	1323.984	17.3x	15.2x	437.4x
circuit	3	< 0.001	0.001	0.001	0.002	12.6x	12.4x	18.8x
	4	< 0.001	0.007	0.007	0.006	18.0x	18.1x	17.1x
	5	0.002	0.034	0.034	0.043	21.9x	21.8x	27.7x
	6	0.007	0.221	0.218	0.317	32.4x	32.1x	46.5x
	7	0.037	1.365	1.311	2.778	36.7x	35.3x	74.8x
	8	0.202	9.267	8.670	32.630	45.9x	42.9x	161.5x
	9	1.230	59.804	53.152	632.213	48.6x	43.2x	514.0x
coauthors	3	< 0.001	0.010	0.010	0.030	14.1x	14.4x	42.0x
	4	0.006	0.077	0.077	0.172	12.0x	11.9x	26.7x
	5	0.053	0.633	0.617	1.635	12.0x	11.7x	31.0x
	6	0.442	5.711	5.568	18.465	12.9x	12.6x	41.8x
	7	4.088	50.748	49.579	284.614	12.4x	12.1x	69.6x
	8	40.560	481.432	464.540	6669.196	11.9x	11.5x	164.4x
ppi	3	0.004	0.086	0.092	0.092	21.8x	23.4x	23.4x
	4	0.068	3.106	3.053	1.450	45.9x	45.1x	21.4x
	5	1.507	84.959	84.852	39.182	56.4x	56.3x	26.0x
	6	40.275	2922.555	2934.426	1092.221	72.6x	72.9x	27.1x
power	3	0.002	0.017	0.018	0.215	7.9x	8.3x	98.3x
	4	0.008	0.101	0.102	0.845	12.6x	12.7x	105.3x
	5	0.029	0.481	0.486	5.164	16.6x	16.8x	178.1x
	6	0.119	2.947	2.913	35.729	24.7x	24.4x	299.7x
	7	0.600	17.891	17.285	313.052	29.8x	28.8x	521.5x
	8	3.247	120.695	112.065	3840.250	37.2x	34.5x	1182.7x
internet	3	0.423	11.571	12.206	6.865	27.3x	28.8x	16.2x
	4	204.442	11044.788	10827.744	3390.107	54.0x	53.0x	16.6x

Table 6.13 – Comparison of g-tries with other algorithms when doing a full k -census on original undirected networks.

CHAPTER 6. EXPERIMENTAL EVALUATION

Network	k	k -census execution time (s)				Speedup of g-tries vs		
		GTR	ESU	KAV	GRO	ESU	KAV	GRO
neural	3	0.004	0.043	0.047	0.031	12.3x	13.2x	8.9x
	4	0.112	1.989	1.981	1.676	17.7x	17.6x	14.9x
	5	3.778	107.646	111.350	124.329	28.5x	29.5x	32.9x
	6	128.364	5382.221	5837.072	17214.101	41.9x	45.5x	134.1x
metabolic	3	0.003	0.062	0.067	0.035	22.3x	24.0x	12.6x
	4	0.178	5.037	5.215	3.214	28.3x	29.3x	18.1x
	5	10.611	428.613	424.673	238.696	40.4x	40.0x	22.5x
links	3	0.080	1.086	1.110	0.719	13.6x	13.9x	9.0x
	4	9.427	124.496	124.852	110.434	13.2x	13.2x	11.7x
	5	971.269	16984.341	16895.159	16766.414	17.5x	17.4x	17.3x
odlis	3	0.062	1.027	1.105	1.015	16.5x	17.8x	16.3x
	4	5.394	237.232	237.193	145.371	44.0x	44.0x	26.9x
company	3	0.016	0.296	0.308	0.656	18.7x	19.5x	41.5x
	4	1.751	67.476	70.457	35.434	38.5x	40.2x	20.2x
	5	241.378	13460.332	13459.479	2127.840	55.8x	55.8x	8.8x
foldoc	3	0.352	2.720	2.799	20.343	7.7x	8.0x	57.8x
	4	18.338	379.087	393.359	1124.439	20.7x	21.5x	61.3x

Table 6.14 – Comparison of g-tries with other algorithms when doing a full k -census on original directed networks.

6.2. SEQUENTIAL ALGORITHMS

Specific speedups obtained can vary from method to method and from network to network, but what is consistent is that using g-tries take less time to compute. Figure 6.4 gives a more graphical view of the obtained speedups. Note that the speedup scale is logarithmic and that almost all speedups seem to have the tendency to grow as the subgraph size increases, meaning that g-tries appear to scale a little better than the other algorithms.

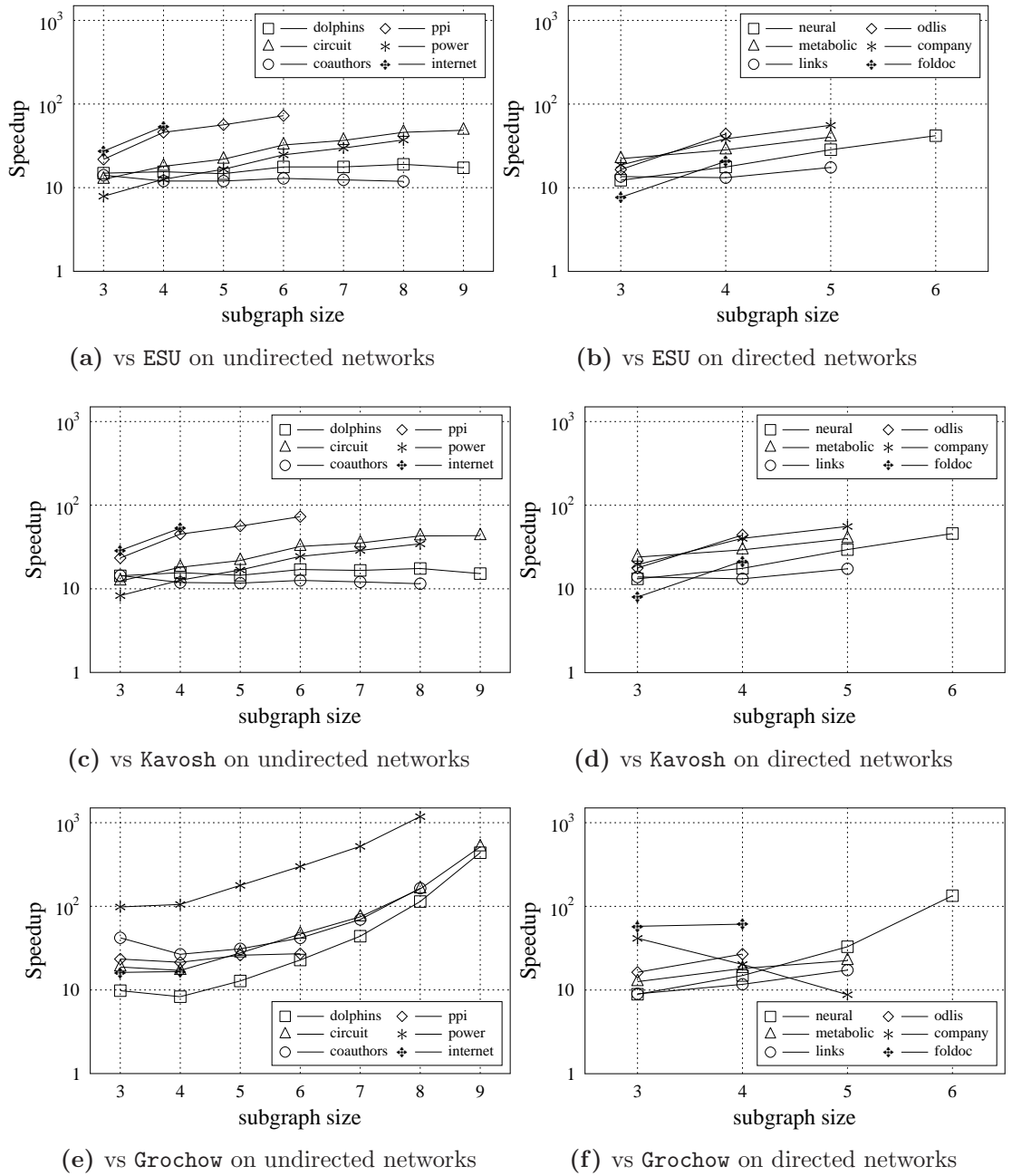


Figure 6.4 – G-Tries speedup over other algorithms on original network census.

CHAPTER 6. EXPERIMENTAL EVALUATION

For all the networks, the speedup is at least very close to an order of magnitude faster. If we consider the last computed subgraph size, there is only one network (`company`) where our algorithm is less than 10x faster than all the other algorithms, and even so it is 8.8x faster than the closest competitor. In the other cases it is much faster, like in the case of `circuit` or `neural`, where it is more than 40x faster than all other approaches.

Regarding this experiment, one can observe that the behaviour of the network-centric methods (ESU and Kavosh) is very similar to each other, with only a marginal difference. Grochow, on the other hand, performs differently, having difficulties when the number of subgraph classes increases. This is also a problem with our approach, since increasing k will likely make the g-trie too big to fit in memory. However, remember that this last experiment pertained to the computation of the full k -census in the original network. As shown in Section 4.7, we can opt for an hybrid approach and use a network-centric method on the original network, and only after that populate the g-trie with the found classes of subgraphs, which typically will be a much smaller percentage of the whole set of possible subgraphs.

Moreover, in the current motif discovery flow, the real bottleneck comes after when the census must be computed for the ensemble of similar random networks. Since the size of this set is normally large (one hundred is a typical minimum use case) and since the random networks have the same number of nodes and edges of the original network, the time spent on this step of the computation takes precedence over everything else. It is therefore important to test the algorithms on randomized networks generated in this way.

In order to do this test, we applied a random Markov-Chain process (see Section 3.4) like it was done in [MSOI⁺02], with 3 swaps per edge and computed the census of all subgraphs that appeared on the original network. Note that the motif definition (see Section 2.2.1) allows one to specify an uniqueness threshold, specifying the minimum frequency that a subgraph must have in order to be considered a motif. In a real use case we can discard some subgraphs that are below the threshold and not count them on the randomized networks. However, for the sake of a more complete experiment, here we consider that this threshold is zero, meaning that every subgraph class that appears at least once in the original network is considered.

We use the exact the same randomized networks in all algorithms, obtained by using the same pseudo-random number generation seed. For network-centric methods, the only option is to do an exhaustive census. For subgraph-centric methods capable of

6.2. SEQUENTIAL ALGORITHMS

querying a single subgraph, namely `Grochow`, we only queried the pertinent subgraphs. For the g-tries, we created a g-trie precisely with the subgraphs for which we want to know the frequency in the random networks. We produced the same type of data as we had for the original network, but this time we ran the algorithms on 25 different random networks.

Considering that in a real use case, normally a minimum number of 100 random networks is used, we stopped when the average time per random network meant that computing the census in a set of 100 randomized networks with the slowest algorithm for that case would take more than 10 hours. In other word, the average execution time per randomized network must be smaller than 360s. In the cases where this meant that there would only be results for subgraphs of 3 nodes, we also report results on subgraphs of 4 nodes.

Tables 6.15 and 6.16 show the obtained average execution time for each similar randomized network, and the average speedup of g-tries against the other approaches. The standard deviation for the time spent in each randomized network, not shown in the table for the sake of legibility, is always smaller than 25% of the average, meaning that this average is a good indicator and that the time needed per randomized network does not suffer large variations.

First, observe that the computation time for a single randomized network tends to be larger than the time to compute the census in the original network. This is caused by the fact that randomization creates a network with more subgraph occurrences to be found, due to a more chaotic structure.

Nevertheless, g-tries again consistently outperform all other algorithms, being substantially faster for every network and for every subgraph size. Moreover, the speedups versus the network centric methods are generally even better, partially due to the fact that we are now using smaller g-tries, populated with only the relevant subgraphs, instead of all possible subgraphs of that size.

Observing the speedup against all other algorithms on the last subgraph sizes computed in each network, we can see that it always larger than 10x for all networks, and even larger than 20x for 8 out of the 12 networks.

Also note that, again, the speedups show a tendency to increase as the subgraph size is incremented, as illustrated in Figure 6.5. This points to even further gains for larger sizes.

CHAPTER 6. EXPERIMENTAL EVALUATION

Network	k	Avg. execution time per network (s)				Avg G-Tries Speedup		
		GTR	ESU	KAV	GRO	ESU	KAV	GRO
dolphins	3	< 0.001	< 0.001	< 0.001	< 0.001	17.7x	18.3x	10.7x
	4	< 0.001	0.008	0.008	0.003	24.8x	24.5x	9.4x
	5	0.003	0.068	0.065	0.036	25.7x	24.6x	13.6x
	6	0.021	0.576	0.572	0.393	27.8x	27.6x	19.0x
	7	0.155	4.403	3.988	4.211	28.4x	25.8x	27.2x
	8	1.121	32.399	30.140	44.293	28.9x	26.9x	39.5x
circuit	3	< 0.001	0.001	0.001	0.002	11.6x	12.0x	16.9x
	4	< 0.001	0.007	0.007	0.006	18.8x	19.3x	17.0x
	5	0.002	0.036	0.036	0.037	23.1x	23.2x	23.8x
	6	0.008	0.239	0.236	0.231	31.7x	31.3x	30.6x
	7	0.040	1.514	1.509	1.412	38.0x	37.9x	35.5x
	8	0.228	10.799	10.656	8.641	47.4x	46.8x	37.9x
	9	1.403	74.652	72.969	55.307	53.2x	52.0x	39.4x
coauthors	3	< 0.001	0.015	0.016	0.029	19.0x	20.1x	37.1x
	4	0.006	0.231	0.229	0.198	38.3x	38.0x	32.9x
	5	0.060	2.873	2.900	2.281	48.1x	48.5x	38.2x
	6	0.747	48.128	49.532	29.687	64.4x	66.3x	39.7x
ppi	3	0.004	0.097	0.096	0.089	25.9x	25.6x	23.6x
	4	0.072	3.657	3.659	1.516	50.6x	50.7x	21.0x
	5	1.862	115.058	115.597	47.668	61.8x	62.1x	25.6x
power	3	0.002	0.020	0.020	0.198	8.9x	8.8x	85.9x
	4	0.014	0.132	0.134	0.848	9.7x	9.9x	62.4x
	5	0.050	0.758	0.746	5.433	15.2x	15.0x	109.2x
	6	0.211	5.528	5.499	39.124	26.2x	26.1x	185.6x
	7	1.064	40.585	40.399	304.141	38.2x	38.0x	285.9x
internet	3	0.491	11.608	12.378	6.472	23.7x	25.2x	13.2x
	4	245.065	11495.764	11148.303	3614.261	46.9x	45.5x	14.7x

Table 6.15 – Comparison of g-tries with other algorithms when computing a k -census in the similar undirected randomized networks.

6.2. SEQUENTIAL ALGORITHMS

Network	k	k -census execution time (s)				Speedup of g-tries vs		
		GTR	ESU	KAV	GRO	ESU	KAV	GRO
neural	3	0.004	0.049	0.055	0.033	13.5x	15.1x	9.2x
	4	0.143	2.496	2.431	2.330	17.4x	17.0x	16.3x
	5	5.506	139.271	140.120	158.658	25.3x	25.5x	28.8x
metabolic	3	0.003	0.059	0.061	0.019	22.7x	23.6x	7.3x
	4	0.109	3.955	4.086	1.271	36.3x	37.5x	11.7x
	5	4.418	308.769	304.222	68.257	69.9x	68.9x	15.4x
links	3	0.089	1.369	1.469	0.791	15.5x	16.6x	8.9x
	4	13.031	193.734	197.661	171.364	14.9x	15.2x	13.2x
odlis	3	0.075	1.173	1.222	1.165	15.6x	16.2x	15.4x
	4	8.848	259.565	262.449	199.647	29.3x	29.7x	22.6x
company	3	0.025	0.308	0.338	0.595	12.1x	13.3x	23.4x
	4	1.396	68.241	69.906	35.375	48.9x	50.1x	25.3x
foldoc	3	0.608	4.200	4.252	23.804	6.9x	7.0x	39.1x
	4	32.907	520.484	526.030	1661.945	15.8x	16.0x	50.5x

Table 6.16 – Comparison of g-tries with other algorithms when computing a k -census in the similar directed randomized networks.

CHAPTER 6. EXPERIMENTAL EVALUATION

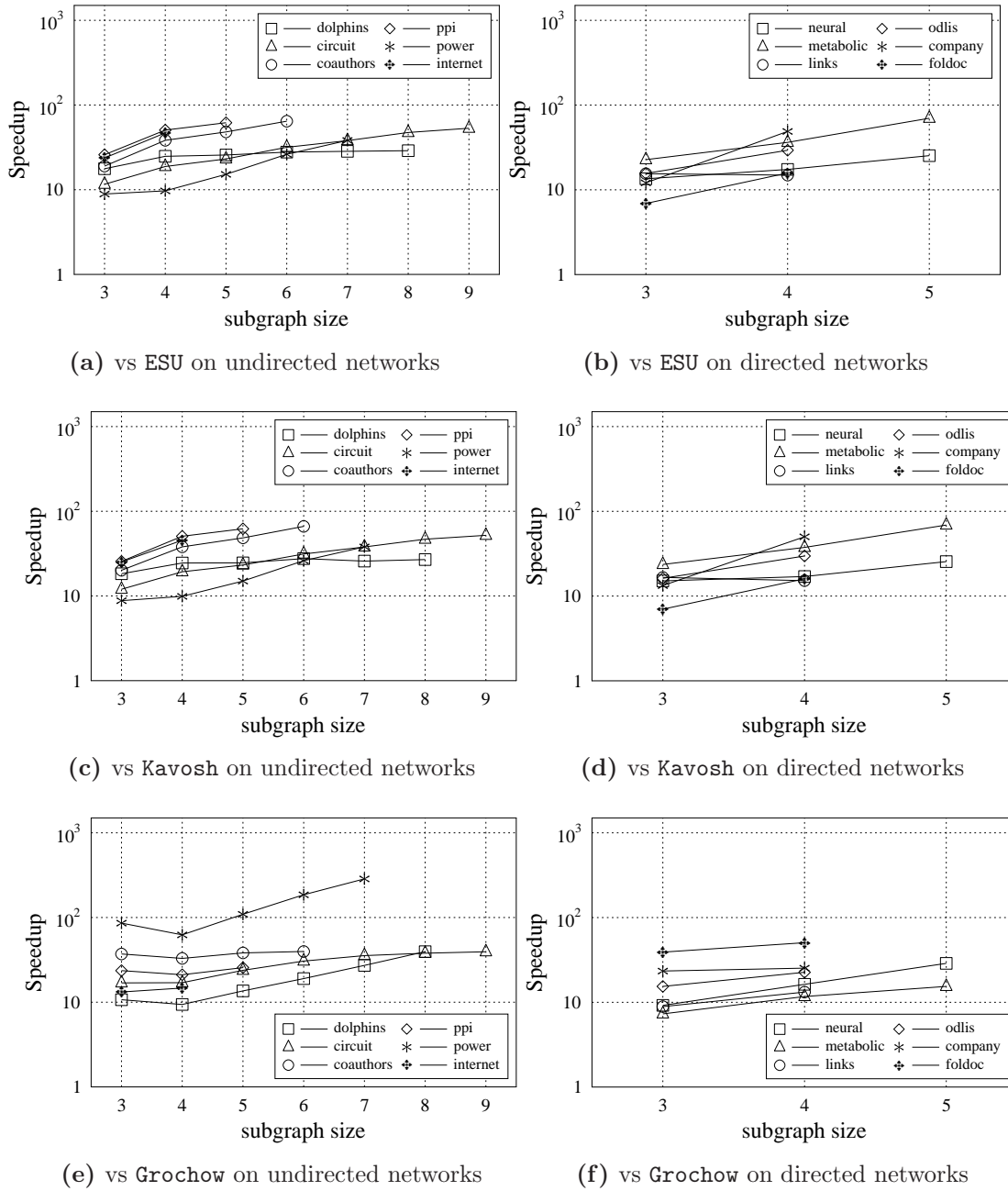


Figure 6.5 – G-Tries speedup over other algorithms on the randomized set of networks.

6.2. SEQUENTIAL ALGORITHMS

Combining the observations on the original network with the observations on the randomized networks, we can estimate the time needed for discovering motifs, that will be essentially dominated by the time spent in the large set of similar networks. The result is a new much faster algorithm for motif detection.

Table 6.17 shows the average speedup obtained with g-tries when compared against the three other competing algorithms. We used as a basis the results of the last two tables (execution time for a census in the randomized networks) and computed the average speedup per network, giving the same weight to each network in the calculation of the global average. We show the g-tries speedup against each method category (network-centric: **ESU** and **Kavosh**; and subgraph-centric: **Grochow**) and subgraph type (undirected and directed). We also show this average speedup when considering all the subgraph sizes computed and also when only considering the last subgraph size computed for each network, which corresponds to larger execution times.

Subgraph Size	Subgraph Type	Average Speedup vs method type		
		All methods	Network-centric	Subgraph-centric
All	All	30.1x	28.6x	34.4x
Measured Sizes	Undirected	37.2x	33.5x	47.3x
	Directed	23.0x	23.8x	21.5x
Last	All	44.3x	40.1x	49.9x
Measured Size	Undirected	57.2x	46.0x	73.8x
	Directed	31.4x	34.1x	26.0x

Table 6.17 – Average speedup of g-tries against competing algorithms.

Globally, g-tries are 30.1x faster, on average, than its competing algorithms. The speedup is more pronounced on undirected subgraphs (37.2x) than on directed ones (23.0). The speedup is also larger, on average, against subgraph-centric methods (34.4x) than against the network-centric methods (28.6x). This last observation is inverted in the case of directed subgraphs, where the speedup of network-centric methods (23.8x) is larger than the subgraph-centric one (21.5x).

If we just take into account the last subgraph size computed for each network (which factors bigger sizes, more subgraph occurrences, and larger execution times), the results are similar but the speedup obtained is even larger, with a global average speedup of 44.3x against the competing algorithms.

CHAPTER 6. EXPERIMENTAL EVALUATION

6.2.4 G-Tries Sampling

We will now analyze the sampling option of g-tries and overview the balance between accuracy and execution speed. For the purposes of this section we will limit the choice of probability parameters to three levels of quality. In order to sample a fraction f of all k -subgraph occurrences, we will use one of the following options, with the given name being a hint to the predicted accuracy:

- **very high**: $\{P_0 = 1, \dots, P_{k-2} = 1, P_{k-1} = f\}$
- **high**: $\{P_0 = 1, \dots, P_{k-3} = 1, P_{k-2} = \sqrt{f}, P_{k-1} = \sqrt{f}\}$
- **medium**: $\{P_0 = 1, \dots, P_{k-4} = 1, P_{k-3} = \sqrt[3]{f}, P_{k-2} = \sqrt[3]{f}, P_{k-1} = \sqrt[3]{f}\}$
- **low**: $\{P_0 = 1, P_1 = \sqrt[k]{f}, P_2 = \sqrt[k]{f}, \dots, P_{k-1} = \sqrt[k]{f}\}$

Basically, **very high** means that we will traverse the entire tree up to the last level and only then will we make a probabilistic choice. This reduces the variability on the results but will potentially take more time. **high** starts the probabilistic choices one level before and **medium** another level more, trading some potential accuracy for better speed. **low** represents the other extreme, by distributing the non-determinism to all levels except the first and lower level of the tree.

For the rest of this section we will use the term **RAND-GTRIE** to designate sampling with g-tries. Our first test is to analyze the speed at which we are able to generate samples. For that we computed the sampling ratio measuring how many subgraphs we are finding per second. We compare the performance with **RAND-ESU**, the previous most efficient network centric method that also allows sampling in a way similar to ours (by specifying a similar set of probabilities for each level). **Kashtan**, the other major comparable alternative for sampling, was shown to be much slower than **RAND-ESU** and it does not scale well [Wer06].

We use one undirected and one directed network, and we compute the sampling ratio both for a complete enumeration (all $P_i = 1$) and with only 10% of the k -subgraphs obtained by sampling with the **high** quality level. We used motif sizes from 4 to 6, and the results are shown in Figure 6.6.

The main aspect to note is that **RAND-GTRIE** is always faster. This was also the case for all other networks and quality levels tested, with the speedups of the sequential exact algorithm propagating to the sampling version. Also, as before, **RAND-GTRIE**

6.2. SEQUENTIAL ALGORITHMS

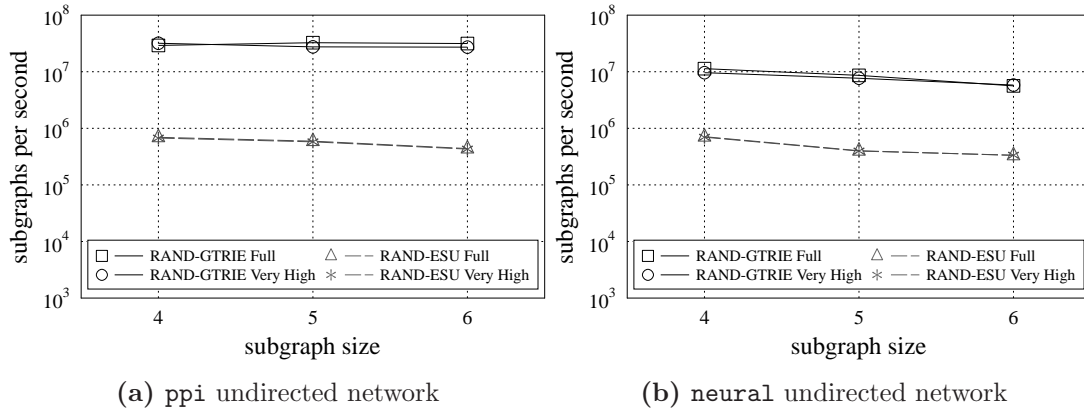


Figure 6.6 – Subgraph discovery ratio with RAND-GTRIE and RAND-ESU.

is able to sustain the subgraph discovery ratio, scaling well when the subgraph size increases.

In order to test the accuracy of our algorithm, we applied all levels of sampling quality, while increasing the fraction of subgraphs being sampled, taking note of the percentage of subgraphs correctly identified. We considered an estimate to be accurate when it was within a 20% error margin of the correct perfect value. We took 100 samples for each fraction and level and only considered the estimate correct when at least 75 of those samples were accurate. For accuracy purposes, we do not consider subgraph classes which are sampled less than 10 times, reflecting what would probably happen in a real case, where such a small number of samples would not give any certainties to the user.

The results for the same two networks as before, with 5 as motif size, are shown on Figure 6.7. As expected, higher probabilities in lower depths correspond to better sampling quality (less variance). Note the completely different shape of the curves in different networks, which suggests that the accuracy is influenced by the topology of the network.

If we measure the execution time for the exact same tests, we can see that the opposite happens, with better quality sampling taking more execution time as detailed in Figure 6.8. All quality levels have an execution growth proportional to the percentage of samples, but higher quality levels have a minimum time bigger than lower quality minimum time. For example, on the `ppi` network, sampling just 0.1% of the subgraphs in `very high` level of quality already takes around 18.01% of the time it takes to do a full enumeration. This is because we are traversing the entire tree up to depth $k - 1$.

CHAPTER 6. EXPERIMENTAL EVALUATION

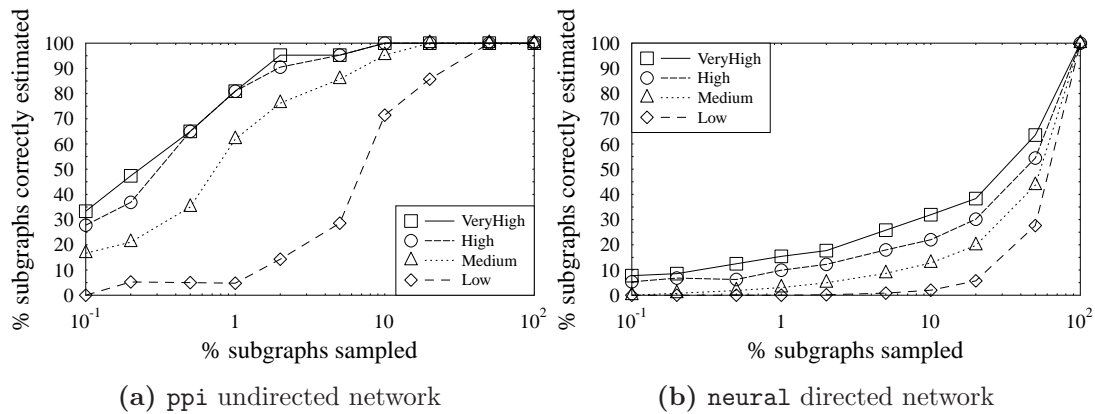


Figure 6.7 – Accuracy of g-tries sampling of 5-subgraphs.

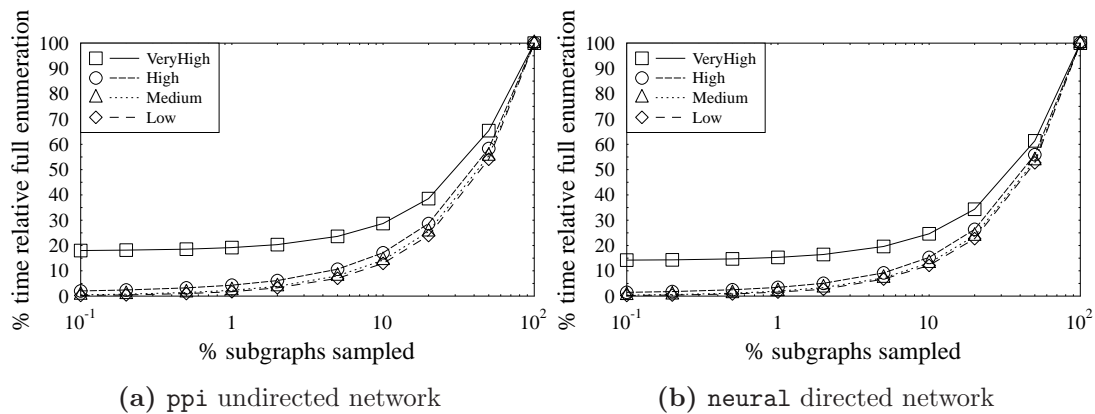


Figure 6.8 – Time spent in g-tries sampling of 5-subgraphs.

Judging by our empirical tests, the ideal choice of parameters for sampling with g-tries can vary with the network. However, the **high** level results seem to offer a good balance between execution time and sampling quality.

If we take a closer look to what g-tries sampling is computing, we can see that the more rare subgraphs are the ones with less estimation quality. This is because a smaller number of occurrences will obviously imply more variance in the estimated values (a “miss” has more weight). Almost all of the subgraph classes not correctly estimated appear less than 100 times in the sample. On the other hand, with a low percentage of samples, the few that are estimated correctly correspond to subgraph classes that were sampled at least 1000 times.

Note that the registered accuracy does not have a direct proportional impact on the possible motifs found. It is true that we want the algorithm to be as accurate as

6.3. PARALLEL ALGORITHMS

possible, but a subgraph not reported as accurate in the last figures can still be reported as a motif, if its z-score is high, meaning that it is still over-represented.

Precisely regarding motif detection, we experimented to discover all motifs of size 5 in the two networks, using 10% sampling with **high** quality level. For motif detection, we used $R = 100$ random networks, minimum frequency $U = 4$, minimum deviation $D = 0.1$ and probability $P = 0.01$.

We were able to find all of the motifs that a full enumeration would find in the **ppi** network, spending less than 20% of the time it would take to compute that full enumeration. In the **neural** network, more than 70% of the motifs were found, again spending less than 20% of time a full enumeration would take.

6.3 Parallel Algorithms

This section shows the results obtained with the developed parallel algorithms for motif discovery. We start by explaining how parameters were chosen and we follow with a more detailed analysis of the obtained speedup with all the proposed strategies.

6.3.1 Parameters Choice

In Chapter 5, for the parallel algorithms, we mentioned three parameters that act as thresholds. The **splitThreshold**, T_{split} , tells where to stop work splitting, and it is measured as the distance to a leaf node. The **messagesThreshold**, T_{check} , controls the frequency with which we poll for incoming messages, and it can be measured in terms of the number of work units computed or time spent. Finally, the **masterThreshold**, T_{master} , controls the frequency with which a worker sends unprocessed units to the master within the master-worker strategy, and it can be measured in terms of the number of work units computed or time spent.

For T_{split} , ideally one would choose the first distance k so that it takes more time to communicate that part of the search than to just compute it. However, the time to compute is related to the topology of the network and can vary significantly for each network.

A value of k for T_{split} means that a worker will prefer to compute locally all the associated “ k -subsubgraphs” that complete the partial match already made. A value

CHAPTER 6. EXPERIMENTAL EVALUATION

of 1 for k , means that the work completes almost instantaneously, as only one more node needs to be added. A value of 2 for k means that we are looking for pairs of connected nodes, and the work is also executed almost instantaneously. A value of 3 for k , however, may mean that in some extreme cases the computation will already be somehow significant (remember for example that sequentially computing all 3 – *subgraphs* of the bigger networks is already measured in seconds - see Tables 6.13, 6.14, 6.15 and 6.16). For this reason we opted for $T_{split} = 2$ and empirically verified that for the networks we use it allows good scalability, as will be seen below.

For T_{check} , we noted that using as a unit the time spent was not feasible. The threshold condition is tested very often, on every work procedure call, and the most efficient language primitive for time check would still degrade considerably the algorithm performance if used that often. We therefore opted for using the number of computed nodes, and since the rate of traversed nodes remains relatively stable for the same search, this means it will be roughly equivalent to using time.

The MPI implementation on our computing environment, as empirically verified, has a very efficient poll primitive (`MPI_Probe`) to check for new incoming messages. With this, we are able to choose relatively small values for T_{check} without degrading the performance of the search. Ultimately, we chose 10,000 for parallel ESU and 100,000 for parallel g-tries, which roughly amounts to an average of less than 0.1s of computation.

Finally, regarding T_{master} , after empirical verification, we chose 100,000 for parallel ESU and 1,000,000 for parallel g-tries, which roughly amounts to 1s of computation, which produced good results when compared to larger or smaller threshold values.

We are aware that these parameter values can be data and system dependent for optimal performance, and that ideally these should be dynamically computed by our own algorithm, and that is indeed in our future plans. But it remains that they add adaptability to the algorithm, instead of detracting it from being efficient on other cases.

6.3.2 Parallel Strategies for Pre-Processing and Aggregation

The pre-processing and aggregation phases, when in presence of a large data set, may be a source for inefficiency, and thus careful strategies need to be thought. Regarding pre-processing we have presented two alternative strategies `all_in_one` and `static_partition`, detailed in Section 5.3.4. The later is clearly a better strategy

6.3. PARALLEL ALGORITHMS

since each processor can immediately start computing, instead of having to ask for work using message passing.

This is better illustrated in Figures 6.9 and 6.10, which have a more detailed look at communication between processors during the work phase of the parallel execution. In these figures, white represents no communication (busy state) and black regions represent communication. We consider that a processor is communicating when it is trying to find work (by sending request messages) or, when it is providing work (by servicing work request messages). In both figures we are using 128 processors to compute the 6-census of the `neural` network and using a distributed queue strategy for load balancing.

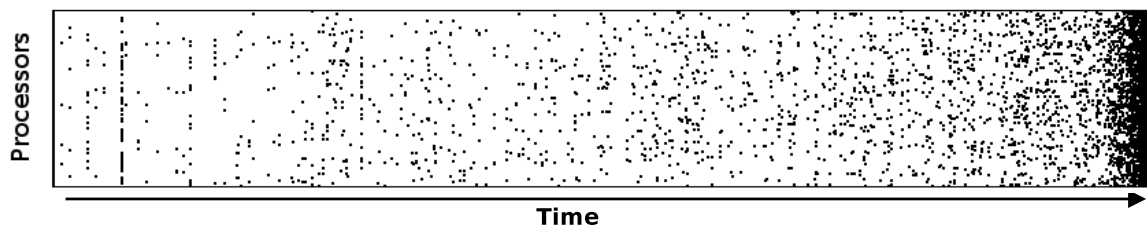


Figure 6.9 – Communication between 128 processors with `all_in_one` pre-processing phase.

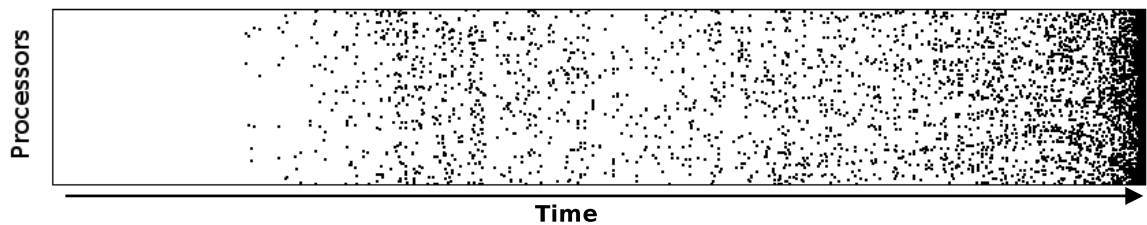


Figure 6.10 – Communication between 128 processors with `static_partition` pre-processing phase.

Note the intense communication in the beginning when using `all_in_one` in the pre-processing phase, with all processors trying to get some initial work. By contrast, in the `static_partition` alternative, all processors are already busy at the beginning, since we provide an initial work allocation. Sometime after, some processors start running out of work (exactly when depends on the topology of the network and the subgraphs of the g-trie). The communication gradually increases as the granularity of work units left to compute starts to decrease, with an obvious peak of communication at the end, where all processors are trying to find the remaining pieces of work and trying to establish that indeed there is nothing more to compute.

CHAPTER 6. EXPERIMENTAL EVALUATION

In the aggregation phase, we must communicate the frequency of each isomorphic class in both the original and randomized networks. We proposed three possible strategies, `naive`, `hierarchical` and `collective`, detailed in Section 5.3.6. Figure 6.11 summarizes the results obtained when using these strategies as we increase the number of subgraph frequencies being communicated. The measured time is the wall clock time required to aggregate all subgraph frequencies in a single processor, suitable for further computing the significance if we are discovering motifs. For this figure we used randomized frequencies, assuming that each frequency is stored on a `MPI_INT` 32-bit integer data type.

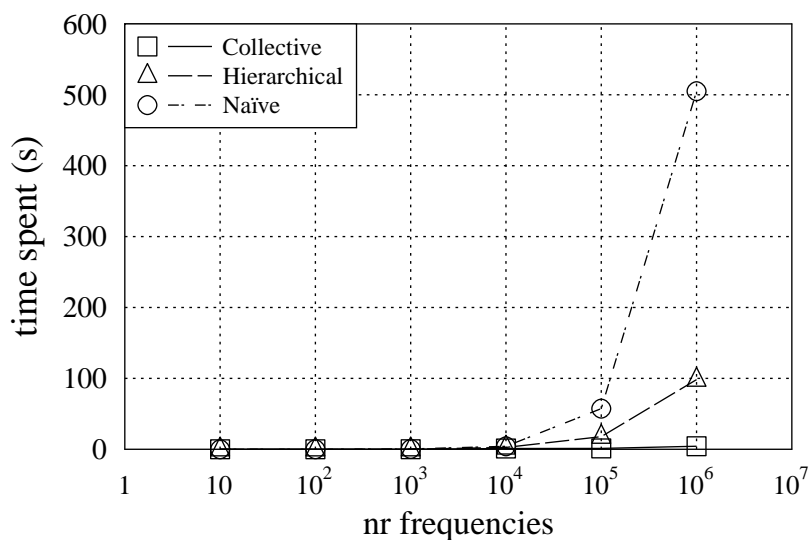


Figure 6.11 – Aggregation times for all strategies as the number of communicated frequencies increases.

The best option in terms of scalability, is clearly the `collective` strategy, meaning that the implementation of the primitive `MPI_Reduce` used is efficient and is this justifiable to be used for gathering all information.

6.3.3 Load Balancing and Scalability

With the best strategies for pre-processing (`static_partition`) and aggregation (`collective`) fixed we will now experiment with the three options available for load balancing.

The core of our test will be to apply these strategies to compute k -motifs for all of the 12 real networks. We fixed the number R of random graphs to 100, a quantity capable of already giving meaningful results in terms of subgraph significance (also

6.3. PARALLEL ALGORITHMS

used in many articles, such as [SK04]).

We want to show that our strategies are able to effectively parallelize the problem at hand, so we need to use cases where the sequential execution time is big enough to justify parallelization up to the 128 processors. For that purpose we measured the sequential execution time for computing all motifs of size k as we increase k , generating 100 similar random networks, and we stopped when that time was larger than one hour (in two cases we stopped a little before because the next k would mean computing for more than one day). We also calculated the average growth ratio, that is, for increasing values of k , how much did the execution time grew, the approximate total number of different occurrences of k -subgraphs in all the networks (original and randomized) and the number of different subgraph types (isomorphic classes) found in the original network. We used our hybrid approach, with ESU for discovering the subgraphs in the original network, and a g-trie only with those subgraphs to find all k -motifs. The results can be seen in Table 6.18.

Network	k	Execution time (s)	Average Growth	Approx. total nr subgraphs	Nr Iso-morphic classes
dolphins	10	5,728.442	6.75 \pm 0.4	$\approx 3 \times 10^{10}$	295,236
circuit	11	7,942.371	5.79 \pm 0.9	$\approx 6 \times 10^{10}$	13,462
neural	6	27,438.786	39.24 \pm 2.3	$\approx 2 \times 10^{11}$	286,376
metabolic	6	48,736.553	55.26 \pm 2.9	$\approx 3 \times 10^{11}$	5,633
links	4	1,424.454	138.94 \pm 0.0	$\approx 1 \times 10^{10}$	199
coauthors	8	15,260.855	10.81 \pm 4.2	$\approx 4 \times 10^{11}$	2,612
ppi	6	8,835.332	29.54 \pm 5.0	$\approx 2 \times 10^{11}$	112
odlis	4	1,101.509	116.24 \pm 0.0	$\approx 2 \times 10^{10}$	199
power	9	5,026.64	5.19 \pm 1.2	$\approx 7 \times 10^{10}$	31,543
company	5	42,284.77	131.37 \pm 105.5	$\approx 1 \times 10^{10}$	310
foldoc	4	3,676.80	57.45 \pm 0.00	$\approx 3 \times 10^{10}$	198
internet	4	34,840.27	580.34 \pm 0.00	$\approx 2 \times 10^{11}$	6

Table 6.18 – First motif sizes k for which a sequential program takes more than 1 hour.

The average growth ratios have a relatively low standard deviation. This means that we can make rough estimates on how much more time we would need to compute greater motif sizes. Note also that growth differs significantly from network to network,

CHAPTER 6. EXPERIMENTAL EVALUATION

and it is not directly proportional to the size of network. Typically, directed networks present a larger growth, since they feature more connectivity options and more possible subgraph types.

We can now show the global behaviour of our parallel algorithms using the three load balancing strategies. Table 6.19 details the absolute speedups we obtained, i.e., the relative time gain when compared to running the sequential version with only one processor. This time was measured inside the program itself to exclude the MPI initialization process, which takes at most a couple of seconds.

All the strategies have very good performances and scale up to 128 processors. The distributed strategies are better options since all processors can be used for the computation, with no CPU power wasted. As expected, the **distributed snapshot** is the overall better option, since it does not introduce the overhead of reading and storing work units (see Section 5.3.5.3).

Regarding this last option we can see that it obtains almost linear speedup for 128 processors (the numbers indicated in bold). Only in two cases the speedup is smaller than 100x (**links** and **odlis**), but this is caused by the relatively small value of the initial sequential time. For example, in the **odlis** case, 1,101.509s was the time spent by the sequential program. This means that a program running 128x times faster would have to run in less than 8.6 seconds. In fact, the 128 processor version only took less than 4 seconds more than that (12.212s), but with such low values this was not enough for reducing the speedup. For larger sequential execution times this effect does not happen, and hence the fact that all the bigger speedups obtained correspond to sequential versions that run for longer times.

Note also that we can effectively combine the power of g-tries with the power of parallelization. If we compare to the previously available sequential approaches, and by multiplying the speedup obtained by using g-tries by the parallel speedup, we obtain an algorithm that is in the order of several thousands of times faster than what was previously available. If we recall the average growth for computing k -motifs as we increase k (Table 6.18), we see that in the same amount of time we will be able to achieve bigger sizes, effectively unlocking new information for the practitioners.

Our parallel algorithms are very flexible and work well both for cases where the number of random networks is greater than the number of CPUs, and when this number is smaller. In order to further verify this claim, we used the full 128 processors on two of the networks (one undirected, other directed) using the distributed snapshot strategy, while varying the number of random networks. The most extreme lower case is when

6.3. PARALLEL ALGORITHMS

Network	k	Strategy	#CPUs: speedup		
			32	64	128
dolphins	10	Master-Worker	24.8	48.4	92.3
		Distributed Queue	27.7	53.7	102.0
		Distributed Snapshot	30.8	59.4	112.7
circuit	11	Master-Worker	25.1	50.5	98.4
		Distributed Queue	28.3	56.0	109.1
		Distributed Snapshot	31.3	61.7	121.2
neural	6	Master-Worker	25.4	51.2	102.3
		Distributed Queue	28.6	57.1	111.5
		Distributed Snapshot	31.4	62.5	122.8
metabolic	6	Master-Worker	25.5	51.6	104.2
		Distributed Queue	28.6	57.9	113.5
		Distributed Snapshot	31.5	62.9	126.0
links	4	Master-Worker	23.6	44.0	77.5
		Distributed Queue	26.8	49.8	88.0
		Distributed Snapshot	30.0	57.1	95.9
coauthors	8	Master-Worker	25.3	51.4	101.8
		Distributed Queue	28.4	57.3	111.8
		Distributed Snapshot	31.4	62.6	123.9
ppi	6	Master-Worker	25.1	51.0	99.0
		Distributed Queue	28.4	56.7	109.7
		Distributed Snapshot	31.4	62.0	122.1
odlis	4	Master-Worker	23.0	42.4	72.1
		Distributed Queue	26.2	48.3	82.4
		Distributed Snapshot	29.7	55.9	90.2
power	9	Master-Worker	24.9	49.1	94.2
		Distributed Queue	28.0	54.7	105.7
		Distributed Snapshot	31.1	61.0	118.8
company	5	Master-Worker	25.4	52.2	103.7
		Distributed Queue	28.9	57.0	114.1
		Distributed Snapshot	31.3	62.8	125.2
foldoc	4	Master-Worker	24.8	48.6	92.8
		Distributed Queue	28.2	54.7	102.6
		Distributed Snapshot	30.9	60.6	116.9
internet	4	Master-Worker	25.4	51.3	103.9
		Distributed Queue	28.6	57.2	114.5
		Distributed Snapshot	31.4	62.9	125.7

Table 6.19 – Parallel performance of motif discovery with g-tries.

CHAPTER 6. EXPERIMENTAL EVALUATION

we have zero random networks (in that case we are merely doing a subgraph census of the original network, with no statistical significance) and we extend the number of random networks up to 1,000. Table 6.20 details the results we obtained.

Network	k	Nr of random networks			
		0	10	100	1000
metabolic	6	124.0	124.3	124.2	123.1
internet	4	123.7	123.9	123.5	122.6

Table 6.20 – Parallel performance with 128 processors as the number of random networks change.

The algorithm scales well for all cases, being able to create a good load balance, regardless of the variation in the number of networks. Note that computing more random networks results in bigger aggregation times and in some cases this could be a factor in decreasing the speedup. In the networks presented in this table, the number of isomorphic classes is small, and therefore the increase on the number of random networks does not have a significant negative impact on the speedup obtained.

As described before, both the g-tries and the ESU algorithms allows one to trade accuracy for execution time. We experimented to sample 10% of all subgraphs with various quality levels and we obtained speedups comparable to running the normal exhaustive version. This means that the algorithm adapts well to the even more unpredictable search tree shapes and is able to maintain scalability. The speedup can however be relatively smaller than the one obtained with the complete enumeration since with sampling we take less time to enumerate but we still need roughly the same time for the aggregation phase, because the number of isomorphic classes and random networks does not change.

6.4 Summary

In this chapter we presented a thorough experimental evaluation of our proposed algorithms. First we have shown the common materials, namely the computational environment, the set of used networks and the competing algorithms used for comparison purposes. We then evaluated g-tries creation, showing high compressibility ratios and how g-tries can be reused by storing them in the file system. After, the census algorithm was examined, showing the power of the canonical labeling used and of the

6.4. SUMMARY

symmetry breaking conditions. We did an empirical asymptotically evaluation of our algorithm showing that it is able to maintain a stable subgraph discovery ratio as the network and motif sizes change. We then compared our algorithm to all previous existing algorithms, and the results show that it clearly outperforms all competing algorithms in all tested cases. We continued by showing the scalability of our parallel approach, by first deciding on what threshold parameters and strategies to use, and then by showing that it obtains almost linear scalability up to 128 processors.

*The outcome of any serious research can
only be to make two questions grow where
only one grew before.*

Thorstein Veblen

7

Conclusions and Future Work

Complex Networks are ubiquitous in artificial and natural systems. The identification of network motifs provides a new tool towards the understanding of their meaning and function. Motif discovery has already been applied to networks in many fields, such as biology, chemistry, sociology and engineering. This is however a *hard* problem from a computational point of view and current usage of motifs is strongly limited by the time it takes identify them in a network within a reasonable amount of time. Faster methods to identify motifs would have a strong multidisciplinary impact on our understanding of networks. The purpose of this work was precisely to provide such methods.

This chapter summarizes the main results obtained, points out the main contributions and then discusses limitations and directions for future research.

7.1 Summary

This work overviews the state-of-the-art in network motif discovery, describing all major methods with an emphasis on their algorithmic aspects. Despite being a relatively young field, with the core definition and initial algorithm appearing only in 2002 [MSOI⁺02], the related published literature is already substantial.

The core of the work resides on computing frequencies of subgraphs. From a conceptual point of view, current methods follow two very distinct approaches: either

CHAPTER 7. CONCLUSIONS AND FUTURE WORK

all possible subgraphs of a given size are computed, by enumerating them and then doing isomorphism tests (as in **ESU** or **Kavosh**) or single subgraphs have to be queried, with the isomorphism tests incorporated in the search (as in **Grochow and Kellis** or **MODA**).

The bulk of the work to be done is due to the computation of the frequency of subgraphs on a large set of randomized networks. However, we are only interested in the frequency of the set of interesting subgraphs that appear in the original network. Computing the full census means potentially spending time finding frequencies that will be discarded because the respective subgraph does not appear in the original network. By contrast, the alternative is to search individually for each subgraph, without taking advantage of the fact that we are really looking for more than one subgraph.

One of the main innovations introduced in this work is to escape these two extreme visions and look at the problem from a new angle. By acknowledging that we are in fact trying to compute the frequency of a set of subgraphs, new solutions can be pursued.

The first question is how to represent this set of subgraphs? **G-Tries** were created with this purpose in mind. They represent a general data-structure able to efficiently store collections of graphs. Its main property is that it identifies common smaller substructures and organizes them hierarchically in a tree.

The first consequence is that we are able to compress the topological information given by the collections of graphs. Moreover, this property enables us to create a novel efficient algorithm capable of finding the frequency of these graphs as subgraphs of another graph. By traversing the tree, we can identify that a set of nodes is already a structural partial match to several possible descendant graphs. This avoids the need to postpone the isomorphism comparison to the end of the computation, as in the network-centric methods. It also avoids having to start over when searching for another graph, as in the subgraph-centric methods. By doing an extensive empirical evaluation we have shown that **g-tries** consistently outperform any other previous sequential method at this task.

Taking advantage of the tree based nature of **g-tries**, we presented a method for uniformly taking samples from it. This enabled the creation of a flexible sampling methodology capable of trading accuracy for faster execution times, further improving the potential of **g-tries**.

Exploiting parallelism in motif discovery has been scarcely done in the past. Here too, we make a contribution by first identifying and describing the opportunities for parallelism that are present in the motif discovery algorithms. We then parallelized the g-trie and ESU algorithms, allowing for a completely parallel execution of motif discovery. In order to obtain a scalable approach, several strategies were proposed. In particular, a novel approach was taken in which we can efficiently stop, store and most importantly divide a computation during execution time. The main result is that we were able to obtain almost linear speedups up to 128 processors.

With all of the algorithms and data-structures developed, we obtained a new motif discovery algorithm that can be several orders of magnitude faster than previous approaches. In doing this, we effectively push the limits on the applicability of network motifs, allowing the identification of larger motifs in larger networks.

7.2 Main contributions

This work makes the following major contributions:

- **Survey and comparison of previous sequential algorithms:** we presented a complete survey of previous algorithms, creating an associated taxonomy able to distinguish methods, a time line of their appearance, an algorithmic view with pseudo-code for all major existing algorithms, a common implementation, and their experimental evaluation and comparison.
- **G-Trie Data Structure and its associated algorithms:** we created and implemented a novel specialized data-structure designed to efficiently store collections of graphs, exploiting common substructures. It allows the efficient computation of subgraph frequencies, by using symmetry breaking techniques and taking advantage of an efficient custom canonical representation that produces a high level of overlapping topologies between subgraphs.
- **G-Trie Sampling:** we designed a flexible way of trading accuracy for better execution times by allowing one to sample only a fraction of all the subgraph occurrences, providing the first method able to sample a specific set of subgraphs of another larger network.
- **Characterization of parallelization opportunities in motif discovery:** we identified opportunities for exploiting parallelism in motif discovery, created

CHAPTER 7. CONCLUSIONS AND FUTURE WORK

an associated taxonomy and showed where it is applicable.

- **General scalable parallelization of subgraph counting:** we designed and implemented a general scalable parallel strategies for g-tries and ESU census of subgraphs, with almost linear speedup up to 128 processors.
- **Novel efficient g-trie based motif discovery algorithm:** by combining the power of g-tries census, g-tries sampling and g-tries parallelization, a new much improved motif detection was developed. It ultimately outperforms previous algorithms by several orders of magnitude.

7.3 Future Work

Despite the contributions to the motif discovery problem, there are still limits on its applicability and areas where much can still be improved. We will now point out some issues that deserve further research.

7.3.1 Overall Contributions to the Community

Given the fact that we have pushed the limits on motif discovery, we feel we are in a position to greatly contribute to the complex network analysis community.

We intend to make our software framework available as an easy to use tool for the practitioners of any network field. This will help in disseminating the g-tries methodology and will allow end users to more efficiently identify network motifs and potentially reach larger motif and network sizes than it was before possible.

We also intend to actively pursue practical applications of our methods, both by computing larger motif sizes for already existing networks, observing if the increase in the scale can provide new insight, and by finding collaborators which have networks as a fundamental data set and have interesting research questions that could be solved by applying our developed algorithms.

Another path is to have a look at new angles of analysis that can be now pursued given the provided novel algorithmic tools. One example that we see as very important is to directly look at very large motifs and understand what is their purpose and usefulness. What is their relationship with smaller motifs? Do they provide new information or do they just reflect the smaller level topology?

7.3.2 Efficiency and Flexibility of the Developed Methods

The developed algorithms can still be improved and here we present several possible research directions.

Automatic and dynamic parameters for the parallel algorithms

The current parallel strategies use parameters that were set empirically. We would like to study how this process can be automated, with the thresholds being determined by the tool itself for any general case of data and computation environment. One possible first step is to automate the process one would empirically do, by testing several possible values, embedding it as a preliminary step to be run before using the parallel algorithms.

Adaptive sampling

Currently the sampling version of g-tries uses a static set of parameters, the probabilities for each level. We intend to create an adaptive version of our sampling algorithm that is able to make an initial quicker estimation and then keeps refining it for the subgraphs that do not have enough estimation quality. For example, one could remove all frequent subgraphs from the g-trie and only repeat the search for the less frequent ones, with a higher fraction of samples. This includes the necessity to have an improved formal method for determining the statistical accuracy of the results obtained.

Study graph labeling

Since our symmetry breaking conditions rely on the order of the vertices, can we relabel the graph in a way that improves the efficiency of the search? Does this labeling affect sampling quality? Should we generate a different set of symmetry breaking conditions that is better able to exploit the node labels? Currently, not much work was done to examine in detail the influence of the network node labels in all aspects of our algorithms, and we intend to do it in a systematic way.

Adaptive hybrid motif discovery

Searching for a single subgraph with a subgraph-centric algorithm such as **Grochow** should be quicker than using a g-trie solely with that subgraph, since no order for discovering the nodes is imposed. As we start to increase the size of the set of subgraphs to look for, when do g-tries start to be faster? We would like to explore this direction and be able to produce an hybrid approach that dynamically adapts to the user request and uses the most efficient way for producing the desired results.

CHAPTER 7. CONCLUSIONS AND FUTURE WORK

Larger networks support

A major limitation of our methodology is that currently it assumes that the entire network will be available in main memory. This limits its applicability to relatively small networks. But nowadays networks with millions or even billions of nodes are becoming widely available, like the twitter interaction network collected by Cha et al. [CHBG], with almost 55 million nodes and 2 billion connections. All other major existing algorithms suffer from this and the first step is to actually be able to compute even very small motifs for such large networks. General tools for very large network analysis such as Pegasus [KTF10], with its usage of the Hadoop [Whi09] platform and the MapReduce concept [Coh09] provide some possible initial directions for this.

7.4 Final remarks

Researching network motif discovery has been a very interesting and challenging task. In the end we feel that the major goals of this dissertation were met and the author has gained an invaluable insight into the world of complex networks as a whole.

From a computer science point of view, the process of creating a new data-structure was most rewarding. Having a strong background on programming contests and a lifelong interest on understanding (and teaching) the beauty of algorithms and data-structures, it felt like a dream come true to be able to contribute with a fundamental way of storing and using collections of graphs.

From an applications point of view, we want the algorithms and data-structures presented in this work to be helpful to practitioners from many fields. This is the ultimate goal of this work and we hope that in the near future this can be achieved.

When finishing this step and closing the window on this dissertation, we feel that many new research doors are being opened. We hope that traveling through those unexplored paths will be as enjoyable and fruitful as traveling along the road that led us here.

Make your mental image clear enough, picture it vividly in every detail, and the Genie-of-your-Mind will speedily bring it into being as an everyday reality.

Robert Collier



G-Tries Anatomy

This appendix contains images of fully functional g-tries generated with the algorithms described in Chapter 4, that is, generated with the **GTCanon** labeling method described in Section 4.4.2.3, and doing the four steps of filtering to reduce the symmetry breaking conditions as described in Section 4.5.2.3.

The g-tries drawing was automatically generated, with the exception of the symmetry breaking conditions. Our own implementation allows for the creation of a PNG image file showing the anatomy of any g-trie stored in memory.

We now present g-tries containing all possible undirected k -subgraphs, as k increases (Figures A.1, A.2 and A.3). Empty condition sets are not shown for better legibility, and the other sets of conditions are shown near the respective nodes. A group of sets of conditions at a node implies that a partial match of a subgraph passing through that node must respect at least one of those sets of conditions.

APPENDIX A. G-TRIES ANATOMY

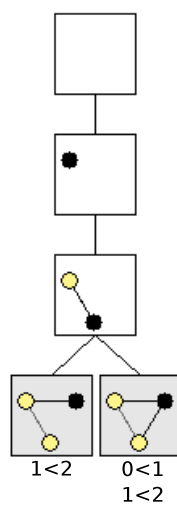


Figure A.1 – A g-trie containing all 2 undirected 3-subgraphs.

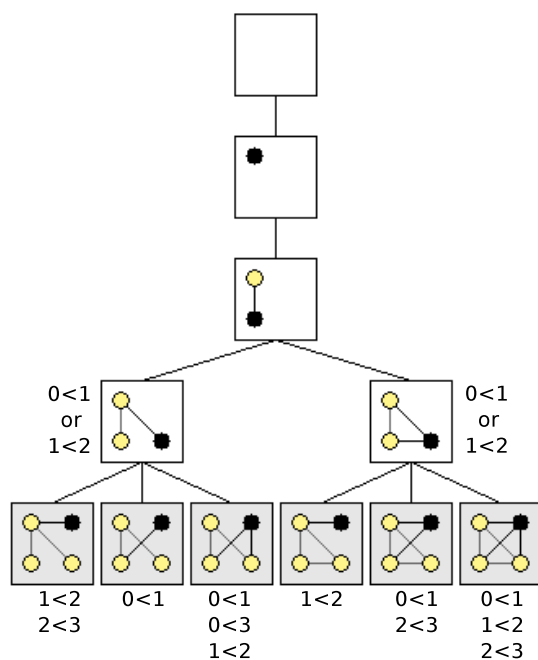


Figure A.2 – A g-trie containing all 6 undirected 4-subgraphs.

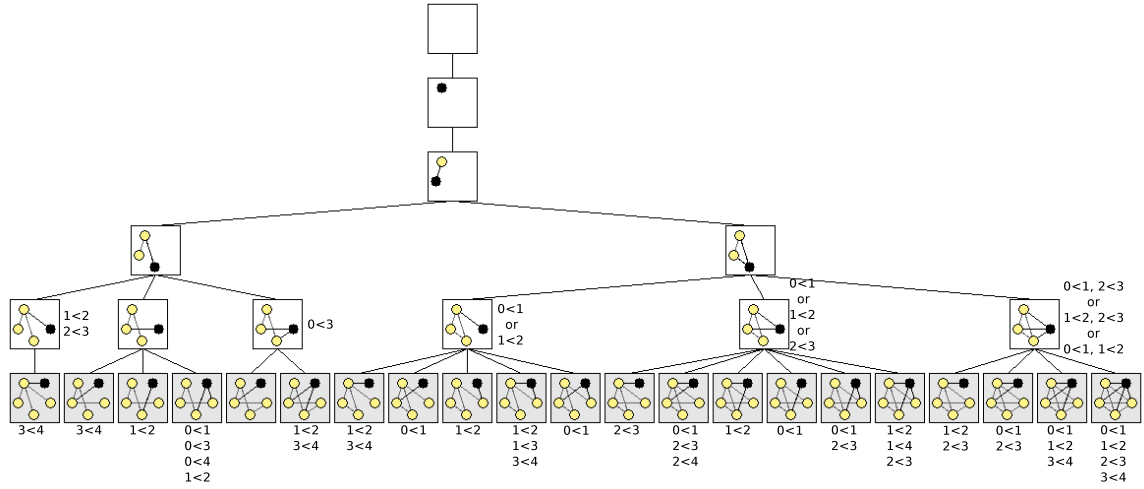


Figure A.3 – A g-trie containing all 21 undirected 5-subgraphs.

The g-trie with all the 112 undirected 6-subgraphs is already too big to be legible on paper. Instead, Figure A.4 shows a g-trie containing only a subset of all the 6-subgraphs, namely the 33 6-subgraphs found in the `circuit` network (see Section 6.1.2 for details on the network). Note that this would be the g-trie used to count the subgraphs appearing in similar randomized networks if we were discovering network motifs on `circuit`.

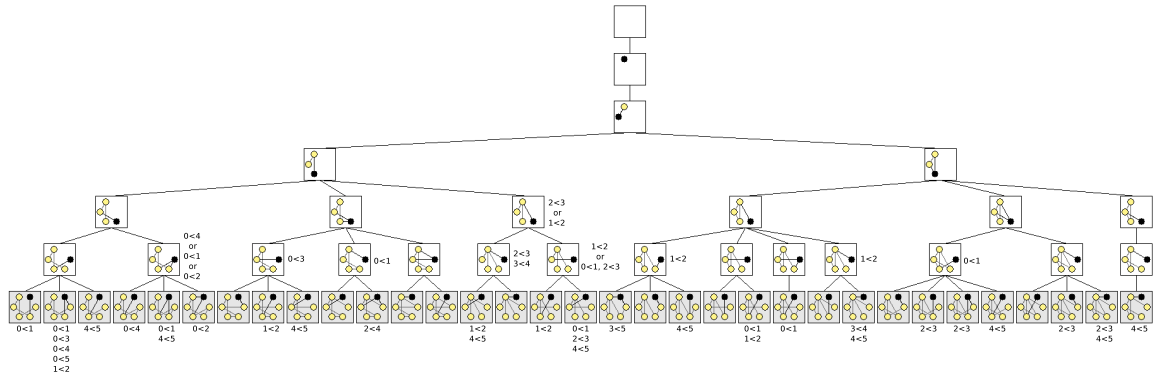


Figure A.4 – A g-trie containing the 33 undirected 6-subgraphs found in `circuit`.

APPENDIX A. G-TRIES ANATOMY

Figure A.5 shows a g-trie containing all 13 possible directed 3-subgraphs. As the g-trie containing all 199 directed 4-subgraphs is too big to be legible on paper, Figure A.6 shows a g-trie containing a subset of these, namely the 24 directed 4-subgraphs found in the `metabolic` network (see Section 6.1.2). This would be the g-trie used for counting the occurrences of subgraphs in the randomized networks if we were discovering network motifs on `metabolic`.

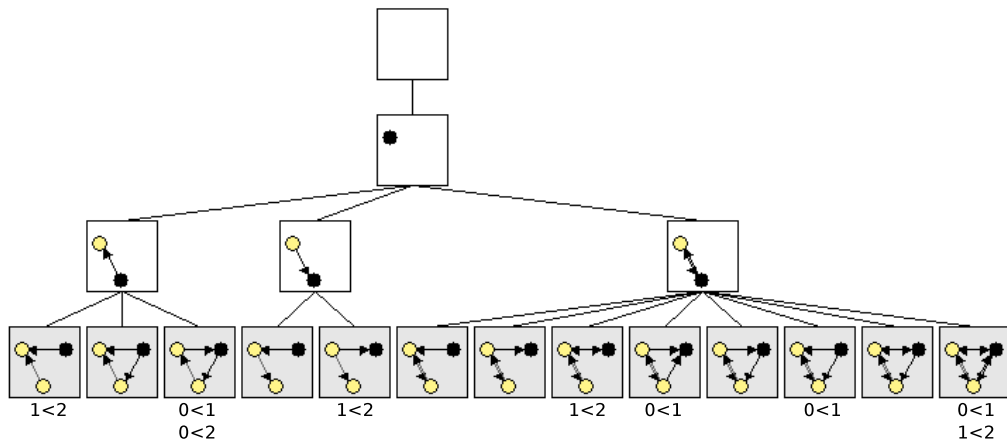


Figure A.5 – A g-trie containing all 13 directed 3-subgraphs.

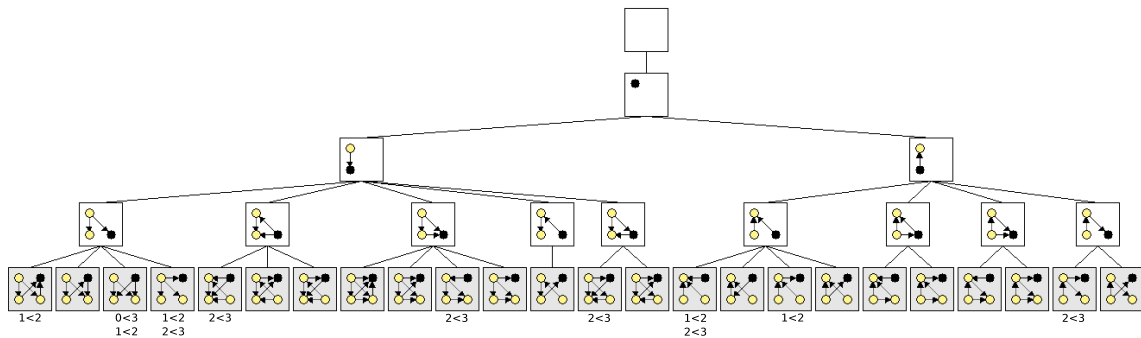


Figure A.6 – A g-trie containing the 24 directed 4-subgraphs found in `metabolic`.

*If you don't think carefully, you might
believe that programming is just typing
statements in a programming language.*

Ward Cunningham

B

G-Tries Implementation

This appendix contains a very brief overview of the current g-tries implementation framework, which was used to produce the results described in Chapter 6.

All code was made in C++ and compiled using gcc 4.1.2, with optimization level 3 turned on (parameter `-O3`). The entire code base is documented using the Doxygen Documentation System [vH11]. Great care was taken to follow good software engineering patterns, to guarantee that the framework would be more easily maintained and updated in the future.

We used the object oriented pattern, but it was not always easy to find a good balance between modularity and efficiency. Whenever possible, general abstractions were used, but in some cases the correspondent overhead meant that some modularity had to be sacrificed for the sake of efficiency and better execution times.

Table B.1 shows the main C++ classes developed, indicating the number of lines of source code for each class (excluding Doxygen related commentaries) and giving a brief description of the functionality. We are aware that lines of source code are a very limited statistic, but our aim is just to give a brief overview on the implementation effort. Classes are shown in lexicographical order.

Figure B.1 shows an high level overview of the main interactions between the used classes. We use two third-party modules in our framework. One is the efficient isomorphism package `nauty`, developed by Brendan McKay [McK81]¹, used for canonical

¹nauty is available at <http://cs.anu.edu.au/~bdm/nauty/>

APPENDIX B. G-TRIES IMPLEMENTATION

Class	Nr Lines	Description
Conditions	217	Generation and maintenance of symmetry breaking conditions
Error	56	A common framework for reporting errors
ESU	142	ESU and Rand-ESU algorithms
Graph	42	An interface for a graph defined as a purely virtual abstract class
GraphFile	256	An experimental implementation of graph using the file system to store graphs that are too large to reside in main memory
GraphList	182	An implementation of graph using adjacency lists
GraphMatrix	195	An implementation of graph using both adjacency matrix and list
GraphTree	255	A binary tree for efficient storage of subgraph frequencies
GraphUtils	146	Utility functions for graph , such as reading from a file
Grochow	401	Grochow algorithm
GTrie	1202	A g-trie implementation and methods to deal with it; uses GTrieNode as an “internal” class; it is the functional core of our implementation
GTrieDraw	115	G-Trie image generation
Isomorphism	495	Isomorphism related functions and canonical representations of graphs
Kavosh	202	Kavosh algorithm
Motifs	341	Motif discovery wrap-up, providing high level functionality
ParCommon	643	Common utilities for the MPI parallel implementations
ParESU	328	Parallel MPI implementation of ESU
ParGTries	472	Parallel MPI implementation of g-tries
Random	76	Generation of random number and networks
Timer	30	Execution time measuring functionality

Table B.1 – Main C++ classes used in our implementation.

labeling. The other is **PNGWriter**², used for generating g-trie image files.

For obtaining the results shown in Chapter 6 we used exclusively the **GraphMatrix** graph implementation, which provides efficient graph primitives by using both an adjacency list and an adjacency matrix, at a cost of a larger amount of main memory used. Our code is however prepared for using other graph implementations that may be slower, but are less demanding in memory terms, such as **GraphFile**, which already provides experimental support for large scale networks in the order of millions of nodes.

We plan to very soon release our source code to the community and make it available online.

²PNGWriter is available at <http://pngwriter.sourceforge.net/>

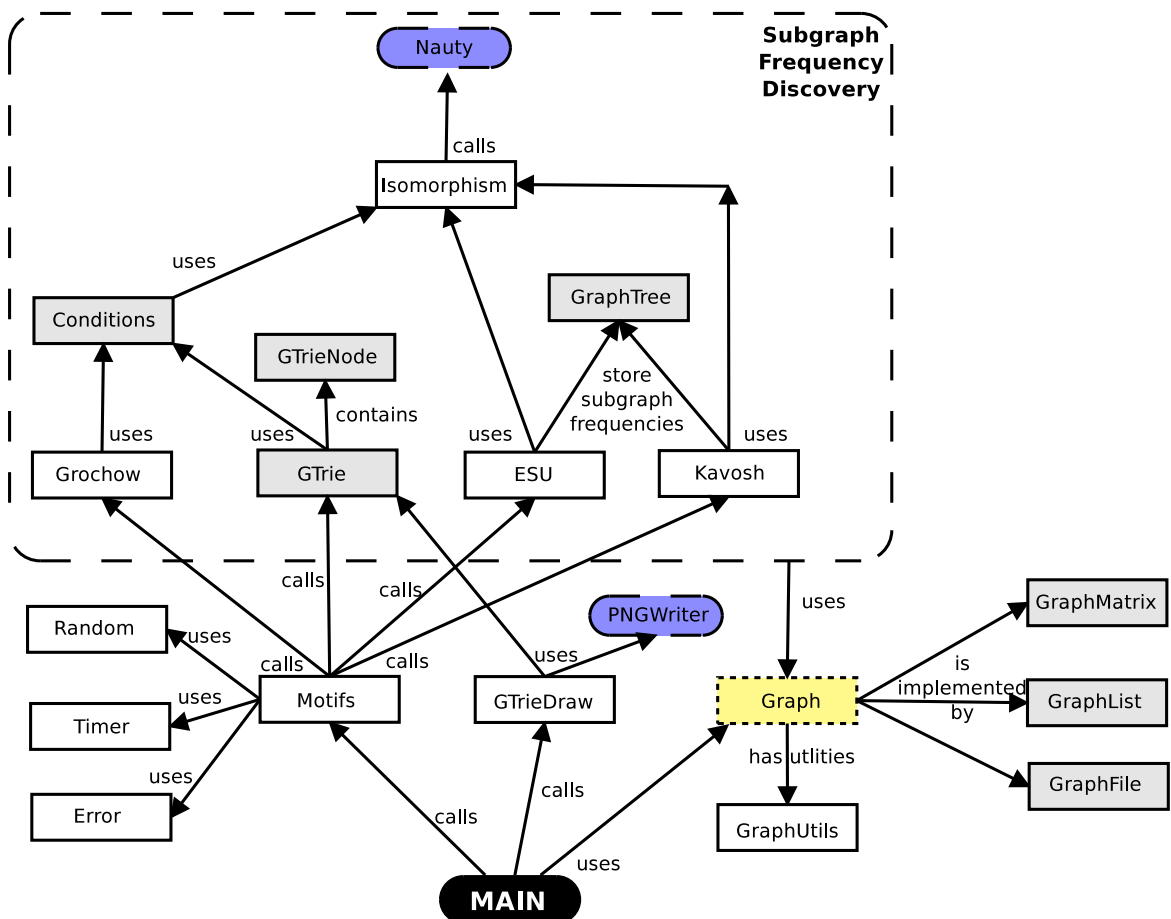
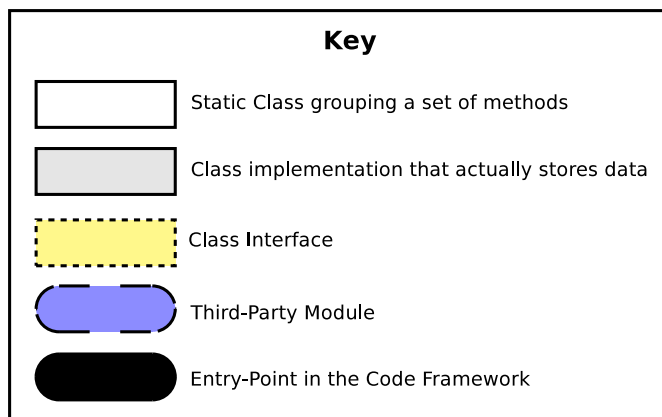


Figure B.1 – Interactions between the C++ classes of our implementation.

References

- [AA03] Eric Alm and Adam P. Arkin. Biological networks. *Current Opinion in Structural Biology*, 13(2):193–202, April 2003.
- [AA04] I. Albert and R. Albert. Conserved network motifs allow protein-protein interaction prediction. *Bioinformatics*, 20(18):3346–3352, December 2004.
- [AB02] Reka Albert and Albert L. Barabasi. Statistical mechanics of complex networks. *Reviews of Modern Physics*, 74(1), 2002.
- [AG05] Lada A. Adamic and Natalie Glance. The political blogosphere and the 2004 U.S. election: divided they blog. In *Proceedings of the 3rd international workshop on Link discovery (LinkKDD)*, pages 36–43, New York, NY, USA, 2005. ACM.
- [Alo03] U. Alon. Biological networks: the tinkerer as an engineer. *Science*, 301(5641):1866–7+, 2003.
- [Are11] Alex Arenas. Network data sets.
<http://deim.urv.cat/~aarenas/data/welcome.htm>, March 2011.
- [ARFBTS02] Yael Artzy-Randrup, Sarel J. Fleishma, Nir Ben-Ta, and Lewi Stone. Technical comment on "network motifs: simple building blocks of complex networks". *Science*, 298(5594):824–827, October 2002.
- [ASBS00] L. A. N. Amaral, A. Scala, M. Barthélemy, and H. E. Stanley. Classes of small-world networks. *Proceedings of the National Academy of Sciences*, 97(21):11149–11152, October 2000.
- [BA99] A. L. Barabasi and R. Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, October 1999.
- [BB02] Christian Borgelt and Michael R. Berthold. Mining molecular fragments: Finding relevant substructures of molecules. In *Proceedings of the 2nd IEEE International Conference on Data Mining (ICDM)*, Washington, DC, USA, 2002. IEEE Computer Society Press.

REFERENCES

- [BHJ] Mathieu Bastian, Sebastien Heymann, and Mathieu Jacomy. Gephi: An open source software for exploring and manipulating networks. In *Proceedings of the 3rd International AAAI Conference on Weblogs and Social Media (ICWSM)*.
- [Bir11] Etienne Birmelé. Detecting local network motifs. *Annals of Applied Probability (in press)*, 2011.
- [BLM⁺06] S. Boccaletti, V. Latora, Y. Moreno, M. Chavez, and D. Hwang. Complex networks: Structure and dynamics. *Physics Reports*, 424(4-5):175–308, February 2006.
- [BM06] Vladimir Batagelj and Andrej Mrvar. Pajek datasets. <http://vlado.fmf.uni-lj.si/pub/networks/data/>, 2006.
- [BZC⁺03] Dongbo Bu, Yi Zhao, Lun Cai, Hong Xue, Xiaopeng Zhu, Hongchao Lu, Jingfen Zhang, Shiwei Sun, Lunjiang Ling, Nan Zhang, Guojie Li, and Runsheng Chen. Topological structure analysis of the protein-protein interaction network in budding yeast. *Nucleic Acids Research*, 31(9):2443–2450, May 2003.
- [CF06] Deepayan Chakrabarti and Christos Faloutsos. Graph mining: Laws, generators, and algorithms. *ACM Computing Surveys*, 38:1, 2006.
- [CG08] G. Ciriello and C. Guerra. A review on models and algorithms for motif discovery in protein-protein interaction networks. *Briefings in Functional Genomics*, 7(2):147–156, 2008.
- [CHBG] Meeyoung Cha, Hamed Haddadi, Fabricio Benevenuto, and Krishna P. Gummadi. Measuring User Influence in Twitter: The Million Follower Fallacy. In *Proceedings of the 4th International AAAI Conference on Weblogs and Social Media (ICWSM)*.
- [CHK⁺01] D. S. Callaway, J. E. Hopcroft, J. M. Kleinberg, M. E. J. Newman, and S. H. Strogatz. Are randomly grown graphs really random? *Physical Review E (Statistical, Nonlinear, and Soft Matter Physics)*, 64(4):041902, Sep 2001.
- [CHLN06] J. Chen, W. Hsu, M. Li Lee, and See-Kiong Ng. Nemofinder: dissecting genome-wide protein-protein interactions with meso-scale network motifs. In *Proceedings of the 12th ACM SIGKDD international conference*

REFERENCES

- on Knowledge Discovery and Data Mining (KDD)*, pages 106–115, New York, NY, USA, 2006. ACM.
- [Coh09] Jonathan Cohen. Graph Twiddling in a MapReduce World. *Computing in Science and Engineering*, 11(4):29–41, July 2009.
- [Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the 3rd annual ACM symposium on Theory of computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM.
- [DA05] J. Duch and A. Arenas. Community detection in complex networks using extremal optimization. *Physical Review E (Statistical, Nonlinear, and Soft Matter Physics)*, 72:027104, 2005.
- [DBBO04] R. Dobrin, Q. K. Beg, A. Barabasi, and Z. Oltvai. Aggregation of topological motifs in the escherichia coli transcriptional regulatory network. *BMC Bioinformatics*, 5:10, 2004.
- [dFCOT⁺07] Luciano da F. Costa, Osvaldo N. Oliveira Jr, Gonzalo Travieso, Francisco A. Rodrigues, Paulino R. Villas Boas, Lucas Antiqueira, Matheus P. Viana, and Luis E. C. da Rocha. Analyzing and modeling real-world phenomena with complex networks: A survey of applications. *ArXiv e-prints*, 0711(3199), 2007.
- [dFCRTB07] Luciano da F. Costa, Francisco A. Rodrigues, Gonzalo Travieso, and P. R. Villas Boas. Characterization of complex networks: A survey of measurements. *Advances In Physics*, 56:167, 2007.
- [DM02] S. N. Dorogovtsev and J. F. F. Mendes. Evolution of networks. *Advances in Physics*, 51(4):1079–1187, June 2002.
- [ELZ85] Derek L. Eager, Edward D. Lazowska, and John Zahorjan. A comparison of receiver-initiated and sender-initiated adaptive load sharing (extended abstract). *ACM SIGMETRICS - Performance Evaluation Review*, 13(2):1–3, 1985.
- [ER59] P. Erdős and A. Rényi. On random graphs. I. *Publicationes Mathematicae Debrecen*, 6:290–297, 1959.
- [FFHV07] M. Fellows, G. Fertin, D. Hermelin, and S. Vialette. Sharp tractability borderlines for finding connected motifs in vertex-colored graphs. In *Pro-*

REFERENCES

- ceedings of the 34th International Colloquium on Automata, Languages and Programming (ICALP)*, 2007.
- [Fre60] Edward Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, September 1960.
- [GK07] J. Grochow and M. Kellis. Network motif discovery using subgraph enumeration and symmetry-breaking. *Research in Computational Molecular Biology*, pages 92–106, 2007.
- [GWS06] Richard Graham, Timothy Woodall, and Jeffrey Squyres. Open mpi: A flexible high performance mpi. In *Parallel Processing and Applied Mathematics*, volume 3911 of *Lecture Notes in Computer Science*, pages 228–239. Springer, 2006.
- [HBP⁺06] Jun Huan, Deepak B, Jan Prins, Jack Snoeyink, Er Tropscha, and Wei Wang. Distance-based identification of structure motifs in proteins using constrained frequent subgraph mining. In *Proceedings of the Conference in Computational Systems Bioinformatics (CSB)*, pages 227–238, 2006.
- [HJ05] J. Hallinan and P. Jackway. Network motifs, feedback loops and the dynamics of genetic regulatory networks. In *Proceedings of the IEEE Symposium on Computational Intelligence in Bioinformatics and Computational Biology (CIBCB)*, 2005.
- [How10] Denis Howe. Foldoc, free online dictionary of computing. <http://foldoc.org/>, 2010.
- [HSL⁺00] Elisa Heymann, Miquel A. Senar, Emilio Luque, , and Miron Livny. Evaluation of an adaptive scheduling strategy for master-worker applications on clusters of workstations. In *Proceedings of the 7th International Conference on High Performance Computing (HiPC 2000)*, Bangalore, India, December 2000.
- [Hus99] Lars Paul Huse. Collective communication on dedicated clusters of workstations. In *Proceedings of the 6th European PVM/MPI Users’ Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 469–476, London, UK, 1999. Springer-Verlag.

REFERENCES

- [HWP03] Jun Huan, Wei Wang, and Jan Prins. Efficient mining of frequent subgraphs in the presence of isomorphism. In *Proceedings of the 3rd IEEE International Conference on Data Mining (ICDM)*, page 549, Washington, DC, USA, 2003. IEEE Computer Society Press.
- [IA05] Shalev Itzkovitz and Uri Alon. Subgraphs and network motifs in geometric networks. *Physical Review E (Statistical, Nonlinear, and Soft Matter Physics)*, 71(2):026117, Feb 2005.
- [ILK⁺05] S. Itzkovitz, R. Levitt, N. Kashtan, R. Milo, M. Itzkovitz, and U. Alon. Coarse-graining and self-dissimilarity of complex networks. *Physical Review E (Statistical, Nonlinear, and Soft Matter Physics)*, 71(1 Pt 2), January 2005.
- [ISS06] P. J. Ingram, M. PH Stumpf, and J. Stark. Network motifs: structure does not determine function. *BMC Genomics*, 7:108, 2006.
- [JHQ09] Lin Gao Jialu Hu and Guimin Qin. Evaluation of subgraph searching algorithms detecting network motif in biological networks. *Frontiers in Computational Science*, pages 412–416, 2009.
- [JMA07] Ruoming Jin, Scott McCallen, and Eivind Almaas. Trend motif: A graph mining approach for analysis of dynamic complex networks. In *Proceedings of the 7th IEEE International Conference on Data Mining (ICDM)*, pages 541–546, Washington, DC, USA, 2007. IEEE Computer Society Press.
- [KÖ8] Lasse Kärkkäinen. Yet another java vs. c++ shootout. <http://zi.fi/shootout/>, 2008.
- [KAE⁺09] Zahra Kashani, Hayedeh Ahrabian, Elahe Elahi, Abbas Nowzari-Dalini, Elnaz Ansari, Sahar Asadi, Shahin Mohammadi, Falk Schreiber, and Ali Masoudi-Nejad. Kavosh: a new algorithm for finding network motifs. *BMC Bioinformatics*, 10(1):318, 2009.
- [KIMA04a] N. Kashtan, S. Itzkovitz, R. Milo, and U. Alon. Efficient sampling algorithm for estimating subgraph concentrations and detecting network motifs. *Bioinformatics*, 20(11):1746–1758, 2004.
- [KIMA04b] N. Kashtan, S. Itzkovitz, R. Milo, and U. Alon. Topological generalizations of network motifs. *Physical Review E (Statistical, Nonlinear, and Soft Matter Physics)*, 70(3 Pt 1), September 2004.

REFERENCES

- [KKM00] Ton Kloks, Dieter Kratsch, and Haiko Müller. Finding and counting small induced subgraphs efficiently. *Information Processing Letters*, 74(3-4):115–121, 2000.
- [KL08] Arun Konagurthu and Arthur Lesk. On the origin of distribution patterns of motifs in biological networks. *BMC Systems Biology*, 2(1):73+, 2008.
- [KNS08] Johannes F. Knabe, Chrystopher L. Nehaniv, and Maria J. Schilstra. Do motifs reflect evolved function? - no convergent evolution of genetic regulatory network subgraph topologies. *BioSystems*, 94(1-2):68–74, 2008.
- [Kon08] Michio Kondoh. Building trophic modules into a persistent food web. *Proceedings of the National Academy of Sciences*, 105(43):16631–16635, 2008.
- [KS99] Donald L. Kreher and Douglas R. Stinson. Combinatorial algorithms: generation, enumeration, and search. *SIGACT News*, 30(1):33–35, 1999.
- [KST93] Johannes Köbler, Uwe Schöning, and Jacobo Torán. *The graph isomorphism problem: its structural complexity (Progress in Theoretical Computer Science)*. Birkhauser Verlag, Basel, Switzerland, 1993.
- [KTF10] U Kang, C.E Tsourakakis, and C. Faloutsos. Pegasus: Mining peta-scale graphs. *Knowledge and Information Systems*, 2010.
- [LFR08a] Andrea Lancichinetti, Santo Fortunato, and Filippo Radicchi. Benchmark graphs for testing community detection algorithms. *Physical Review E (Statistical, Nonlinear, and Soft Matter Physics)*, 78(4):046110, Oct 2008.
- [LFR08b] Andrea Lancichinetti, Santo Fortunato, and Filippo Radicchi. Benchmark graphs for testing community detection algorithms. *Physical Review E (Statistical, Nonlinear, and Soft Matter Physics)*, 78(4), 2008.
- [LFS06] V. Lacroix, C. G. Fernandes, and M. F. Sagot. Motif search in graphs: Application to metabolic networks. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 3(4):360–368, Oct.-Dec. 2006.

REFERENCES

- [LRR⁺02] T. I. Lee, N. J. Rinaldi, F. Robert, D. T. Odom, Z. Bar-Joseph, G. K. Gerber, N. M. Hannett, C. T. Harbison, C. M. Thompson, I. Simon, J. Zeitlinger, E. G. Jennings, H. L. Murray, D. B. Gordon, B. Ren, J. J. Wyrick, J. B. Tagne, T. L. Volkert, E. Fraenkel, D. K. Gifford, and R. A. Young. Transcriptional regulatory networks in *saccharomyces cerevisiae*. *Science*, 298(5594):799–804, October 2002.
- [LSB⁺03] D. Lusseau, K. Schneider, O. J. Boisseau, P. Haase, E. Slooten, and S. M. Dawson. The bottlenose dolphin community of doubtful sound features a large proportion of long-lasting associations. can geographic isolation explain this unique trait? *Behavioral Ecology and Sociobiology*, 54(4):396–405, 2003.
- [MA03] S. Mangan and U. Alon. Structure and function of the feed-forward loop network motif. *Proceedings of the National Academy of Sciences*, 100(21):11980–11985, October 2003.
- [MBV05] A. Mazurie, S. Bottani, and M. Vergassola. An evolutionary and functional assessment of regulatory network motifs. *Genome Biology*, 6:R35, 2005.
- [McK81] Brendan McKay. Practical graph isomorphism. *Congressus Numerantium*, 30:45–87, 1981.
- [McK98] B. McKay. Isomorph-free exhaustive generation. *Journal of Algorithms*, 26(2):306–324, 1998.
- [MIK⁺04] R. Milo, S. Itzkovitz, N. Kashtan, R. Levitt, S. Shen-Orr, I. Ayzenshtat, M. Sheffer, and U. Alon. Superfamilies of evolved and designed networks. *Science*, 303(5663):1538–1542, March 2004.
- [MSB⁺06] C. Matias, S. Schbath, E. Birmelé, J.-J. Daudin, and S. Robin. Network motifs: mean and variance for the count. *REVSTAT*, 4:31–35, 2006.
- [MSOI⁺02] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon. Network motifs: simple building blocks of complex networks. *Science*, 298(5594):824–827, October 2002.
- [MZA03] S Mangan, A Zaslaver, and U Alon. The coherent feedforward loop serves as a sign-sensitive delay element in transcription networks. *Journal of Molecular Biology*, 334(2):197 – 204, 2003.

REFERENCES

- [MZW05] M. Middendorff, E. Ziv, and C. H. Wiggins. Inferring network mechanisms: the drosophila melanogaster protein interaction network. *Proceedings of the National Academy of Sciences*, 102(9):3192–3197, March 2005.
- [New03] M. E. J. Newman. The structure and function of complex networks. *SIAM Review*, 45, 2003.
- [New06] M. E. J. Newman. Finding community structure in networks using the eigenvectors of matrices. *Physical Review E (Statistical, Nonlinear, and Soft Matter Physics)*, 74(3):036104, Sep 2006.
- [New09] Mark Newman. Network data.
<http://www-personal.umich.edu/~mejn/netdata/>, June 2009.
- [NK04] Siegfried Nijssen and Joost N. Kok. Frequent graph mining and its application to molecular databases. In *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics (SMC)*, pages 4571–4577. IEEE, 2004.
- [NLGC02] K. Norlen, G. Lucas, M. Gebbie, and J. Chuang. EVA: Extraction, Visualization and Analysis of the Telecommunications and Media Ownership Network. In *Proceedings of the International Telecommunications Society 14th Biennial Conference (ITS), Seoul Korea,*. International Telecommunications Society, August 2002.
- [NS07] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM SIGPLAN Notices*, 42:89–100, June 2007.
- [NSW01] M. E. J. Newman, S. H. Strogatz, and D. J. Watts. Random graphs with arbitrary degree distributions and their applications. *Physical Review E (Statistical, Nonlinear, and Soft Matter Physics)*, 64(2):026118, Jul 2001.
- [oSA99] The Editors of Scientific American. *Scientific American Book of the Brain*. The Lyons Press, 1999.
- [OSMN09] Saeed Omid, Falk Schreiber, and Ali Masoudi-Nejad. Moda: An efficient algorithm for network motif discovery in biological networks. *Genes & genetic systems*, 84(5):385–395, 2009.

REFERENCES

- [PBTL99] Nicolas Pasquier, Yves Bastide, Rafik Taouil, and Lotfi Lakhal. Discovering frequent closed itemsets for association rules. In *ICDT '99: Proceedings of the 7th International Conference on Database Theory*, pages 398–416, London, UK, 1999. Springer-Verlag.
- [PDK⁺08] F Picard, J-J J. Daudin, M Koskas, S Schbath, and S Robin. Assessing the exceptionality of network motifs. *Journal of Computational Biology*, February 2008.
- [Rei02] Joan Reitz. Odlis: Online dictionary of library and information science. <http://vlado.fmf.uni-lj.si/pub/networks/data/dic/odlis/odlis.pdf>, 2002.
- [RI07] Yoram Louzoun Royi Itzhack, Yelena Mogilevski. An optimal algorithm for counting network motifs. *Physica A*, 387:482–490, 2007.
- [Ros06] Martin Rosvall. *Information Horizons in a Complex World*. PhD thesis, Umea University, 2006.
- [RS10a] Pedro Ribeiro and Fernando Silva. Efficient subgraph frequency estimation with g-tries. In *International Workshop on Algorithms in Bioinformatics (WABI)*, volume 6293 of *Lecture Notes In Computer Science*, pages 238–249. Springer, September 2010.
- [RS10b] Pedro Ribeiro and Fernando Silva. G-tries: an efficient data structure for discovering network motifs. In *Proceedings of the 25th ACM Symposium on Applied Computing (SAC)*, pages 1559–1566. ACM, March 2010.
- [RSK09] Pedro Ribeiro, Fernando Silva, and Marcus Kaiser. Strategies for network motifs discovery. In *Proceedings of the 5th IEEE International Conference on e-Science*, pages 80–87, Oxford, UK, December 2009. IEEE Computer Society Press.
- [RSL10a] Pedro Ribeiro, Fernando Silva, and Luís Lopes. Efficient parallel subgraph counting using g-tries. In *Proceedings of the IEEE International Conference on Cluster Computing (Cluster)*, pages 1559–1566. IEEE Computer Society Press, September 2010.
- [RSL10b] Pedro Ribeiro, Fernando Silva, and Luís Lopes. Parallel calculation of subgraph census in biological networks. In *Proceedings of the 1st International Conference on Bioinformatics*, pages 56–65, Valencia, Spain, January 2010. INSTICC.

REFERENCES

- [RSL11] Pedro Ribeiro, Fernando Silva, and Luís Lopes. A parallel algorithm for counting subgraphs in complex networks. In *3rd International Joint Conference on Biomedical Engineering Systems and Technologies - Revised Selected Papers*, volume 127 of *Communications in Computer and Information Science*, pages 380–393. Springer, 2011.
- [RSLar] Pedro Ribeiro, Fernando Silva, and Luís Lopes. Parallel discovery of network motifs. *Journal of Parallel and Distributed Computing*, 2011 (to appear).
- [RSM03] Ricardo Rocha, Fernando Silva, and Rolando Martins. YapDSS: an Or-Parallel Prolog System for Scalable Beowulf Clusters. In *Proceedings of the 11th Portuguese Conference on Artificial Intelligence, EPIA 2003, Beja, Portugal.*, volume 2902 of *LNAI*, pages 136–150. Springer-Verlag, December 2003.
- [San94] Peter Sanders. A detailed analysis of random polling dynamic load balancing. In *Proceedings of the International Symposium on Parallel Architectures Algorithms and Networks*, pages 382–389. IEEE, 1994.
- [San99] Peter Sanders. Asynchronous random polling dynamic load balancing. In *Proceedings of the International Symposium on Algorithms and Computation*, 1999.
- [SCBB08] Michael Schatz, Elliott Cooper-Balis, and Adam Bazinet. Parallel network motif finding. 2008.
- [SK04] Olaf Sporns and Rolf Kotter. Motifs in brain networks. *PLoS Biology*, 2, 2004.
- [SLS08] S. Schbath, V. Lacroix, and M. Sagot. Assessing the exceptionality of coloured motifs in networks. *EURASIP Journal on Bioinformatics and Systems Biology*, 2008.
- [SOMMA02] S. S. Shen-Orr, R. Milo, S. Mangan, and U. Alon. Network motifs in the transcriptional regulation network of escherichia coli. *Nature Genetics*, 31(1):64–68, May 2002.
- [SpOKK05] Jari Saramaki, Jukka pekka Onnela, Janos Kertész, and Kimmo Kaski. Characterizing motifs in weighted complex networks. In *Science of Complex Networks: From Biology to the Internet and WWW*, volume 776 of *AIP Conference Proceedings*, pages 108–117, 2005.

REFERENCES

- [SS04] Falk Schreiber and Henning Schwobbermeyer. Towards motif detection in networks: Frequency concepts and flexible search. In *Proceedings of the International Workshop on Network Tools and Applications in Biology (NETTAB)*, pages 91–102, 2004.
- [Tar71] Robert Tarjan. Depth-first search and linear graph algorithms. In *Proceedings of the Annual IEEE Symposium on Foundations of Computer Science*, pages 114–121, Los Alamitos, CA, USA,, 1971. IEEE Computer Society.
- [VDS⁺04] A. Vazquez, R. Dobrin, D. Sergi, J. P. Eckmann, Z. N. Oltvai, and A. L. Barabasi. The topological relationship between the large-scale attributes and local interaction patterns of complex networks. *Proceedings of the National Academy of Sciences*, 101:17945, 2004.
- [vH11] Dimitri van Heesch. Doxygen: Source code documentation generator tool. <http://www.stack.nl/~dimitri/doxygen/>, 2011.
- [VS05] Sergi Valverde and Ricard V. Solé. Network motifs in computational graphs: A case study in software architecture. *Physical Review E (Statistical, Nonlinear, and Soft Matter Physics)*, 72(2), 2005.
- [Wer05] Sebastian Wernicke. A faster algorithm for detecting network motifs. In *International Workshop on Algorithms in Bioinformatics (WABI)*, volume 3692 of *Lecture Notes in Computer Science*, pages 165–177. Springer, 2005.
- [Wer06] Sebastian Wernicke. Efficient detection of network motifs. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 3(4):347–359, 2006.
- [Wer10] Sebastian Wernicke. Comment on ‘an optimal algorithm for counting networks motifs’. *Physica A*, 2010.
- [WFI94] Stanley Wasserman, Katherine Faust, and Dawn Iacobucci. *Social Network Analysis : Methods and Applications (Structural Analysis in the Social Sciences)*. Cambridge University Press, November 1994.
- [Whi09] Tom White. *Hadoop: The Definitive Guide*. O’Reilly Media, 1 edition, June 2009.

REFERENCES

- [WHV⁺95] Yi-Min Wang, Yennun Huang, Kiem-Phong Vo, Pe-Yu Chung, and C. Kintala. Checkpointing and its applications. In *Proceedings of the Twenty-Fifth International Symposium on Fault-Tolerant Computing, FTCS '95*, pages 22–, Washington, DC, USA, 1995. IEEE Computer Society.
- [WOB03] S. Wuchty, Z. Oltvai, and A. Barabasi. Evolutionary conservation of motif constituents within the yeast protein interaction network. *Nature Genetics*, 35:176, 2003.
- [WP04] Chao Wang and Srinivasan Parthasarathy. Parallel algorithms for mining frequent structural motifs in scientific data. In *ACM International Conference on Supercomputing (ICS)*, 2004.
- [WP08] Jun Wang and Gregory Provan. Generating application-specific benchmark models for complex systems. In *Proceedings of the 23rd national conference on Artificial intelligence*, volume 1, pages 566–571. AAAI Press, 2008.
- [WS98] D. J. Watts and S. H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393(6684):440–442, June 1998.
- [WSTB86] J. G. White, E. Southgate, J. N. Thomson, and S. Brenner. The Structure of the Nervous System of the Nematode *Caenorhabditis elegans*. *Philosophical Transactions of the Royal Society of London. B, Biological Sciences*, 314(1165):1–340, November 1986.
- [WTZ⁺05] Tie Wang, Jeffrey W. Touchman, Weiyi Zhang, Edward B. Suh, and Guoliang Xue. A parallel algorithm for extracting transcription regulatory network motifs. In *Proceedings of the IEEE International Symposium on Bioinformatics and Bioengineering*, pages 193–200, Los Alamitos, CA, USA, 2005. IEEE Computer Society Press.
- [YCLH06] Kai-Hsiang Yang, Kun-Yan Chiou, Hahn-Ming Lee, and Jan-Ming Ho. Web appearance disambiguation of personal names based on network motif. In *Proceedings of the IEEE/WIC/ACM International Conference on Web Intelligence (WI)*, pages 386–389, Washington, DC, USA, 2006. IEEE Computer Society Press.
- [YH02] Xifeng Yan and Jiawei Han. gspan: Graph-based substructure pattern mining. In *Proceedings of the 2nd IEEE International Conference on*

REFERENCES

- Data Mining (ICDM)*, page 721, Washington, DC, USA, 2002. IEEE Computer Society Press.
- [Zac77] W. W. Zachary. An information flow model for conflict and fission in small groups. *Journal of Anthropological Research*, 33:452–473, 1977.
- [ZKMW05] Etay Ziv, Robin Koytcheff, Manuel Middendorf, and Chris Wiggins. Systematic identification of statistically significant network measures. *Physical Review E (Statistical, Nonlinear, and Soft Matter Physics)*, 71:016110, 2005.