University of Szeged

Szegedi Tudományegyetem

Természettudományi és Informatikai Kar

SZTE Informatika Doktori Iskola

Doctoral Dissertation

The k-Clique Problem

Usage, Modeling Expressivity, Serial and Massively Parallel Algorithms

Zaválnij, Bogdán

Szeged, 2020.

Advisors:
Dr. Krész, Miklós
Dr. Szabó, Sándor

Contents

List of Figures

List of Tables

Notations

G graph	In this work we consider $G = (V, E)$ finite, undirected and unweighted graphs. It consists of the V set of nodes, and the E set of edges, where an $u, v \in V, \{u, v\} \in E$ edge is an unordered pair of two nodes.
$G(V')$	Induced subgraph in $G(V, E)$ over the node set $V' \subseteq V$. The edges E' of $G(V')$ are the same of G, that is $V' \subseteq V, E' \subseteq E$, and iff $u, v \in V', \{u, v\} \in E$ then $\{u, v\} \in E'$.
Δ clique	$\Delta = (V', E')$ is an all connected spanned subgraph of G. That is $\Delta = G(V')$ and if $u, v \in V'$ then $\{u, v\} \in E'$. We call the size of the Δ clique the size of the set of its nodes $\|V'\|$.
k-clique	We call a clique k-clique, if its size is equal to k.
maximum clique	A clique Δ of G called a maximum clique of G, if no other clique of G has a bigger size than Δ.
$\omega(G)$	The clique size of the G graph, which is the size of a maximum clique Δ of G.
$N(u)$	Neighborhood of the node u, thus all the nodes of G which are adjacent to u. That is the set of all $v \in V$ if $\{u, v\} \in E$. $N(v) = \{u \in V \mid \{u, v\} \in E\}$
$N(\{u, v\})$	Common neighborhood of the node u and v, $N(u) \cap N(v)$. Usually these nodes connected, thus they form an edge of the graph, $\{u, v\} \in E$.
$N(\{u_1, \ldots, u_k\})$	Common neighborhood of the nodes $\{u_1, u_2, \ldots, u_k\}$, $\bigcap_{i=1}^{k} N(u_i)$. Usually these nodes connected, thus they form a k-clique in G.

Chapter 1

Introduction – graphs and cliques

Our thesis work is focused on discrete optimization problems, and specifically on problems represented by graphs. These problems emerge in various applications, and form an interesting subclass of Mathematical Programming. Our thesis concentrates on a special problem of this class, the k-clique problem. We shall in some cases also mention the maximum clique problem as well. The k-clique problem is well known problem from mathematics [Karp1972], but our aim is to show its usage as a modeling and problem solving tool.

As the problems in question belong to NP-complete and NP-hard problem class, these problems are considered to be hard even for medium sized problems. Thus in order to solve them we may want to use more efficient algorithms and more computational power, for example supercomputers. This will lead us to other problems, as dividing the problem to subproblems, scheduling these subproblems, and gathering the results. For discrete optimization problems especially problematic is the huge variation of the complexity of subproblems, and this is known to be a challenging problem for graphs [Madd2007]. If approached by a poor algorithm one may end with a subproblems of which one (or more) will be no easier to solve then the original problem. In this case no or little speedup will be achieved on any supercomputer. The other problem is to deal with the massive core numbers of today's supercomputers. Parallel algorithms developed and test for few cores won't scale up in arcitectures with thousands or even million of cores that must be employed for effective work. In our thesis we will show algorithms especially developed and tested in such environments.

The first chapter of the present work defines the problem class in question, and notes some related problems – those lay outside the scope of our work. We also detail the motivation background of our work.

In the second and third chapter we shall discuss the expressive power of modeling by k-clique. We shall show how to solve problems from different fields by graph modeling and clique search. In the second chapter we try to summarize such problems. We list models for combinatorial games and puzzles like Latin squares, Sudoku game, the problem of non attacking queens, Costas Arrays and combinatorial problems arising from coding theory. A problem class based on subgraph isomorphism including molecule search, protein docking, fingerprint recognition is detailed. We show modeling for complex scheduling like open shop, flow shop and job shop problems. Finally we show a few application in network analysis as market graph or brain graph.

The third chapter shall focus on one problem class, the graph coloring problems, extensively. With detailed examples we show graph based models for legal node coloring, 3-free coloring of nodes and coloring nodes of hypergraphs. We show an application of the last one to an open question by Voloshin [Volo2002].

The fourth chapter tries to show the landscape of the clique search community, lists best practices and approaches. After a short historical review we analyze and compare in details different upper bound procedures. Finally, we speak about nowadays important kernelization techniques.

In the fifth chapter we introduce our own algorithm for k-clique search. We detail the main advantages of our approach, and compare it to other state-of-the-art solvers. The comparison is made by comparing our program to maximum clique solvers, and for that aim we constructed a maximum clique solver from our k-clique solver. It turned out, that our construction even have advantages against other maximum clique algorithms. (Note, that to our knowledge there is no other specialized k-clique solver but ours.)

The sixth chapter is about the theoretical concepts of parallelization, with special focus on parallelization of combinatorial problems. We speak about the hardware and software background, detail some problems of parallel algorithms, and speak about their evaluation.

In the seventh chapter we are introducing the concept of disturbing structures and propose methodology for algorithm parallelization. We will show that dealing with the k-clique problem instead of the maximum clique problem has its major advantages in parallelization, as it opens up quite a lot possibilities.

In the eight chapter, based on the ideas of the previous chapter, we present an implementation of a parallel algorithm. This algorithm is capable of massive parallelization, as we will show that it can scale up even on several hundreds of cores. Driven from the experiment we are introducing the Las Vegas method of dealing with combinatorial optimization subproblems.

The last chapter draws conclusions and aims for future work. In this chapter we also point out which of the results in this work is my own result.

1.1 Definition of the problems

Let $G = (V, E)$ be a finite simple graph. Here V is the set of nodes of the graph, and E is a subset of the Cartesian product $V \times V$. The set V is finite and consequently the set E is also finite. The graph does not contain any double edges and the graph does not contain any loops. Of course the graph cannot contain any triple or quadruple edges. The edges are undirected and there are no weights assigned to the nodes nor to the edges.

Consider a subgraph $\Delta = (U, F)$ of G. We say that Δ is a clique in G if $F = U \times U$. In other words Δ is a clique in G if each two distinct nodes of Δ are adjacent in G. The number of nodes of Δ, that is, the size of the set U is called the size of the clique Δ. Instead of saying that Δ is a clique of size k we sometimes say that Δ is a k-clique in the graph G.

A clique Δ in the graph G is called a maximal clique in G if for any clique Ω in G for which $\Delta \subseteq \Omega$ holds it follows that $\Delta = \Omega$. In other words Δ is a maximal clique in G if Δ cannot be extended to a larger clique in G by adding a node of G to Δ.

A clique Δ is a maximum clique in G if G does not contain any clique whose size is bigger then the size of Δ.

It is an empirical fact that finding cliques in a given graph has many applications inside and outside of computer science. We state the most commonly occurring clique search problems in a formal matter.

Problem 1.1. *We are given a finite simple graph G. Let us determine the size of a maximum clique in G.*

The size of all the maximum cliques in G is well defined common value and it is called the clique number of G. The clique number of G is denoted by $\omega(G)$. Problem 1.1 is referred as the maximum clique problem.

The following decision problem is commonly called the k-clique problem:

Problem 1.2. *Given a finite simple graph G and given a positive integer k. The task is to decide if G contains a k-clique.*

The complexity theory of algorithms teaches us that the maximum clique problem is an NP-hard problem. while the k-clique problem is a well known NP-complete problem, and appears 3$^{\text{rd}}$ among Karp's original 21 NP-complete problems [Karp1972]. In our thesis we focus on this problem, the k-clique problem, although sometimes we will refer to the first one as well.

The solutions for these problems are fall into two categories. Our thesis will deal exclusively with exact methods, but we need to mention that heuristic methods that won't certainly lead to optimum solution are also widely used. There are some smaller problems, which with the aid of modern exact algorithms can be solved efficiently and fast. Other problems are too big for these and heuristic methods applied for finding non exact solution. These solutions may be of great use in some cases, but in others one certainly needs to find an exact solution. In our thesis we would like to widen the abilities of the exact methods by using massively parallel algorithms and supercomputers.

The two problems listed are obviously connected. A program that solves one can be used to solve the other as well. A maximum clique search program obviously also answers if there is a k-clique present in the graph for any k. A program that solves the k-clique problem also can be used for finding maximum clique by a sequence of several runs with different values of k. This method will be described in details in Chapter 5.

1.1.1 Other related problems

There are several connected problems to the already described ones. Obviously the problem of independent sets – maximum independent set, independent set of size k – are the same problems: one should apply a maximum clique or k-clique algorithm on the complement graph. The k vertex cover – and the minimum vertex cover problem – is also the same, its solution is the complement set of the $n-k$ independent set – or maximum independent set.

Some related problems are not just about if there is a k-clique present in the graph, but the question is rater the number of these k-cliques. Thus sometimes one needs to enumerate all k-cliques or all maximum cliques as well [Ebl2012].

Another variations of these problems are connected to different types of graphs. There can be weights assigned to nodes – or edges, or both – in the graph. In this case one can search for a maximum (node or edge) weight clique, or a k node clique with the biggest weight, or a clique with the prescribed weight. Also, the graph can be a directed one, and we can search for directed cliques, which in the literature called transitive tournament. One can search for a maximum one or for one with a prescribed size [Kivi2016].

Next related problem is the problem of the quasi cliques [Patt2013b, Abe1999]. Here we search for a big subgraph, but with eased constrain of the subgraph being a complete graph only a dense one. There is no definition of a quasi clique, but with a given definition one can search for a maxium one or one of size k.

Finally, we would like to mention the problem of motif search. It asks us to find some (usually small) subgraph, named motif, present in the given graph [Milo2002, Schl2016]. Obviously, if the motif is a complete graph we get the same problem as the k-clique.

1.2 Motivation and background

In management science or operations research the main task one faced is to solve a real life problem. This is done by the means of mathematical programming, which basically consists of two steps. First step is the modeling of the problem in some well known approach, and the second one is to solve this problem with the aid of specialized solver.

There are numerous ways of modeling, and these usually can be freely interchanged. So the decision of choosing the model is rather backed up by the software tool at hand. The most widely used approach is to use a Linear Programming (LP) toolkit, or its variants according the specialty of the problem like Mixed Integer Linear Programming (MILP) or Integer Linear Programming (ILP), or Zero-One Linear Programming (0–1 LP). For combinatorial optimization or decision problems ILP or 0–1 LP is used, but there are other methods, such as Satisfiability (SAT or MaxSAT) or Constraint Programming (CP). While most times any of them can be used there are efficiency differences. These differences caused by two phenomena. First, some problems are more suitable for one model then the other, and so the software solving the problem may be more efficient. Second, some software is simply more advanced then the other, as more developers work on its perfection. Consequently, for any combinatorial optimization problem the first choice is an ILP formulation, given its versatility and easy to model feature and the very developed software. Note though, that this is not necessarily the most efficient approach, and in the case of a harder problem may lead to failure. And there is another problem, which invovles the reliability of the computations. ILP solvers use LP as an auxiliary algorithm, so rounding errors may affect the result [Aki2016]. If one needs reliable computation she or he needs to choose another solver, which is free of such defects, and of course the clique solvers are such using exclusively integer and bit computations.

The present work aim to widen the possibility of modeling by showing that modeling by graphs and finding a maximum clique or k-clique of given size is a good approach for solving several problems. In the present work we shall show the versatility of this modeling. Also, it is important to express that the current state-of-the-art clique solvers are as well developed as the SAT solvers, CP solvers or perhaps close to ILP solvers as well. This means that there can be some problems for which the clique formulation is more natural, and so the solution using graph modeling is more efficient.

We shall also in detail show that the graph formulation is more suitable for developing a massively parallel algorithm, and so using supercomputers to aid in solution of some hard problems. This task, of efficient parallelization, is usually considered very problematic in combinatorial optimization.

Chapter 2

Modeling expressivity

In this chapter we would like to list some problems related or solvable with k-clique or maximum clique search. First, we enumerate some simple problems and in details show a possible according graph model. Second, we list real life problems connected to subgraph isomorhism, where the solution may be obtained by clique search or some clique search algorithm may prove useful as an auxiliary algorithm. Third, in detail we show how can some complex sheduling problems modeled and solved by k-clique approach. Finally, we point out some possible connections with network analysis.

We need to mention that the proposed methods are not the only possible mathematical models for these problems. Some of these are solved with Integer Programming, SAT solvers, Constraint Programming, specialized backtracking, set cover or graph coloring algorithms. Out aim is solely to demonstrate the possibility of graph modeling and solution using a k-clique algorithm.

For some problems we will include numerical experiments and such results. We do not claim, that these graph models and our k-clique solver would be the best possible solution to solve these problems. Our aim is simply to demonstrate that some non trivial problems can be solved in this way, even if some other methods would be better. But we do claim, that there is a great potential of such models. First, with future extensions like kernelization or symmetry breaking, we think that this approach can sometimes even be better for some few problems. Second, in contrast to ILP formulation, which is the most often used option, our method does not suffer from rounding errors. Of course, ILP solvers can be modified to be also rounding error free – by usage of rational numbers or interval arithmetic –, but then they will be a magnitude slower, and our method would turn out faster.

2.1 Puzzles, games, codes and other combinatorial problems

Modeling with graphs and searching for cliques in these can be utilized for solving various combinatorial problems, puzzles and games. It is well known that different puzzles can be solved in graph theoretical methods [Foul1992]. Here we would like to list some of those, particularly ones where the solution could be obtained by searching for a k-clique in an auxiliary graph. For the extended descriptions of these puzzles see [Krai1953]. For each problem we construct an auxiliary graph G and show that the original question can be answered by finding a maximum or k-clique of given size.

2.1.1 Latin squares

Given an array of $n \times n$ we need to fill it with numbers (or symbols) $1 \dots n$ (or $A \dots Z$), such that each symbol must appear once on each row and column, see Figure 2.1.

D	B	F	E	G	A	C	I	H
H	E	G	B	I	C	A	D	F
A	C	I	D	F	H	B	G	E
I	G	A	C	H	E	F	B	D
E	D	C	G	B	F	H	A	I
F	H	B	A	D	I	G	E	C
G	I	D	F	C	B	E	H	A
B	F	E	H	A	D	I	C	G
C	A	H	I	E	G	D	F	B

Figure 2.1: Example of a 9×9 Latin square

We will show how to solve this problem with the aid of constructing an auxiliary graph and deciding if there is a k-clique present, with a given k.

The auxiliary graph $G = (V, E)$ constructed as follows. The nodes V of graph noted by (x, y, z) triples, $x, y, z \in \{1 \dots n\}$, where x and y represent the coordinate, while z represents the symbol written on that coordinate. V is the list of all possible triplets, that is all possible way to write a symbol at any cell in the array. The edges of the graph will represent agreeable pairs of triples that is when two actions represented by two triples can occur in the same solution. For example triples $(2, 4, D)$ and $(2, 9, D)$ are in conflict, as the symbol D cannot appear in the same row, thus there will be no edge between them. But triples $(2, 4, D)$ and $(2, 9, F)$ are agreeable, thus there will be an edge between them. Also, one cannot write two symbols in the same cell, so triples where $x_1 = x_2, y_1 = y_2$ are not connected.

Formally, two triples (x_1, y_1, z_1) and (x_2, y_2, z_2) connected iff:

1. $x_1 \neq x_2, y_1 \neq y_2$ or,
2. $x_1 = x_2, y_1 \neq y_2, z_1 \neq z_2$ or,
3. $x_1 \neq x_2, y_1 = y_2, z_1 \neq z_2$.

We shall call this type of graph an *agreement graph*[1].

A solution means n^2 number of agreeable triples as one needs to fill in n^2 number of cells. That is if we search for a $k = n^2$ k-clique, then finding one we obtain a solution for our problem.

Let us see a small example of board size 3×3. The tripples are:

- $(1, 1, A), (1, 2, A), (1, 3, A), (2, 1, A), (2, 2, A), (2, 3, A),$
  $(3, 1, A), (3, 2, A), (3, 3, A)$
- $(1, 1, B), (1, 2, B), (1, 3, B), (2, 1, B), (2, 2, B), (2, 3, B),$
  $(3, 1, B), (3, 2, B), (3, 3, B)$
- $(1, 1, C), (1, 2, C), (1, 3, C), (2, 1, C), (2, 2, C), (2, 3, C),$
  $(3, 1, C), (3, 2, C), (3, 3, C)$

For the sake of the example let us see to which nodes the node $(1, 1, A)$ is connected. We list the connecting nodes according the previous enumeration of the rules:

1. $(2, 2, A), (2, 3, A), (3, 2, A), (3, 3, A), (2, 2, B), (2, 3, B), (3, 2, B), (3, 3, B),$
   $(2, 2, C), (2, 3, C), (3, 2, C), (3, 3, C)$
2. $(1, 2, B), (1, 3, B), (1, 2, C), (1, 3, C)$
3. $(2, 1, B), (3, 1, B), (2, 1, C), (3, 1, C)$

We do can find a $(3^2 = 9)$ 9-clique in this graph, for example the node set $\{(1, 1, A), (1, 2, B), (1, 3, C), (2, 1, B), (2, 2, C), (2, 3, A), (3, 1, C), (3, 2, A), (3, 3, B)\}$, which solution is depicted on Figure 2.2.

[1]On the other hand, the graph type, where the edges represent conflicts between the nodes usually called the *conflict graph* in the literature. Note, that they are complement graphs.

A	B	C
B	C	A
C	A	B

Figure 2.2: Example of a 3×3 Latin square

There are several possible modifications to the original Latin square problem. One can add constrains as for example the diagonals are also forbidden to have same symbols. The famous game of sudoku is also a variation. The the constrain of nine 3×3 boxes is added, where no two symbols can appear at the same time. All these problems can be solved by the same method with modified rules on the edge connections.[2]

Most usually if one encounters a puzzle of this sort, the sudoku puzzle is of different appearance. It is set up the following way. Some of the symbols are already placed and one needs to fill in the missing cells, see 2.3.

	2		5		1		9	
8			2		3			6
	3			6			7	
		1				6		
5	4						1	9
		2				7		
	9			3			8	
2			8		4			7
	1		9		7		6	

4	2	6	5	7	1	3	9	8
8	5	7	2	9	3	1	4	6
1	3	9	4	6	8	2	7	5
9	7	1	3	8	5	6	2	4
5	4	3	7	2	6	8	1	9
6	8	2	1	4	9	7	5	3
7	9	4	6	3	2	5	8	1
2	6	5	8	1	4	9	3	7
3	1	8	9	5	7	4	6	2

Figure 2.3: Example of a sudoku puzzle and it's solution

Thus, the original proposed method must be altered to meet this demand. For finding the solution using the auxiliary graph we need first to find the common neighborhood of the triples representing the filled in places. Then one simply needs to find a k-clique in the reduced graph where k is the number of empty places.

[2]There is another method to solve sudoku, which is done by constructing a specific conflict graph as an auxiliary graph and performing a coloring of this graph. The reader can notice that the two methods are connected through the modeling described in the Chapter 3.

2.1.2 Non-attacking queens

A somehow similar problem arises from the game chess. The question is if one can place eight queens on the chessboard such that none of them threaten each other. The problem also can be rephrased for placing n queens on an $n \times n$ chessboard. On Figure 2.4 $n = 5$. Again a similar n^3 graph can be constructed, where the nodes will consist of triples (x, y, z), x and y representing the coordinate and z representing the number of the queen. An agreement graph can be constructed the same way as in the previous example. The nodes should be connected if they represent different queens not in threatening each other. A $k = n$ k-clique will represent a solution.

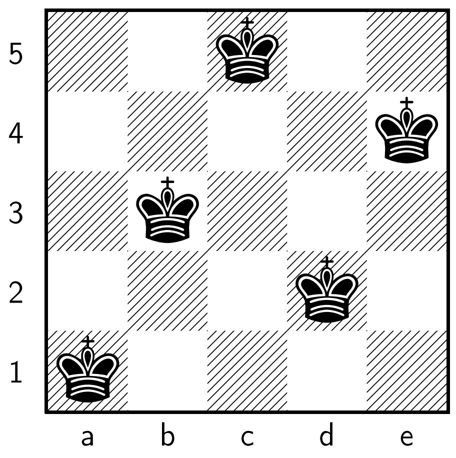


Figure 2.4: Example of a Five Queens problem solution

This previous example uses too many nodes and thus be easily reduced. As there always should be one queen per row we can omit the number of the queen. The nodes so will be pairs (x, y), representing a queen standing at this coordinate. A k-clique will represent k queens on the chessboard not threatening each other.

2.1.3 Monotonic matrices

As an example of modeling a more complex problem we would like to detail the so called Monotonic Matrices problem. From Wolfram Web [Weisst] "A monotonic matrix of order n is an $n \times n$ matrix in which every element is either 0 or contains a number from the set $\{1, \ldots, n\}$ subject to the conditions:

1. The filled-in elements in each row are strictly increasing,
2. The filled-in elements in each column are strictly decreasing, and
3. Positive slope condition: for two filled-in cells with same element, the one further right is in an earlier row."

An example depicted on Figure 2.5, where the possibly maximum 23 cells are filled in.

		2			4	7	8
		1	7	8			
7	8						
	2		4				6
	1				3	6	
4				6			
2				3			5
1			3			5	

Figure 2.5: Example of an $n = 8$ size Monotonic Matrix

The graph reformulation of the problem in question again similar to the previous ones. The nodes of the auxiliary graph are the triples (x, y, z), x and y representing the coordinate and z representing the positive number written in the cell. The edges will consist of agreeable nodes, where the above mentioned properties hold. For details see [Szab2013, Öst2019].

2.1.4 Costas arrays

Finally, we would like to show the expressing power of the graph representation with a special problem. The proposed representation – to our knowledge – is not known in the literature. We propose a graph representation to the Costas array problem [Cost1965, Cost1984]. This problem derives from radar and sonar technology, namely phased array radar engineering. The solution helps generating radar and sonar signals with ideal ambiguity functions. The formalization of the problem as follows. Given an $n \times n$ array one needs to fill it in with n dots. The constrain is that no two dots may lay on the same row or column, and the displacement vectors of any two pairs of dots must be distinct from all the other such displacement vectors. A possible solution for $n = 8$ demonstrated on Figure 2.6.

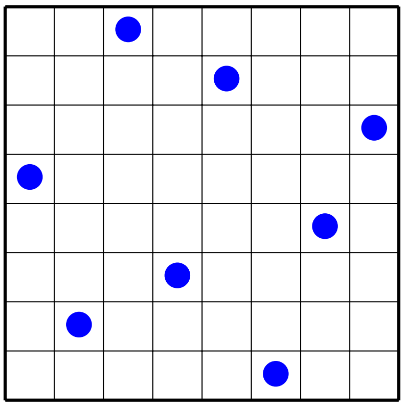


Figure 2.6: Example of an $n = 8$ size Costas array

The graph representation of this problem is more complex then the previous examples. As the constrain of different displacement vectors instruct us about pairs of dots (4 coordinates altogether in one pair), the nodes of the auxiliary graph $G(V, E)$ shall be denoted by quadruples (x_1, y_1, x_2, y_2), where the coordinates of the represented dot-pair are (x_1, y_1) and (x_2, y_2). One could list all possibilities, but clearly only pairs of agreeable nodes needed to be listed, thus the rows and columns of the two dots must be different. That means that we omit for example the quadruple $(1, 1, 1, 3)$, because the two dots lie in the same column. The size of this graph is $n^2(n-1)^2/2$. The edges of the graph represent the agreement between two dot-pairs. We need to take special care of the case when one of the dots of a pair coincides with a dot from the other dot-pair. In this case we have 3 instead of 4 distinct dots.

Formally there should be *no* edge between two nodes (x_1, y_1, x_2, y_2) and (a_1, b_1, a_2, b_2), iff:

1. If two pairs of dots have same row or column;
2. If one pair of dots have same row or column and there are 4 distinct nodes;
3. If one pair of dots have same row or column and there are 3 distinct nodes, and some of the displacement vectors of these three are the same;
4. If no pair of dots have same row or column and there are 4 distinct nodes, and some of the displacement vectors of these four are the same.

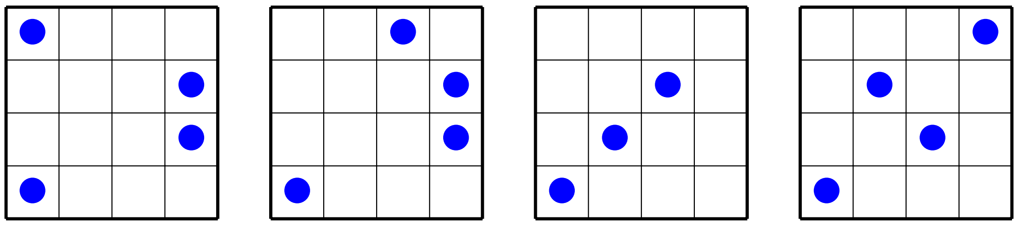


Figure 2.7: Depiction of constraints 1, 2, 3 and 4.

In other cases there will be an edge between these two nodes. A k-clique of size $k = n(n-1)/2$ represents all pairs from n dots, and thus it is a solution to the problem in question.

With the help of this simple formulation – even without using any kernelization techniques – one can find one solution of the non trivial 14×14 array in few seconds and calculate all possible solutions in half an hour.

2.1.5 Communication and coding theory

In theory of communication one important aspect is the construction of codes that are resilient to deletion or flipping errors. We would like to detect or reconstruct the digital information, and for this purpose special codes are constructed. Because of the (always) limited bandwidth one would like such construction be of less overhead, so we would like to find the maximum number of code words under specific conditions. One approach to this task is to list all possible code words – they will be the nodes of the graph – and construct a conflict graph. The maximum independent set is the optimum code. For more see [But2002, But2009, Sloan, Bogd2001, Öst2020]. Note that some of the widely used maximum clique test problems are coming from coding theory, such as the Hamming graphs of Johnson graphs.

There is a special question in coding theory, which asks for the maximum amount of information that can be sent over a noisy channel using multiple signal code words. The limit of this is the so called Shannon capacity. If we represent the confusion between the code signals by a graph, then we can speak about the Shannon capacity of a graph. It is uniquely hard to compute this number for most cases, but we can still calculate some upper bound. One way of doing such calculation is to calculate the size of the independent set in a (finite) sequence of product graphs [Pol2019, Math2017].

2.2 Subgraph isomorphism

After pure mathematical puzzles let us demonstrate the usefulness of graph representation and k-clique search for solving real life problems. First category is the problem class, when the problems can be modeled with induced subgraph isomorphism. Formally, given the graphs $H = (W, F)$ and $G = (V, E)$, there is a subset $V' \subset V$, where the induced graph $G(V')$ isomorphic to H. The problem called as isomorphic embedding problem as H can be isomorphically injected into G. The problem is to decide if two given graphs maintain this property or not. Many different computationally challenging problems with important practical applications fall into this category. For example in computer vision, biochemistry, and model checking. Especially wide usage of this method is found in chemistry where similarities between drug compounds checked and databases built up on that information [Konc2007]. Also similar method can be used for checking protein docking abilities of drugs.

The induced subgraph isomorphism problem modeled with graph by using a modular product of the given two graphs. A k-clique of the size of the smaller graph, $k = |W|$ will give an answer if H is an isomorphic subgraph of G.

2.2.1 Chemistry

The maximum common induced subgraph problem is used in chemistry as a means of comparing shapes of molecules, either as 3D scans or molecular graphs, which represent the structural formula directly [Konc2010, Konc2012, Leš2020]. An example of such use is in prediction of protein function. The characteristic of proteins, which allows them to function within an organism, is their ability to bind other molecules. From the point of physics this can be described as an energy function, where the bond between two atoms means *lower* energy level. But as molecules have (partially) rigid structures, we cannot place any atom to its best place, but all together needed to be placed at once achieving the energy minimum. Thus proteins bind to other molecules similarly as jigsaw puzzle fit together, by matching their shape to the shape of target molecule. The function of unknown protein can therefore be estimated by comparing its shape to shapes of known proteins with known functions.

One of the traditional ways of solving the maximum common subgraph problem is by reduction to a maximum clique problem, using auxiliary product graph. The two input graphs, in which the maximum common subgraph is to be found, are multiplied to form a product graph, which is then input to the maximum clique algorithm. The result of the latter are used to identify the nodes of the input graphs that form the maximum common subgraph. Although the maximum clique problem can be solved by a modern branch-and-bound based algorithm for general graphs, such approach is far from optimal. Some special properties of the product graph can be exploited to guide the maximum clique search. Namely, the modern state-of-the-art clique search programs use coloring as auxiliary algorithm, but finding a good coloring of a graph itself a hard task.

2.2.2 Pattern matching and Artificial Intelligence

Correspondence between atoms in the molecules gives us a straightforward example of subgraph isomorphism. This method can be extended to be used as pattern recognition. For example in images one can set up some points of interests, and making pairs of such point as a correspondence, and using distance similarity between such points a very similar auxiliary graph as in the case of molecules can be built.

As the theory behind comparing fingerprints already works with such point of interests it seems a natural way of using this method for fingerprint matching [Seg2010].

2.3 Job shop scheduling

As we will discuss later in Chapter 3 the graph coloring problem can be transformed into maximum clique or k-clique problem. So any real world problem that can be represented by graph coloring, such as timetables, simple scheduling and assignment problems [Marx2004], can be solved by clique search as well. But the k-clique modeling can be used to solve more complex scheduling problems as well, such as open shop, flow shop or job shop scheduling problems. We should mention, that with some modification the method can be extended to flexible job shop problems as well. In this section we choose the job shop problem as an example to show the expressive power of the k-clique modeling.

The job sequencing problem is an optimization problem [Jain1999]. Certain products are to be produced on given machines satisfying predetermined technological order. The objective is to determine the sequence of jobs in which the various products are processed on the machines in the least possible time. The well-known standard approach recasts the problem by means of a mixed integer linear program. Here we experiment with a more combinatorial idea.

Given a positive number T we constructand an auxiliary graph G and compute an integer k. The graph G encodes the agreements of the job sequencing problem. If the graph G contains a k-clique, then there is a feasible job sequencing whose total completion time is at most T. So, instead of an optimization problem we are dealing with a decision problem.

2.3.1 The clique reformulation of the problem

One has to make a decision which item should be scheduled to which machine at a specified time. So we consider a triplet (u, v, w), where the number u refers to item u. The number v means that item u is assigned to machine v. Finally, the number w tells that the work on item u is started at the time point w. We consider a list of triplets that are relevant to the scheduling at hand. There are pairs of triplets that cannot be part together of any valid schedule. Such conflicts can be recorded by constructing a conflict graph. It turns out that a specified size conflict free set of triplets defines a valid schedule. The graph G we use is actually the complement of the conflict graph and instead of looking for a independent set of size k we are looking for a k-clique.

The nodes of the auxiliary graph G are the triplets relevant to the schedule. That is all possible job, machine and starting time combinations except for those times that cannot occur – a machine starting too soon or too late. Initially we connect all the pairs of distinct nodes by an edge. Next we delete edges that connects conflicting triplets. Specifically we delete an edge if any of the following conditions holds.

1. The machines are processing a given job not in the technologically prescribed order.
2. Distance in time for two machines processing a given job are not sufficient to fit in the prescribed intermediate processing times.
3. Processing periods of the same machine on different jobs overlap.
4. Processing periods of different machines on the same job overlap.
5. Processing of a fixed job on a fixed machine occurs several times.

Set k = (number of items) $\times$ (number of machines). How large clique can appear in the graph G? The answer is that the graph G can contain an k-clique. A Gantt chart which corresponds to a feasible schedule provides an k-clique in G. The reader may note, that our formulation is quite generic. That means that the three different problem classes, the flow shop, the job shop and the open shop problems, all can be solved by this approach with little variations.

2.3.2 A small example

In order to illustrate the previous considerations we work out small size example in details. Three items are to be scheduled on two machines. The work times are summarized in Table 2.1.

Item	Machine 1 work time	Machine 2 work time
1	$d(1,1) = 1$	$d(1,2) = 3$
2	$d(2,1) = 2$	$d(2,2) = 1$
3	$d(3,1) = 3$	$d(3,2) = 1$

Table 2.1: Processing times on the machines in the small example

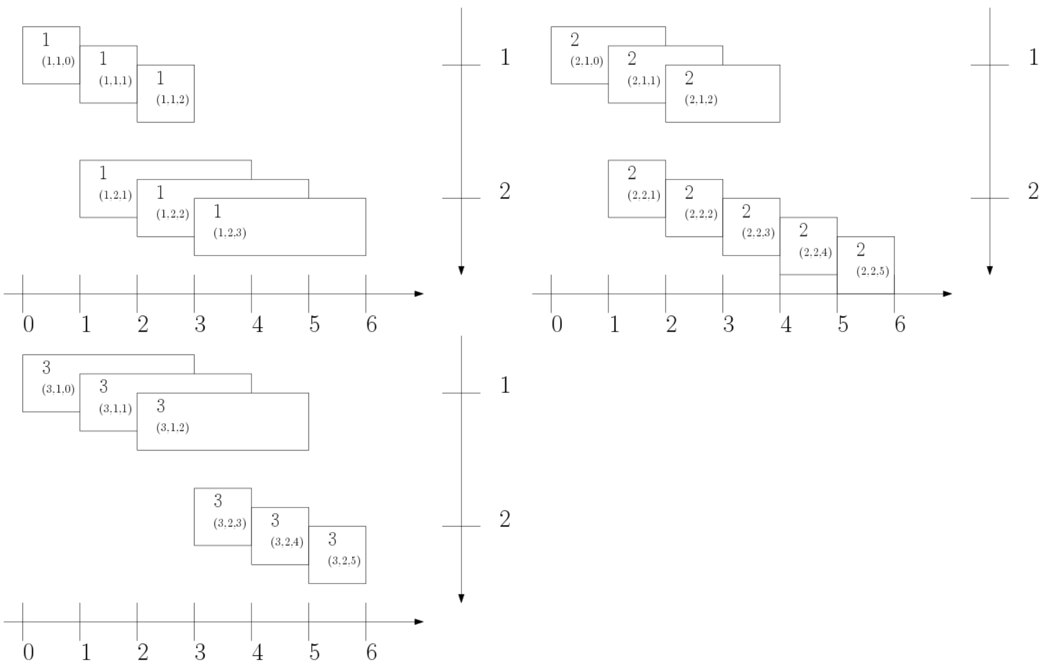


Figure 2.8: Possible time slots (triplets) for the given problem in Table 2.1 with makespan of 6 hours.

Using some greedy heuristics, one can verify that there is a feasible schedule with a completion time 7 hours. We ask if there is a schedule with a completion time 6 hours. Assuming a 6 hours makespan the auxiliary graph G has 20 vertices. The nodes of the graph G are triplets. We numbered the triplets by $1, \ldots, 20$. Table 2.2 lists the triplets together with the corresponding numbers. The adjacency matrix of the graph G is in Table 2.3.

name	triplet	name	triplet	name	triplet	name	triplet
1	(1,1,0)	6	(1,2,3)	11	(2,2,2)	16	(3,1,1)
2	(1,1,1)	7	(2,1,0)	12	(2,2,3)	17	(3,1,2)
3	(1,1,2)	8	(2,1,1)	13	(2,2,4)	18	(3,2,3)
4	(1,2,1)	9	(2,1,2)	14	(2,2,5)	19	(3,2,4)
5	(1,2,2)	10	(2,1,3)	15	(3,1,0)	20	(3,2,5)

Table 2.2: Nodes of the graph G in the small example

The question is if G contains any 6-clique. With exact clique search one can prove that the graph G does not contain any 6-cliques. So the given scheduling problem cannot be completed in $T = 6$ hours makespan, concluding that $T = 7$ is the optimal value.

2.3.3 Numerical experiments

Given the graph reformulation of the flow shop, job shop and open shop problems we are interested in the efficiency of the clique search approach. We considered two large problems, of which the second is still open. As the reader will see our first approach to this problem is not yet as efficient as modern state-of-the-art solvers but can solve moderately hard problems. We used heavy kernelization like detailed Section 4.3, but more sophisticated according to the special features of the auxiliary graph. We do not detail these as they lay outside the scope of the present work.

In fact, for medium problems we could use an exact k-clique solver after the kernelization, as the graph was considerably reduced. For the large instances – apart from some trivial cases –, the reduced graph was still too big for exact solvers. So we could use two heuristic algorithms to set upper and lower bounds. As one can see our approach is stronger for finding lower bounds.

The resulting graphs are very large, so we do not expect that exact clique solvers could solve the k-clique problem. Instead we used two heuristics. First is the DSatur coloring from Brelaz [Brel1979]. If the resulting graph after kernelization can be colored with less then k colors, then there cannot be a k-clique present and thus we can set the lower bound. Second is the KaMIS heuristics [Lam2016] for finding independent sets – we used the complement graph for this purpose. If the resulting graph after kernelization do has a k-clique present this gives us an upper bound.

We tested our algorithm with two known instances from [Ada1988].

The `abz8` instance, which sets up a problem of 20 jobs and 15 machines, has a known lower bound of 648 and upper bound of 665. We could set the

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	×			●	●	●		●	●	●	●	●	●	●		●	●	●	●	●
2		×			●	●			●	●	●	●	●	●			●	●	●	●
3			×			●				●	●	●	●	●				●	●	●
4	●			×			●	●	●	●			●	●	●	●	●		●	●
5	●	●			×		●	●	●	●				●	●	●	●			●
6	●	●	●			×	●	●	●	●					●	●	●			
7				●	●	●	×						●	●				●	●	●
8	●			●	●	●		×					●	●				●	●	●
9	●	●		●	●	●			×				●	●				●	●	●
10	●	●	●	●	●	●				×								●	●	●
11	●	●	●								×				●	●	●	●	●	●
12	●	●	●									×			●	●	●		●	●
13	●	●	●	●			●	●	●				×		●	●	●			●
14	●	●	●	●	●		●	●	●					×	●	●	●			
15				●	●	●					●	●	●	●	×			●	●	●
16	●			●	●	●					●	●	●	●		×			●	●
17	●	●		●	●	●					●	●	●	●			×			●
18	●	●	●				●	●	●	●	●				●			×		
19	●	●	●	●			●	●	●	●	●	●			●	●			×	
20	●	●	●	●	●		●	●	●	●	●	●	●		●	●	●			×

Table 2.3: The adjacency matrix of the graph G in the small example

lower bound by setting the makespan to 599. The auxiliary graph has 66 210 nodes and 2 105 395 622 edges. After kernelization the resulting auxiliary graph has 58 565 nodes and 1 569 659 812 edges, and the DSatur coloring could color the graph with 296 colors, while we are looking for a clique of size 300. This set the lower limit to 600.

For upper limit we used makespan 859. The auxiliary graph has 144 210 nodes and 10 264 886 876 edges. The KaMIS heuristic search found a clique of size 300. This value proved that there is a feasible schedule with the given makespan, leading to an upper bound of 859.

The `abz5` instance, which sets up a problem of 10 jobs and 10 machines, has an optimum of 1234. This problem is solvable by our method to optimality. First we set the makespan to 1233. The auxiliary graph has 45 670 nodes and 956 998 171 edges. After kernelization the resulting graph has 27 072 nodes and 318 810 474 edges, and could be colored by 99 colors with the DSatur algorithm, while we are looking for a clique of size 100. Thus the lower bound is 1234.

With setting the makespan to 1234 the resulting auxiliary graph has 45 770 nodes and 961 249 429 edges. After preconditioning with node dominance the resulting graph has 32 014 nodes and 461 551 291. The KaMIS heuristic search found a clique of size 100, which sets the upper bound to 1234. As the lower and upper bounds are the same we have the optimum solution.

We can conclude from these examples, that the instances are too big for the available exact maximum clique solvers. On the other hand it is possible to find a lower bound by the proposed preconditioning methods and an upper bound by means of approximate methods.

2.4 Network analysis

There is an emerging tool in data science which we call network analysis [Bota2014]. In some of these problems questions about cliques arise [Patt2013a, Chan2014]. Among these problems we find the interaction of people on a telephone network, for which a so called call graph is constructed [Abe1999]. A clique or a quasi clique can hint the operator about being one family or group of friends. One can also look for a terrorist cell in a friendship graph [Kre2002, Hay2006] or find insurance frauds. Marketing information also can be collected into graphs, as for example we can store similar purchases.

The most common way to obtain data from networks is done by community detection, that is clustering. Cliques are used in clique clustering as an auxiliary algorithm [Bota2015]. Also, k-clique search can be useful for preprocessing [Kump2008, Greg2012].

A special usage of networks is the data analysis of stock price for careful portfolio selection. The so called market graph is constructed, where each node represents a stock, and nodes are connected if the price change in values are correlated over a given time period. An independents set or a quasi independent set means that the prices are pairwise independent making those stocks a good portfolio [Bogi2003, Bogi2006, Bogi2014]. Also, one can use the clique number to detect stock market crash. If the maximum clique in the market graph becoming too big – almost all nodes of the graph are in the clique –, then the prices moving in one direction and we should shut down the stock exchange.

An interesting method of modeling the brain – either human or other animals – arise in today's neuroscience. Researchers detect certain regions of the brain (nodes) and find correlations between their work (edges). Graph based analysis done on these so called Brain Graphs, among them clique or quasi clique search [Bull2009]. This method is used for detecting neurological disorders and diseases such as epilepsy [Chi2014] or for characterization of the connectivity of the brain [Rub2009] in brain research.

Chapter 3

Modeling graph and hypergraph coloring with cliques

In the second chapter we presented an extended list of problems that can be modeled and solved through graph representation and searching a k-clique or maximum clique in the corresponding auxiliary graph. In the present chapter we will pick one particular problem class. We shall explore more deeply a case study of different types of graph and hypergraph colorings. In all cases we will show how the problem can be modeled and solved with k-clique search as a versatile method. In the present chapter we use results from [Szab2016b] and [Szab2019b].

In this chapter we first define the legal coloring of the nodes of a graph and show two methods of modeling it by maximum clique and k-clique. Second, we discuss a special graph coloring method of nodes, the 3-free coloring, and its modeling. Third, we conclude with hypergraph coloring problem and models. Finally, with the the aid of proposed modeling method we give a partial answer to an open problem by Voloshin. On usage of graph coloring the reader can find more for example in [Ahm2012, Coop2006].

3.1 Legal coloring of the nodes of a graph

We color the nodes of a graph G satisfying the following conditions.

1. Each node of G receives exactly one color.
2. Adjacent nodes in G cannot receive the same color.

This is the most commonly encountered coloring of the nodes of a graph and it is referred as legal or proper coloring of the nodes.

Problem 3.1. *Given a finite simple graph G find the smallest integer k, such that the nodes of G have a legal coloring using k colors.*

Problem 3.2. *Given a finite simple graph G and given a positive integer k. Decide if the nodes of G have a legal coloring using k colors.*

Problem 3.1 is an optimization problem and belongs to the NP-hard complexity class. Problem 3.2 is a decision problem and belongs to the NP-complete complexity class [Karp1972].

3.1.1 Classical formulation

There is a well known formulation of Problem 3.1, the minimum coloring problem, into an auxiliary graph and a maximum independent set search [Corn2008]. The auxiliary graph Γ has $n^2 + n$ nodes, and it is constructed as follows. We take n copies of the graph G – $G_1, G_2, \ldots, G_n$. We connect all copies of a node pairwise in Γ. We add extra nodes $x_1, x_2, \ldots, x_n$. We connect x_i to all nodes of G_i. A coloring of G will be represented as a union of independent sets in some of the $G_1, G_2, \ldots, G_n$ subgraphs. For each other G_i that has no part of the independent set one can add the node x_i, making a minimal coloring using the maximum number of x_i-s, that is making the independent set maximal. Obviously one can take the complement of the auxiliary graph and instead of searching for the maximum independent set search for the maximum clique, as we will do further.

A newer approach originating from the above was described in [Corn2016]. But we will go to a different direction, because we would like to extend this method to other related problems as well.

3.1.2 The k-clique approach

Both Problems 3.2 – the k coloring problem – and Problem 1.2 – the k-clique problem – are decision problems. From the complexity theory of computations we know that these problems belong to the NP-complete complexity class. Problems 3.2 and 1.2 are polynomially reducible to each other. The point we would like to make here is that reducing Problem 3.2 to Problem 1.2 can be utilized in practical computations.

Here is a way how Problem 3.2 can be reduced to Problem 1.2.

Using the graph $G = (V, E)$ and using the positive integer k one constructs an auxiliary graph $\Gamma = (W, F)$. The nodes of Γ are the ordered pairs

$$(v, a), \quad \text{where} \quad v \in V, \ 1 \le a \le k.$$

The intended meaning of the pair (v, a) is that node v of G receives color a.

Let us pick two distinct nodes

$$w_1 = (v_1, a_1) \quad \text{and} \quad w_2 = (v_2, a_2)$$

of Γ. If the unordered pair $\{v_1, v_2\}$ is an edge of G, then in a legal coloring of the nodes of G the colors a_1, a_2 cannot be identical. When we construct Γ we do not connect the nodes w_1, w_2 if the unordered pair $\{v_1, v_2\}$ is an edge of G and if in addition $a_1 = a_2$ holds. In a coloring of the nodes of G a node cannot receive two distinct colors. Thus when we construct Γ we do not connect w_1, w_2 by an edge in Γ if $v_1 = v_2$.

Let n be the number of vertices of G, that is, let $n = |V|$. The graph Γ has nk vertices.

Observation 3.1. *If the nodes of the graph G have a legal coloring using k colors, then the graph Γ contains a n-clique.*

Proof. Let us assume that the nodes of G can be colored legally using k colors. Let $f : V \to \{1, \ldots, k\}$ be a function which describes this coloring. Let

$$D = \{(v, f(v)) : \ v \in V\}$$

and let Δ be the subgraph of Γ induced by D. Clearly, D has n elements. We claim that Δ is an n-clique in Γ.

In order to verify this claim we pick two distinct nodes

$$w_1 = (v_1, f(v_1)) \quad \text{and} \quad w_2 = (v_2, f(v_2))$$

from D.

If $v_1 = v_2$, then $f(v_1) = f(v_2)$ must hold as the node v_1 receives exactly one color. This means that $w_1 = w_2$. But we know that $w_1 \neq w_2$.

If $v_1 \neq v_2$ and the unordered pair $\{v_1, v_2\}$ is an edge of G, then $f(v_1) \neq f(v_2)$ holds since the coloring defined by f is legal. This means that we have connected the nodes w_1, w_2 by an edge in Γ when we constructed Γ.

If $v_1 \neq v_2$ and the unordered pair $\{v_1, v_2\}$ is not an edge of G, then we have connected the nodes w_1, w_2 by an edge in Γ when we have constructed Γ. □

Observation 3.2. *If the graph Γ contains an n-clique, then the nodes of the graph G can be colored legally using k colors.*

Proof. Let us suppose that Γ has an n-clique Δ and D is the set of nodes of Δ. Let

$$I_v = \{(v, a) : \ 1 \leq a \leq k\}$$

for each $v \in V$. Obviously, I_v has k elements. Note that the sets I_v, $v \in V$ are pair-wise disjoint independent sets in Γ. Further note that the union of these sets is equal to W.

The nodes of Γ can be colored legally using n colors. The sets I_v, $v \in V$ can play the roles of the color classes of the nodes of Γ. Since Δ is a clique in Γ it follows that each I_v contains at most one element from D. Using the fact that $|D| = n$ we can conclude that D is a complete set of representatives of the sets I_v, $v \in V$.

Set

$$T = \{v : \ (v, a) \in D\}$$

We can see that $T = V$. Therefore each $v \in V$ receives exactly one color. We may express this result such that the map $f : V \to \{1, \ldots, k\}$ defined by $f(v) = a$ is a function. It remains to show that the function f describes a legal coloring of the nodes of G.

Suppose that the unordered pair $\{v_1, v_2\}$ is an edge of G. and consider two distinct nodes

$$w_1 = (v_1, f(v_1)) \quad \text{and} \quad w_2 = (v_2, f(v_2))$$

of Δ. When we constructed the graph Γ we have connected the nodes w_1, w_2 by an edge in Γ because $f(v_1) \neq f(v_2)$. □

Theorem 3.1. *Given a graph G on n nodes and an integer k and a auxiliary graph Γ described above. There is a legal k coloring of nodes of G iff there is an n-clique in Γ.*

Proof. Follows from Observation 3.1 and 3.2. □

Note, that the construction described here is quite similar to the classical formulation for maximum independent set.

3.2 3-clique free coloring

As we stated above we would like to extend this modeling method to other related problems. Here we shall show this for 3-clique free coloring problem described in [Szab2012] and detailed in Subsection 4.2.3.

We color the nodes of a simple, finite graph G satisfying the following conditions.

1. Each node of G receives exactly one color.

2. The three nodes of a 3-clique in G cannot receive the same color.

We call this type of coloring of the nodes of G a 3-clique free coloring. Coloring can be used for estimating clique size.

Let us suppose that Δ is an l-clique in G and let us suppose that the nodes of G have a 3-clique free coloring with k colors. Then $l \leq 2k$ holds.

We indicate the proof in the case when l is an even number. A 3-clique free coloring of the nodes of G gives a 3-clique free coloring of the nodes of Δ. Note that in a 3-clique free coloring of the nodes of Δ at least $l/2$ colors must occur. This gives $l/2 \leq k$, as required.

Problem 3.3. *Given a finite simple graph G and given a positive integer k. Decide if the nodes of G have a 3-clique free coloring using k colors.*

Problem 3.3 can be reduced to Problem 1.2. Starting with the the graph $G = (V, E)$ and the positive integer k we construct an auxiliary graph $\Gamma = (W, F)$. The nodes of Γ are the triples

$$(\{u, v\}, a, b), \quad \text{where} \quad \{u, v\} \in E, \ 1 \leq a, b, \leq k.$$

Let m be the number of edges of G, that is, let $m = |E|$. The number of the triples is equal to mk^2.

The triple $(\{u, v\}, a, b)$ intends to code the information that the end points u, v of the edge $\{u, v\}$ are colored with the colors a, b respectively. In this section we assume that each node of the graph G is end point of some edge of G. In other words we assume that the graph G does not contain isolated nodes.

Let us consider two distinct nodes

$$w_1 = (\{u_1, v_1\}, a_1, b_1) \quad \text{and} \quad w_2 = (\{u_2, v_2\}, a_2, b_2)$$

of Γ. Set

$$X = \{u_1, v_1\} \cup \{u_2, v_2\} = \{u_1, v_1, u_2, v_2\}.$$

It is clear that $|X| \leq 4$ and since $u_1 \neq v_1$ we get that $|X| \geq 2$. Thus $2 \leq |X| \leq 4$. Let H_X be the subgraph of G induced by X. The nodes u_1, v_1, u_2, v_2 receive the colors a_1, b_1, a_2, b_2, respectively in the graph H_X.

When $|X| \leq 3$, then these nodes are not pair-wise distinct and it may happen that two distinct colors are assigned to a node in H_X. In this case we call the graph H_X a non-qualifying graph.

It also may happen that there is a 3-clique in H_X and all the three nodes of this 3-clique receive the same color. In this situation again we call the graph H_X a non-qualifying graph. In all the other cases H_X is called a qualifying graph.

When we construct the graph Γ we connect the nodes w_1, w_2 by an edge in Γ if H_X is a qualifying graph.

Observation 3.3. *If the nodes of G have a 3-clique free coloring with k colors, then the graph Γ contains an m-clique.*

Proof. Suppose that the nodes of the graph G have a 3-clique free coloring using k colors. Let $f : V \to \{1, \dots, k\}$ be a function that codes this coloring. Set

$$D = \{(\{u, v\}, f(u), f(v)) : \ \{u, v\} \in E\}$$

and let Δ be the subgraph of Γ induced by D. It is clear that $|D| = m$. We claim that Δ is a clique in Γ.

In order to verify the claim let us choose two distinct nodes w_1, w_2 from D. Let us consider the subgraph H_X associated with w_1, w_2. Since f is a function, each node of H_X receives exactly one color. As f describes a 3-clique free coloring of the nodes of G, it follows that the restriction of f to the nodes of H_X is a 3-clique free coloring of the nodes of H_X. Thus H_X is a qualifying graph. Consequently, we connected w_1, w_2 by an edge in Γ when we constructed Γ. □

Observation 3.4. *If the auxiliary graph Γ contains an m-clique, then the nodes of the graph G have a 3-clique free coloring with k colors.*

Proof. Suppose that Γ contains an m-clique Δ and D is the set of nodes of Δ. Now $|D| = m$.

Set

$$I_{\{u,v\}} = \{(\{u, v\}, a, b) : \ 1 \leq a, b, \leq k\}$$

for each $\{u, v\} \in E$. Obviously, $|I_{\{u,v\}}| = k^2$. Note that the sets $I_{\{u,v\}}$, $\{u, v\} \in E$ are pair-wise disjoint independent sets in Γ.

Indeed, if

$$w_1 = (\{u, v\}, a_1, b_1) \quad \text{and} \quad w_2 = (\{u, v\}, a_2, b_2)$$

are distinct elements of $I_{\{u,v\}}$, then the graph H_X associated with w_1, w_2 has two nodes. From $w_1 \neq w_2$ it follows that $a_1 = a_2$, $b_1 = b_2$ cannot hold. Thus H_X is not qualifying. This means when we constructed Γ we did not connect w_1, w_2 by an edge in Γ.

The nodes of Γ have a legal coloring using m colors. The independent sets $I_{\{u,v\}}$, $\{u, v\} \in E$ can play the roles of the color classes.

As Δ is a clique in Γ each color class contains at most one element from D. Using the cardinality of D we can conclude that D is a complete set of representatives of the color classes.

Set

$$T = \{\{u, v\} : \ (\{u, v\}, a, b) \in D\}.$$

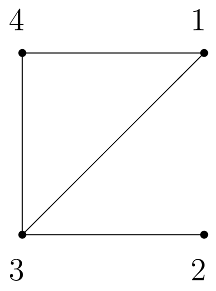


Figure 3.1: A graphical representation of the graph G in Example 3.1.

It follows that $E = T$. Consequently, each node of G which is an end point of at least one edge of G receives at least one color. We claim that each node receives exactly one color.

In order to prove the claim assume on the contrary that more than one colors are assigned to a node of G. In this case there are distinct nodes w_1, w_2 of Δ such that a node receives more than one color in the subgraph H_X associated with w_1, w_2. This means that H_X is not qualifying. On the other hand when we constructed Γ we connected w_1, w_2 by an edge on the base that the subgraph H_X was qualifying.

We may summarize our consideration by saying that we can define a function $f : V \to \{1, \ldots, k\}$ by setting $f(u) = b$ whenever $(\{u, v\}, a, b)$ is a node of Δ. It remains to show that the coloring of the nodes of G described by the function f is a 3-clique free coloring.

Suppose there is a 3-clique Ω in G whose nodes receive the same color. There are distinct nodes w_1, w_2 of Δ such that Ω is a 3-clique in the subgraph H_X associated with w_1, w_2. This means that H_X is not qualifying. On the other hand when we constructed Γ we connected w_1, w_2 by an edge in Γ because the subgraph H_X was qualifying. □

Theorem 3.2. *Given a graph G with m edges and an integer k and a auxiliary graph Γ described above. There is a 3-free coloring of nodes of G with k colors iff there is an m-clique in Γ.*

Proof. Follows from Observation 3.3 and 3.4. □

Example 3.1. *Let the finite simple graph $G = (V, E)$ be given by its adjacency matrix in Table 3.2. The graph has 4 nodes and 4 edges. Figure 3.1 depicts a possible geometric version of G .*

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	×					●			●	●				●		
2		×			●	●			●	●					●	●
3			×				●	●			●	●	●	●		
4				×			●				●	●			●	
5		●			×					●		●			●	
6	●	●				×			●	●	●	●		●		●
7			●	●			×		●	●	●	●	●		●	
8			●					×	●		●			●		
9	●	●				●	●	●	×				●	●		
10	●	●			●	●	●			×					●	●
11			●	●		●	●	●			×				●	●
12			●	●	●	●	●					×	●	●		
13			●				●		●			●	×			
14	●		●			●		●	●			●		×		
15		●		●	●		●			●	●				×	
16		●				●				●	●					×

Table 3.1: The adjacency matrix of the auxiliary graph Γ in Example 3.1.

	1	2	3	4
1	×		●	●
2		×	●	
3	●	●	×	●
4	●		●	×

Table 3.2: The adjacency matrix of the graph G in Example 3.1

1	$(\{1,3\},1,1)$	9	$(\{2,3\},1,1)$
2	$(\{1,3\},1,2)$	10	$(\{2,3\},1,2)$
3	$(\{1,3\},2,1)$	11	$(\{2,3\},2,1)$
4	$(\{1,3\},2,2)$	12	$(\{2,3\},2,2)$
5	$(\{1,4\},1,1)$	13	$(\{3,4\},1,1)$
6	$(\{1,4\},1,2)$	14	$(\{3,4\},1,2)$
7	$(\{1,4\},2,1)$	15	$(\{3,4\},2,1)$
8	$(\{1,4\},2,2)$	16	$(\{3,4\},2,2)$

Table 3.3: The nodes of the auxiliary graph Γ in Example 3.1.

We wish to decide if the nodes of the graph G have a 3-clique free legal coloring with 2 colors. By constructing the auxiliary graph $\Gamma = (W, F)$ the question is reduced to a clique search. The graph Γ has $|V| \cdot k^2 = (4)(2^2) = 16$ nodes. The nodes of Γ are listed in Table 3.3.

3.3 Reducing hypergraph coloring to clique search

Our final example for using this technique will be the coloring of hypergraphs. Namely, we will show how legal coloring of the nodes of a hypergraph can be reduced to clique search in a uniform hypergraph. (More on hypergraphs see [Berg1973, Bret2013].)

Let $H = (V, E)$ be a finite simple hypergraph. The hypergraph has finitely many nodes and finitely many edges. Further it does not have any loop (hyperedge containing only one element) and it does not have double hyperedges.

We color the nodes of the hypergraph H in the following way.

1. Each node receives exactly one color.

2. All the nodes of a hyperedge cannot receive the same color.

This type of coloring of the nodes of the hypergraph is called a legal coloring of the nodes of H. A coloring of the nodes of the hypergraph $H = (V, E)$ can be conveniently given by a map $f : V \to \{1, \ldots, k\}$. Here the numbers $1, \ldots, k$ represent the colors and $f(v)$ is the color of the node $v \in V$. The i-th level set of the function f is commonly referred to as the i-th colors class. The i-th color class C_i is equal to $\{v : \ v \in V, \ f(v) = i\}$. A coloring of the nodes can also be given by the colors classes $C_1, \ldots, C_k$.

The next problem is known as the k-colorability problem

Problem 3.4. *Given a finite simple hypergraph $H = (V, E)$ and given a positive integer k. Let us decide if the nodes of H can be legally colored using k colors.*

For each finite simple hypergraph H there is a well defined positive integer k such that the nodes of H can be legally colored using k colors but the nodes of H cannot be legally colored using $k - 1$ colors. This k is called the chromatic number of H and is denoted by $\chi(H)$.

By the complexity theory of algorithms, Problem 3.4 belongs to the NP-complete complexity class even in the $k = 2$ special case (see [Gare2003, Papa1994].) One may interpret this fact by saying that deciding if the nodes of a given hypergraph can be legally colored using two colors is a computationally demanding problem. Consequently determining the chromatic number of a given hypergraph is a computationally hard problem as well.

There are other types of hypergraph colorings, namely rainbow coloring, mixed coloring, etc. We will describe some in the text, but actually, although they are truly different constructions, from the point of view of our construction there is little difference between them.

A subset I of the nodes of the hypergraph H is called an independent set if a subset of I is never a hyperedge of H. An independent set I is maximal in H if it cannot be extended to a larger independent set by augmenting it by a node of H. An independent set I with cardinality k is a maximum independent set in H if H does not contain any independent set with cardinality $k+1$.

Legally coloring the nodes of an ordinary graph or a hypergraph has many important applications in various fields besides its theoretical significance. Since finding the optimal number of colors of a legal coloring can easily exceed the available computational resources in many practical situation we settle for approximate greedy coloring procedures. In this work we will reduce hypergraph coloring problems to hyperclique search problems in r-uniform hypergraph. We intend to exploit the many possible greedy clique locating procedures to construct approximate legal coloring of the nodes of a given hypergraph.

Let $H = (V, E)$ be a finite simple r-uniform hypergraph. It means that H is a finite simple hypergraph such that each edge contains exactly r nodes. Let C be a subset of V. We say that C is a clique in H if each r pair-wise distinct nodes in C are the nodes of a hyperedge of H. The size of the clique is the cardinality $|C|$ of C. If $|C| = k$ we speak of a k-clique.

The next problem is the so-called k-clique problem for hypergraphs.

Problem 3.5. *Given a finite simple r-uniform hypergraph H and given a positive integer k. Decide if H contains a k-clique.*

For a given finite simple r-uniform hypergraph H there is a well defined positive integer k such that H has a hyperclique of size k and H does not have any hyperclique of size $k+1$. This k is called the clique number of H and is denoted by $\omega(H)$. It is a well-known result from complexity theory that the k-clique problem is in the NP-complete complexity class even in the $r = 2$ particular case (see [Gare2003, Papa1994].) As the k-clique problem is computationally challenging it must hold for the problem of determining the clique number too.

hyperedge	tiles
$\{1,3,6\}$	$\{1,3\},\{6\}$
$\{1,3,8\}$	$\{1,3\},\{8\}$
$\{1,5,8\}$	$\{1,5\},\{8\}$
$\{2,4,5,7\}$	$\{2,4\},\{5,7\}$

Table 3.4: The tiles assigned to the hyperedges in Example 3.2. The hyperedges are cut into two tiles.

To an r-uniform hypergraph H it is customary to assign an r-uniform hypergraph H' such that the nodes of H' are the same as the nodes of H and an r element subset e of the nodes is a hyperedge of H' when e is not a hyperedge of H. The graph H' is called the complement of H. Note that the nodes of hyperclique in H form an independent set in H' and the elements of an independent set in H are the nodes of a hyperclique in H'. We can speak of maximal and maximum cliques in the same way as we spoke about maximal and maximum independent sets.

3.3.1 Reducing hypergraph problems to ordinary graph problems

Let $H = (V, E)$ be a finite simple hypergraph. We assign colors to the nodes of H such that the following two conditions are met.

1. Each node receives exactly one color.
2. Two distinct nodes of a hyperedge never receive the same color.

This type of coloring of the nodes of the hypergraph is called a rainbow coloring of the nodes of H. The following problem can be called as the k-rainbow colorability problem

Problem 3.6. *Given a finite simple hypergraph $H = (V, E)$ and given a positive integer k. Let us decide if the nodes of H can be rainbow colored using k colors.*

One can observe that Problem 3.6 is not a genuine hypergraph problem in the sense that it can be reduced to the coloring of the nodes of an ordinary graph. Let us define an ordinary graph G. The nodes of G are identical to the nodes of the hypergraph H. Two distinct nodes u, v of G will be adjacent in G if u, v are elements of a hyperedge of H simultaneously. It is easy to verify that if the nodes of the hypergraph H have a rainbow coloring with k colors then the nodes of the ordinary graph G have a legal coloring using k colors. And conversely, if the nodes of G have a legal coloring with k colors, then the nodes of H have a rainbow coloring with k colors.

One may define cliques in a hypergraph $H = (V, E)$ in the next way. A subset C of V is a clique in H if for each distinct nodes u, v in V there is a hyperedge of H that contains both u and v. We may consider Problem 3.5 in connection with this clique concept. Locating a k-clique in the hypergraph H can be reduced to find a k-clique in an ordinary graph G. For this purpose it is enough to introduce the ordinary graph G we have described above. In this sense this new clique concept is not a genuine generalization of the ordinary clique concept for hypergraphs.

Let $H = (V, E)$ be a 3-uniform hypergraph and suppose that we are looking for a k-hyperclique in H. This problem can be reduced to a clique search in an ordinary graph G. Let $e_1, \ldots, e_m$ be all the hyperedges of H. These hyperedges will be the nodes of G. Two distinct edges $e_i = \{u_i, v_i, w_i\}$, $e_j = \{u_j, v_j, w_j\}$ are adjacent in G if the unordered triplets

$$\begin{array}{lll} \{u_i, u_j, v_j\}, & \{u_i, u_j, w_j\}, & \{u_i, v_j, w_j\}, \\ \{v_i, u_j, v_j\}, & \{v_i, u_j, w_j\}, & \{v_i, v_j, w_j\}, \\ \{w_i, u_j, v_j\}, & \{w_i, u_j, w_j\}, & \{w_i, v_j, w_j\} \end{array}$$

are all hyperedges of the hypergraph H. Both of the set $\{u_i, v_i, w_i\}$, $\{u_i, v_i, w_i\}$ has three elements. It can happen that these sets are not disjoint. In this case not all of the listed nine sets have three elements. We should check if the three elements subsets among the listed nine sets are hyperedges of the hypergraph H.

We claim that if the hypergraph H has a hyperclique of size k, then the ordinary auxiliary graph G has a clique of size $\binom{k}{3}$.

In order to prove the claim let $C \subseteq V$ with $|C| = k$ such that for each pair-wise distinct $u, v, w \in C$ the unordered triplet $\{u, v, w\}$ is a hyperedge of H. We can form $\binom{k}{3}$ unordered triples from the elements of C. All these triplets are hyperedges of H. Further these hyperedges are pair-wise adjacent nodes in the graph G. Thus G has a clique of size $\binom{k}{3}$.

Next we claim that if G has a clique of size $\binom{k}{3}$, then H has a hyperclique of size k. Let $m = \binom{k}{3}$ and let $e_1 = \{u_1, v_1, w_1\}, \ldots, e_m = \{u_m, v_m, w_m\}$ be all the nodes of a clique of size m in G. Of course $e_1, \ldots, e_m$ are hyperedges of the hypergraph H. Set $C = \{u_1, v_1, w_1, \ldots, u_m, v_m, w_m\}$. There maybe repetition among the elements $u_1, v_1, w_1, \ldots, u_m, v_m, w_m$. In other words these elements are not necessarily pair-wise distinct. Let us suppose that $|C| = t$. As $e_1, \ldots, e_m$ are pair-wise distinct three element subsets of C, it follows that $m = \binom{k}{3} \leq \binom{t}{3}$ and so $k \leq t$.

Choose $u, v, w \in C$ such that u, v, w are pair-wise distinct. By the definition of C there are hyperedges e_p, e_q, e_r of H for which $u \in e_p$, $v \in e_q$, $w \in e_r$ and $e_p, e_q, e_r \subseteq C$. Note that e_p, e_q are adjacent nodes in G. Using the nine subsets in the definition of the adjacency in G we get that there is a hyperedge e_s of H such that $u, v \in e_s$ and $e_s \subseteq C$. Note that e_r, e_s are adjacent nodes in G. We get that there is a hyperedge of H that contains u, v, w. Therefore each three element subset of C is a hyperedge of H. This means that H has a hyperclique of size k.

3.3.2 The auxiliary hypergraph

We pick a hyperedge e of the hypergraph H. We partition e into the subsets $T(e,1), \ldots, T(e,r)$. In other words we choose the subsets $T(e,1), \ldots, T(e,r)$ such that they satisfy the following conditions.

1. $T(e,i) \neq \emptyset$ for each i, $1 \leq i \leq r$.
2. $T(e,1) \cup \cdots \cup T(e,r) = e$.
3. $T(e,i) \cap T(e,j) = \emptyset$ for each i, j, $1 \leq i < j \leq r$.

We will refer to the subsets $T(e,1), \ldots, T(e,r)$ as tiles associated with the hyperedge e.

We pick a tile $T(e,i)$ and color its elements with the k colors in all possible ways. If $T(e,i)$ has t elements, then the number of possible colorings is equal to k^t. We will denote this number by $\alpha(e,i)$. We will denote a colored tile by $[T(e,i), C(e,i,j)]$. Here $C(e,i,j)$ is a coloring of the elements of $T(e,i)$, that is, $C(e,i,j)$ is a map from $T(e,i)$ to the set of colors $\{1, \ldots, k\}$.

We define an ordinary graph Γ_1. The nodes of Γ_1 are the colored tiles we have just constructed. Two distinct colored tiles

$$[T(e_1,i_1), C(e_1,i_1,j_1)], \qquad [T(e_2,i_2), C(e_2,i_2,j_2)]$$

will be adjacent in Γ_1 if the colorings $C(e_1,i_1,j_1)$, $C(e_2,i_2,j_2)$ do not agree on the intersection of the tiles $T(e_1,i_1)$, $T(e_2,i_2)$.

Next we define an r-uniform hypergraph Γ_2. The nodes of Γ_2 are the colored tiles. The pair-wise distinct colored tiles

$$[T(e,1), C(e,1,j_1)], \ldots, [T(e,r), C(e,r,j_r)]$$

form a hyperedge of Γ_2 if all the nodes of the hyperedge e of H receive the same color at the colorings $C(e,1,j_1), \ldots, C(e,r,j_r)$ of the tiles. We call Γ_1, Γ_2 conflict graphs. Both represent situations that obstruct legal coloring of the nodes of the hypergraph H. As it turns out the information contained by the conflict graphs Γ_1, Γ_2 is sufficient to locate legal coloring of the nodes of the hypergraph H. We will state the results formally in two lemmas. Suppose the given hypergraph H has m hyperedges. The m hyperedges are partitioned into mr tiles.

Lemma 3.1. *If the nodes of the hypergraph H can be legally colored using k colors, then the conflict graphs Γ_1, Γ_2 contain an independent set I of size mr simultaneously.*

Proof. Let us assume that the nodes of the hypergraph H are legally colored using k colors and suppose that the map $f : V \to \{1, \dots, k\}$ defines this coloring. Note that the colored tiles

$$[T(e,i), C(e,i,j)], \quad 1 \leq j \leq \alpha(e,i)$$

that are all the colored versions of the tile $T(e,i)$ are pair-wise adjacent in the conflict graph Γ_1. This means that only one of them can be an element of an independent set in Γ_1. It follows that an independent set in Γ_1 can have at most mr elements.

The map f restricted to the tile $T(e,i)$ provides a colored tile $[T(e,i), C(e,i,j)]$ for some j, $1 \leq j \leq \alpha(e,i)$. There are m choices for e and there are r choices for i. Therefore Γ_1 has an independent set I of size mr.

The colored tiles that form a hyperedge of the conflict graph Γ_2 are all associated one fixed hyperedge e of H. Further all these tiles are colored with one fixed color. But the map f cannot assign the same color to each node of e. This shows that the set I is an independent set in the conflict graph Γ_2. □

Lemma 3.2. *If the conflict graphs Γ_1, Γ_2 contain an independent set I of size mr simultaneously, then the nodes of the hypergraph H can be legally colored using k colors.*

Proof. Let us assume that the conflict graphs Γ_1, Γ_2 have an independent set I of size mr simultaneously. As in the previous proof note that the colored tiles

$$[T(e,i), C(e,i,j)], \quad 1 \leq j \leq \alpha(e,i)$$

are pair-wise adjacent in Γ_1. It follows that exactly one of these colored tiles must be an element of I. This means that each tile is colored, that is, no tile remains uncolored. Consequently, each node of the hypergraph H receives a color. The conflict graph Γ_1 guarantees that a node can receive only one color. The conflict graph Γ_2 makes sure that all the nodes of a hyperedge of H cannot receive the same color. □

	1	2	3	4	5	6	7	8
e_1	●		●			●		
e_2	●		●					●
e_3	●				●			●
e_4		●		●	●		●	

Table 3.5: The incidence matrix of the hypergraph H in Example 3.2.

Let W be the set of all colored tiles. The edges of the conflict graph Γ_1 are two element subsets of W. The hyperedges of the conflict graph Γ_2 are r element subsets of W. We add $r-2$ new nodes $\beta_1, \dots, \beta_{r-2}$ to W to get W'. We construct a new conflict graph $\Gamma = (W', F)$. If the unordered pair $\{u, v\}$ is an edge of Γ_1, then we add the hyperedge $\{u, v, \beta_1, \dots, \beta_{r-2}\}$ to Γ. We add the hyperedges of Γ_2 to Γ without any modification. The conflict graph Γ carries exactly the same information as the conflict graphs Γ_1, Γ_2. In order to find a legal coloring of the nodes of H we should locate an independent set I of size $mr + r - 2$ in the conflict graph Γ. Or equivalently we should look for a hyperclique of size $mr + r - 2$ in the complement of the conflict graph Γ.

3.3.3 Examples

Let us consider the hypergraph $H = (V, E)$ with $V = \{1, 2, \dots, 8\}$ and $E = \{e_1, \dots, e_4\}$, where

$$e_1 = \{1,3,6\}, \quad e_2 = \{1,3,8\},$$
$$e_3 = \{1,5,8\}, \quad e_4 = \{2,4,5,7\}.$$

The hypergraph H has 8 nodes and 4 hyperedges. We ask if the nodes of H can be colored legally using two colors. The incidence matrix of the edges of H is in Table 3.5.

Example 3.2. *Using the hypergraph H we construct an auxiliary hypergraph $G = (W, F)$. In this example we cut the hyperedges of H into two tiles. In other words we choose the number r in the construction to be* 2.

Using the hyperedges of the hypergraph $H = (V, E)$ we construct certain subsets of V. As in Section 3.3.2 we will call the family of these subsets tiles. The list of pair-wise distinct tiles is the following

$$T_1 = \{6\}, \quad T_2 = \{8\}, \quad T_3 = \{1,3\},$$
$$T_4 = \{1,5\}, \quad T_5 = \{2,4\}, \quad T_6 = \{5,7\}.$$

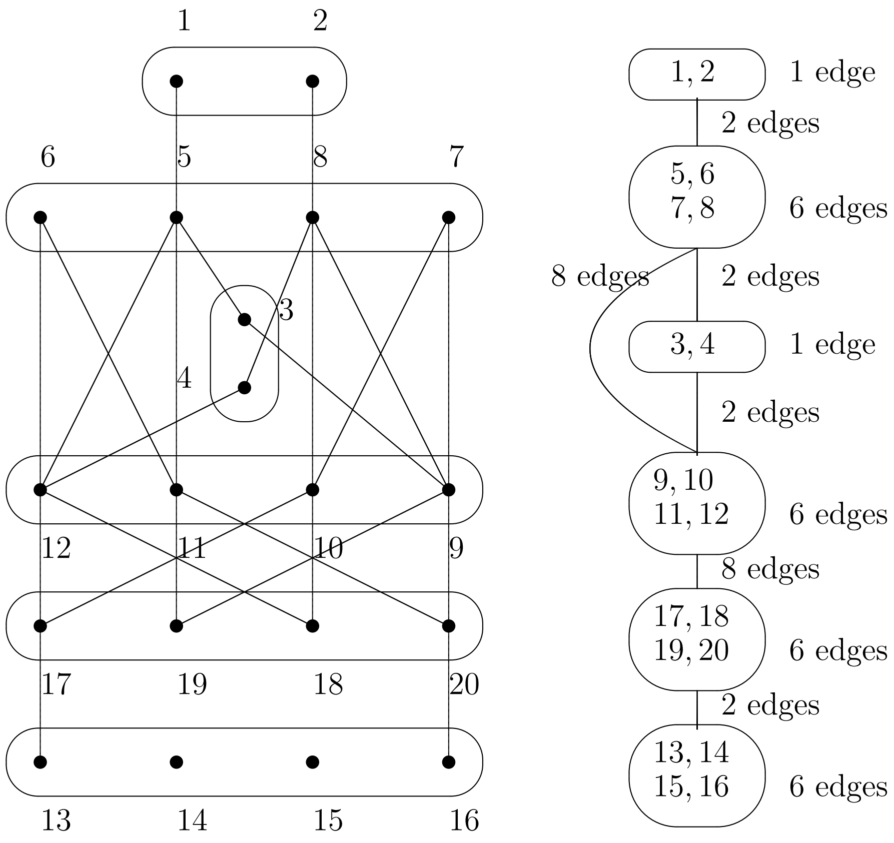


Figure 3.2: On the left is the conflict graph G in Example 3.2. Each distinct two among the elements of $\{5, 6, 7, 8\}$ are adjacent. the same holds for the sets $\{9, 10, 11, 12\}$, $\{13, 14, 15, 16\}$, $\{17, 18, 19, 20\}$, $\{1, 2\}$, $\{3, 4\}$. In order to avoid an overly cluttered picture these edges are not drawn. On the right is a condensed form of the conflict graph. The nodes inside each ovals are pairwise adjacent. An edge between ovals represents several edges. The number of the edges are given near to the ovals and near to the edges.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	×	●			●															
2	●	×						●												
3			×	●	●				●											
4			●	×				●			●									
5	●		●		×	●	●	●			●	●								
6					●	×	●	●			●	●								
7					●	●	×	●	●	●										
8		●		●	●	●	●	×	●	●										
9			●				●	●	×	●	●	●							●	●
10							●	●	●	×	●	●					●	●		
11				●	●	●			●	●	×	●							●	●
12					●	●			●	●	●	×					●	●		
13													×	●	●	●	●			
14													●	×	●	●				
15													●	●	×	●				
16													●	●	●	×				●
17										●		●	●				×	●	●	●
18										●		●					●	×	●	●
19									●		●						●	●	×	●
20									●		●					●	●	●	●	×

Table 3.6: The adjacency matrix of the conflict graph in Example 3.2.

1	5	
2	8	
3	5	9
4	8	12
5	11	12
6	11	12
7	9	10
8	9	10
9	19	20
10	17	18
11	19	20
12	17	18
13	17	
14		
15		
16	20	
17		
18		
19		
20		

Table 3.7: The edges of the conflict graph in Example 3.2. The 9-th row of the table codes the information that the unordered pairs $\{9,19\}$ and $\{9,20\}$ are edges of the conflict graph G.

The way we constructed the tiles is summarized in Table 3.4. There are many ways to divide the hyperedges of H into two tiles. Any of these can be used to construct an auxiliary graph. These auxiliary graphs need not to have the same number of nodes.

After the list of tiles is available we construct a list of colored tiles by assigning colors to the nodes in the tiles in all possible ways. If a tile has n nodes and we try to color the nodes of the hypergraph H using k colors, then from the uncolored tile we will construct k^n colored tiles. The colored tiles are the nodes of the auxiliary hypergraph G.

There is a conflict in the following cases.

1. Two tiles are not disjoint and the common part of the tiles is not colored in the same way in the two tiles.
2. The union of two tiles is equal to a hyperedge of the hypergraph H and all the nodes in the two tiles are receiving the same color.

We are looking for a conflict free collection of colored tiles. In other words we are looking for an independent set in the conflict graph. Or we are looking for a clique in the complement of the conflict graph. Only one colored version of each of the six uncolored tiles can enter into an independent set. On the other hand each uncolored tile must occur in one colored version in the independent set. As there are six uncolored tiles we are looking for an independent set of size six in the conflict graph. Equivalently, we are looking for a clique of size six in the complement of the conflict graph.

An inspection shows that the colored tiles numbered 1, 3, 6, 10, 13, 19 form an independent set in the conflict graph. From this we can read off a coloring of the node of the given hypergraph H.

node	1	2	3	4	5	6	7	8
color	1	1	2	1	2	1	1	1

This coloring of the nodes is a legal coloring of the nodes of H using two colors.

Example 3.3. *Using the hypergraph H we construct an auxiliary hypergraph $G = (W, F)$. In Section 3.3.2 we did not cover the case when the tiles are identical with the hyperedges of H. In this example we choose the number r to be* 1.

In this case the tiles coincide with the hyperedges of the hypergraph H. For the sake of a unified treatment we listed the tiles in Table 3.8. The list of pair-wise distinct tiles is the following

$$T_1 = \{1, 3, 6\}, \quad T_2 = \{1, 3, 8\},$$
$$T_3 = \{1, 5, 8\}, \quad T_4 = \{2, 4, 5, 7\}.$$

hyperedge	tile
$\{1,3,6\}$	$\{1,3,6\}$
$\{1,3,8\}$	$\{1,3,8\}$
$\{1,5,8\}$	$\{1,5,8\}$
$\{2,4,5,7\}$	$\{2,4,5,7\}$

Table 3.8: The tiles coincide with the hyperedges in Example 3.3. The hyperedges are cut into one tile.

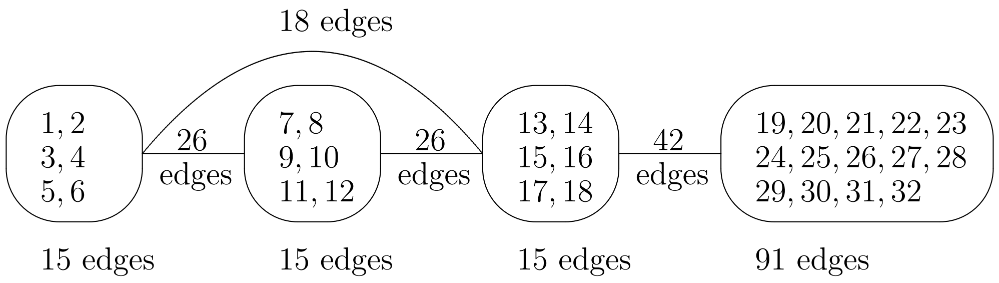


Figure 3.3: The condensed form of the conflict graph G in Example 3.3. The nodes inside an oval are pair-wise connected by edges. Edges between ovals represent many edges. The number of edges are written near to the ovals and near to the edges.

Table 3.9 lists the colored tiles. We dropped the colored tiles whose elements are colored with one color. The remaining 32 colored tiles are the nodes of the conflict hypergraph G.

There is a conflict in the following case.

1. Two tiles are not disjoint and the common part of the tiles is not colored in the same way in the two tiles.

The conflict hypergraph is a 2-uniform hypergraph, that is, an ordinary graph. We are looking for an independent set of size 4 in the conflict graph. Or a clique of size 4 in the complement of the conflict graph. The 4 tiles we constructed must be colored in same way and the corresponding 4 colored tiles must be conflict free.

$$\begin{bmatrix}1&3&6\\1&1&1\end{bmatrix} \quad 1:\begin{bmatrix}1&3&6\\1&1&2\end{bmatrix} \quad 2:\begin{bmatrix}1&3&6\\1&2&1\end{bmatrix} \quad 3:\begin{bmatrix}1&3&6\\1&2&2\end{bmatrix}$$

$$4:\begin{bmatrix}1&3&6\\2&1&1\end{bmatrix} \quad 5:\begin{bmatrix}1&3&6\\2&1&2\end{bmatrix} \quad 6:\begin{bmatrix}1&3&6\\2&2&1\end{bmatrix} \quad \begin{bmatrix}1&3&6\\2&2&2\end{bmatrix}$$

$$\begin{bmatrix}1&3&8\\1&1&1\end{bmatrix} \quad 7:\begin{bmatrix}1&3&8\\1&1&2\end{bmatrix} \quad 8:\begin{bmatrix}1&3&8\\1&2&1\end{bmatrix} \quad 9:\begin{bmatrix}1&3&8\\1&2&2\end{bmatrix}$$

$$10:\begin{bmatrix}1&3&8\\2&1&1\end{bmatrix} \quad 11:\begin{bmatrix}1&3&8\\2&1&2\end{bmatrix} \quad 12:\begin{bmatrix}1&3&8\\2&2&1\end{bmatrix} \quad \begin{bmatrix}1&3&8\\2&2&2\end{bmatrix}$$

$$\begin{bmatrix}1&5&8\\1&1&1\end{bmatrix} \quad 13:\begin{bmatrix}1&5&8\\1&1&2\end{bmatrix} \quad 14:\begin{bmatrix}1&5&8\\1&2&1\end{bmatrix} \quad 15:\begin{bmatrix}1&5&8\\1&2&2\end{bmatrix}$$

$$16:\begin{bmatrix}1&5&8\\2&1&1\end{bmatrix} \quad 17:\begin{bmatrix}1&5&8\\2&1&2\end{bmatrix} \quad 18:\begin{bmatrix}1&5&8\\2&2&1\end{bmatrix} \quad \begin{bmatrix}1&5&8\\2&2&2\end{bmatrix}$$

$$\begin{bmatrix}2&4&5&7\\1&1&1&1\end{bmatrix} \quad 19:\begin{bmatrix}2&4&5&7\\1&1&1&2\end{bmatrix} \quad 20:\begin{bmatrix}2&4&5&7\\1&1&2&1\end{bmatrix} \quad 21:\begin{bmatrix}2&4&5&7\\1&1&2&2\end{bmatrix}$$

$$22:\begin{bmatrix}2&4&5&7\\1&2&1&1\end{bmatrix} \quad 23:\begin{bmatrix}2&4&5&7\\1&2&1&2\end{bmatrix} \quad 24:\begin{bmatrix}2&4&5&7\\1&2&2&1\end{bmatrix} \quad 25:\begin{bmatrix}2&4&5&7\\1&2&2&2\end{bmatrix}$$

$$26:\begin{bmatrix}2&4&5&7\\2&1&1&1\end{bmatrix} \quad 27:\begin{bmatrix}2&4&5&7\\2&1&1&2\end{bmatrix} \quad 28:\begin{bmatrix}2&4&5&7\\2&1&2&1\end{bmatrix} \quad 29:\begin{bmatrix}2&4&5&7\\2&1&2&2\end{bmatrix}$$

$$30:\begin{bmatrix}2&4&5&7\\2&2&1&1\end{bmatrix} \quad 31:\begin{bmatrix}2&4&5&7\\2&2&1&2\end{bmatrix} \quad 32:\begin{bmatrix}2&4&5&7\\2&2&2&1\end{bmatrix} \quad \begin{bmatrix}2&4&5&7\\2&2&2&2\end{bmatrix}$$

Table 3.9: The colored tiles assigned to the hyperedges in Example 3.3. The first rows of the matrices contain the tiles and the second rows contain the colors.

1	8	9	10	11	12	16	17	18
2	7	10	11	12	16	17	18	
3	7	10	11	12	16	17	18	
4	7	8	9	12	13	14	15	
5	7	8	9	12	13	14	15	
6	7	8	9	10	11	13	14	15
7	14	16	17	18				
8	13	15	16	17	18			
9	14	16	17	18				
10	13	14	15	17				
11	13	14	15	16	18			
12	13	14	15	17				
13	20	21	24	25	28	29	32	
14	19	22	23	26	27	30	31	
15	19	22	23	26	27	30	31	
16	20	21	24	25	28	29	32	
17	20	21	24	25	28	29	32	
18	19	22	23	26	27	30	31	
19								
.								
.								
.								
32								

Table 3.10: The edges of the conflict graph in Example 3.3. The 7-th row of the table holds the information that the unordered pairs $\{7,14\}$, $\{7,16\}$, $\{7,17\}$, $\{7,18\}$ are edges of the conflict graph G.

3.3.4 An application

Voloshin [Volo2002] introduced the following type of coloring of the nodes of a hypergraph. The edges of the given hypergraph H are labeled as C type or D type hyperedges. An edge may belong to both types or may belong to neither. We color the nodes of the hypergraph H in the following way.

1. Each node receives exactly one color.
2. All the nodes of a D type hyperedge cannot receive the same color.
3. The nodes of a C type hyperedge cannot receive all different colors.

It is easy to see that the proposed construction of Γ_1 and Γ_2 conflict graphs can easily be carried out for this type of graph coloring too.

One remarkable property of mixed hypergraph coloring is that there are some mixed hypergraphs that cannot be colored properly at all. In his book Voloshin proposes some open questions and this particular one is among them. “Develop a probabilistic method for the colorability problem. Let $H = (V, C, D)$ be a mixed hypergraph with the probability of each C edge given by p and the probability of each D edge is given by q. What is the probability, as a function of p and q that H will be colorable.” We are not going to solve the proposed problem, but can back up this question with some extended computational results. The question is ambiguous as it leaves open if the same subset can be a C and D edge at the same time or not. So we considered both possibilities.

We constructed a big series of 3-uniform mixed hypergraphs. In series A all 3 element subset of the nodes were either a C edge or a D edge or no edge. A random number $0 \leq z < 1$ was generated for each 3 nodes, and if $z < p$ these nodes became a C edge, if $p \leq z < q$ these nodes became a D edge. In series B we allowed the same 3 size subset to be a C edge with probability of p and to be a D edge with probability of q. That means that the same subset can be either a C edge, or a D edge, or both, or not an edge. As in each graph at least one C edge was present that meant that the graph cannot be colored by $|V|$ colors. So we asked the question if it can be colored by $|V| - 1$ colors or less, and constructed an auxiliary graph Γ accordingly. There are few practical software for hyperclique or hyper independent search. A recent publication [Tor2017] can deal with only small hypergraphs. We used $r = 2$, that is normal graphs for this construction. This also meant that we did not need two but only one auxiliary graph. We performed the k-clique search using the algorithm described in Chapter 5.

The nodes of the auxiliary graph Γ are all possible pairs of the set V colored by all possible colors. At this juntion we would like to point that when we ask if a the nodes of a hypergraph can be colored legally using k colors we actually mean k or less than k colors.

The series of experiments calculated the colorability of graphs of size 6,8,10,12 and 14. We set the values for p and q all possible ways by 5% steps. We generated 20 instances with the same p and q values, and checked the colorability of the resulting graphs. The results are the f frequency of the colorable ratio of these graphs, and pictured in Figure 3.4 for A series, and Figure 3.5 for B series.

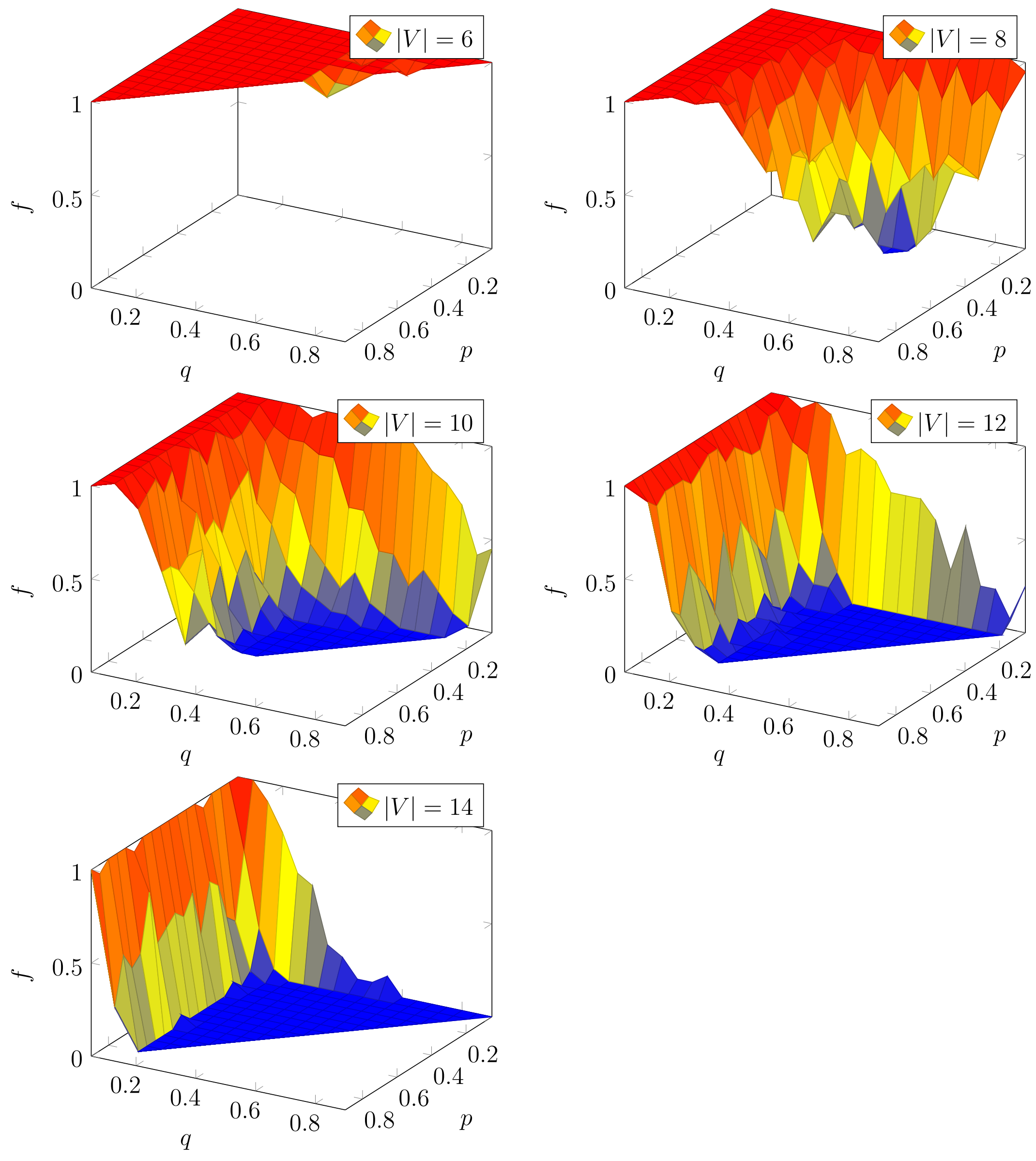


Figure 3.4: Results of the A series, where an adge can be a C edge or a D edge but not both

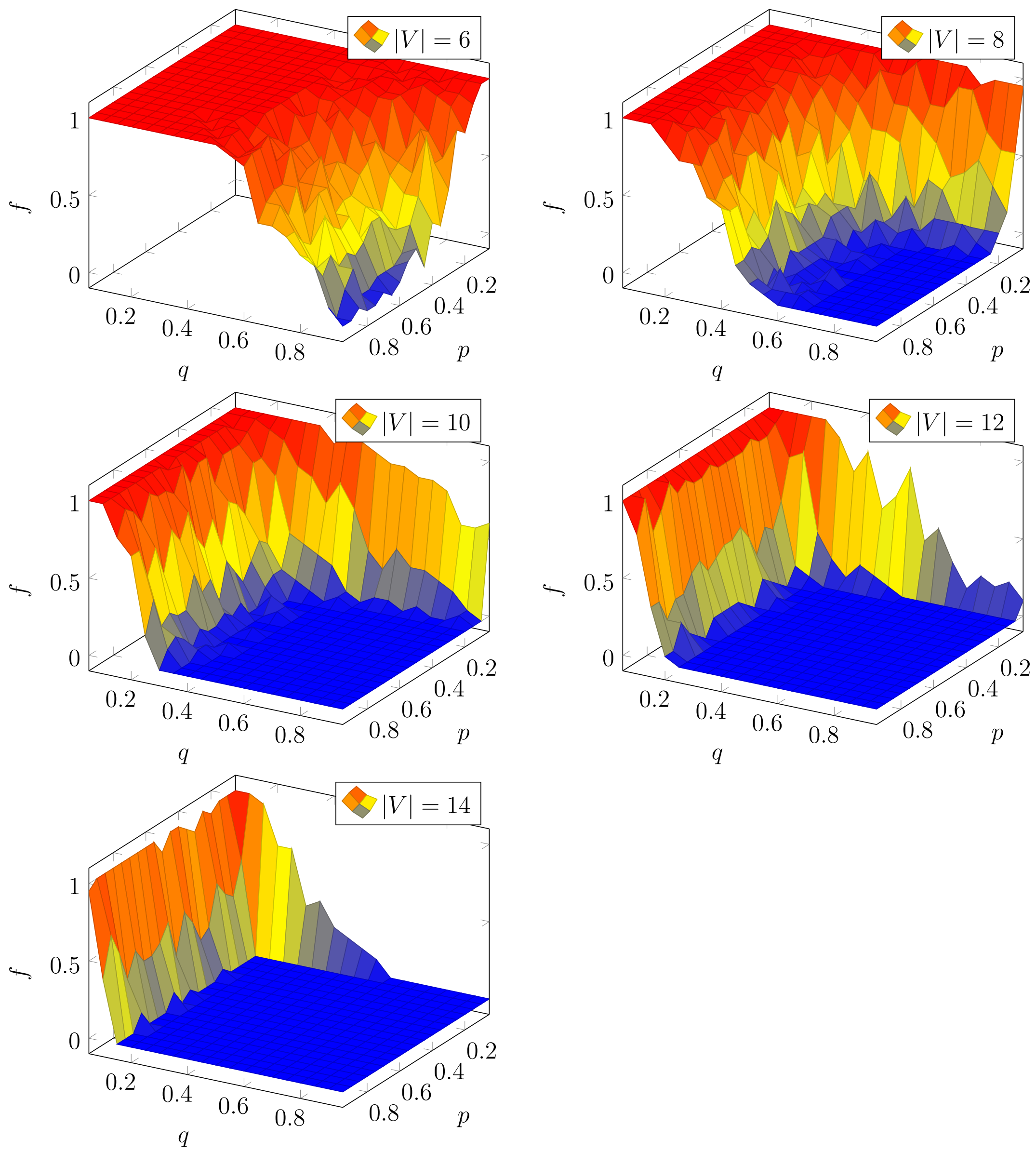


Figure 3.5: Results of the B series, where an edge can be C edge and D edge at the same time

Chapter 4

Maximum clique solvers, kernelization, auxiliary algorithms

In the present chapter we wold like to summarize the current results on maximum clique solvers. First, we shortly list the history of these solvers and try to point out the special steps in the development of the maximum clique search algorithms. Second, we should compare the auxiliary algorithms used in these solvers for bounding and cutting the search tree. The results are from [Szab2017]. Finally, we would like to focus on kernelization, that is the preconditioning techniques. These are becoming a very interesting branch for solving hard combinatorial optimization problems, although they are not yet found their way into the usage of the clique search community.

Apart from exact methods we would like to mention that the maximum clique problem, as being very hard for even medium sized graphs, is often solved by some heuristic approach [Bom1997]. Apart from solving it by usual computer program [Lam2016], some even use custom FPGA for a Markov-chain Monte-Carlo search, which the authors name "Digital Annealer" [Nagh2019], and even quantum computing is used [Chap2019].

4.1 Sequential algorithms

The history of exact maximum clique search programs is a long one [Bom1999]. The efficient programs always use some Branch-and-Bound technique [Carm2012].

The first specialized algorithm for clique search was the Bron-Kerbosh [Bron1973]. It lists all maximal cliques, and today is not considered good for maximum clique search. The second important algorithm was by Carraghan and Pardalos [Carr1990]. This algorithm makes the base for almost all algorithms by today, at least any author would claim that.

The refinement of the Carraghan-Pardalos algorithm was using node coloring for bounding function instead of just the size of the 'prospective' nodes set. Many researchers was doind research in this area including Tomita [Tom2003], Kumlander [Kuml2005, Kuml2020] and Konz [Konc2007]. An interesting variation of the program was embedding the search into a russian doll techique by Östergård [Öst2002].

Nowadays the basic direction of the refinement is to use better bounds then the bound coming from coloring. They would call such bound 'infrachromatic' noting this feature. Researchers like San Segundo [Seg2011, Seg2012, Seg2013, Seg2014, Seg2015, Kom2015, Nik2015] or Li [Li2010a, Li2013, Li2017] are the most prominent researchers in this field.

4.2 Auxiliary algorithms

In this section we would like to focus on different upper bounds for the clique size. These bounds can be used for estimating such upper bound for large problems and then compared to some heuristically found result. But they are also extendedly used in the state-of-the-art clique solvers. These bounding functions give us different results – some are sharper then others –, but the algorithms that giving us these results are also of different complexity. Also, some bounds can be computed exactly, while calculation the exact value for others is NP-hard, so heuristic algorithms are in use. One would be interested to compare these algorithms and bounds, and that is exactly we would like to do in this section.

4.2.1 Coloring

The most common method for establishing a bound for $\omega(G)$ is determining a legal coloring of the nodes of the graph G such that:

1. Each node receives exactly one color.
2. Two adjacent nodes cannot have the same color.

More formally, a node coloring of G with r colors, also an r-coloring, is a surjective map $f : V \to \{1, \ldots, r\}$. Here we identify the r colors with the numbers $1, \ldots, r$ respectively. The level sets of f are the so-called color classes of the coloring. The i-th color class $C_i = \{v : v \in V, f(v) = i\}$ consists of all the nodes of G that are assigned color i. The color classes $C_1, \ldots, C_r$ form a partition of V. Vice versa, the coloring is uniquely determined by the color classes $C_1, \ldots, C_r$.

The smallest number of colors required by any legal coloring of the nodes of the graph G is called the chromatic number of the graph and denoted by $\chi(G)$. As the nodes of a clique Δ are all pairwise adjacent, any legal coloring of the nodes in Δ will require at least as many colors as the cardinality of the clique. Consequently, the chromatic number is always an upper bound for the size of the largest clique in the graph, that is, $\chi(G) \geq \omega(G)$.

Finding the chromatic number of a graph is well known to be NP-hard. In practice, approximate coloring algorithms are used as bound for the clique number. Specifically, we are interested in two coloring heursitics. The first is the well known Dsatur algorithm from Daniel Brélaz [Brel1979]. The second one is the iterative coloring heuristic from Joseph C. Culberson [Culb1992], in the following IC.

There is a vast literature on coloring, such as [Ross2014]. But different coloring methods lay outside the scope of our work.

4.2.2 Fractional and b-fold coloring

[Wood1997]

Any legal k-coloring of the nodes of a graph $G = (V, E)$ assigns the same color number $i = 1 \dots k$ to the nodes of each independent set C_i that determines a partition of V. On the other hand, *fractional coloring* assigns one real number as the weight of a color to *every* independent set [God2001, pp. 135–138.]. More formally, let $I(G, u)$ denote the independent sets of G that contain the node u. A fractional coloring of the nodes of G is a nonnegative real function f such that, for any node $u \in V$,

$$\sum_{S \in I(G,u)} f(S) \geq 1$$

The sum of the values of f for every node is called the *weight* of the fractional coloring, and the minimum possible weight of every fractional coloring of the nodes of G is called the *fractional chromatic number* $\chi_f(G)$.

A simpler variation of the fractional coloring, the *b-fold coloring* of G, is an assignment of a set of b colors to every one of its vertices such that adjacent vertices receive disjoint sets. An *a:b-coloring* is a b-fold coloring out of a available colors. Finally, the b-fold chromatic number $\chi_b(G)$ is the smallest a such that an a:b-coloring exists. The special case of $b = 1$, the 1-fold coloring, reduces to finding a legal node coloring, so consequently finding the b-fold chromatic number is NP-hard. The connection between fractional and b-fold chromatic number is the following:

$$\chi_f(G) = \lim_{b \to \infty} \frac{\chi_b(G)}{b} \tag{4.1}$$

A b-fold coloring of the nodes of a graph G also bounds the clique number from above. Also, each k-clique must receive $b \times k$ colors, thus $\omega(G) \leq \frac{\chi_b(G)}{b}$. Since computing the fractional chromatic number is NP-hard, we consider the heuristic b-fold coloring as substitute problem. To further reduce computation resources we consider only small values of b, in particular b has been set to 5 in our experiments.

From (4.1) it can easily be established that the bound derived from any b-fold coloring is also an upper bound for the clique number of the graph. Our choice of b-fold coloring is motivated by the fact that this coloring can be easily reformulated as a legal node coloring of a graph [Szab2016b]. This is done by using an auxiliary graph $\Gamma = (W, F)$ constructed from the given $G = (V, E)$ graph. The nodes of Γ are ordered pairs $(v_i, k) \in W, v_i \in V, 1 \leq k \leq b$. The edges are defined as follows:

$$F = \begin{cases} \{(v_i, k), (v_i, l)\} & \text{if } k \neq l \qquad\qquad 1 \leq k, l \leq b \\ \{(v_i, k), (v_j, l)\} & \text{if } \{v_i, v_j\} \in E \quad 1 \leq k, l \leq b \end{cases} \tag{4.2}$$

It is easy to see from (4.2) that any legal node coloring of the graph Γ represents a b-fold coloring of the nodes of the graph G and vice versa. The key idea is that the different color numbers assigned to the nodes (v_i, k) in Γ become the set of colors assigned to v_i in the corresponding b-fold coloring of the nodes of G. Figure 4.1 shows Γ in a 5:2-fold coloring of the nodes of C_5 cycle, that is, a 2-fold coloring using 5 colors.

If we are given a b-fold coloring to G, which means that a node v_i of G is assigned b different colors, then assigning these b colors to the nodes $v_{i,1}, v_{i,2}, \ldots, v_{i,b}$ in result we get a legal coloring for Γ. The same way backwards we can construct an b-fold coloring for G in the case if we are given a legal coloring to Γ.

4.2.3 s-clique free coloring

Another modification to legal coloring may be proposed [Szab2012]. Let $G = (V, E)$ be a finite simple graph and let s be a positive integer such that $s \geq 2$.

Definition 4.1. *A subset U of V is called an s-free set if the graph spanned by U in G does not contain any s-clique. A partition $U_1, \ldots, U_r$ of V is called an s-clique free partition of V if U_i is an s-clique free subset of V for each i, $1 \leq i \leq r$.*

We color the nodes of G such that each node receives exactly one color of the given r colors. A coloring of the nodes of G with r colors can be

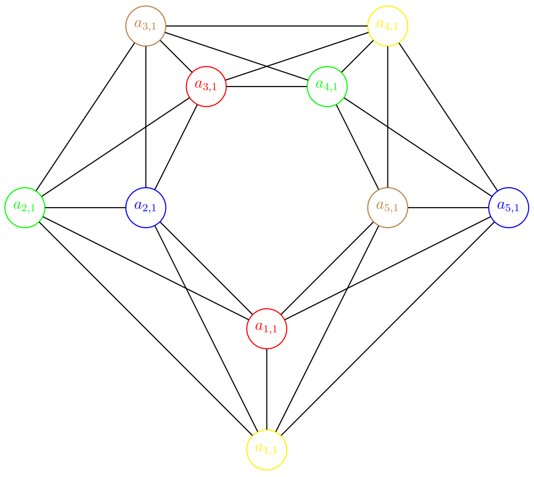


Figure 4.1: The Γ auxiliary graph for C_5 and its node coloring with 5 colors.

described more formally as a surjective map $f : V \to \{1, \ldots, r\}$. Here we identify the r colors with the numbers $1, \ldots, r$, respectively. The level sets of f are the so-called color classes of the coloring. The i-th color class $C_i = \{v : v \in V, f(v) = i\}$ consists of all the nodes of G that are colored with color i. The color classes $C_1, \ldots, C_r$ form a partition of V. Obviously, the coloring is uniquely determined by the color classes $C_1, \ldots, C_r$.

Definition 4.2. *A coloring of the nodes of G with r colors is called an s-clique free coloring if the color classes $C_1, \ldots, C_r$ are all s-clique free subsets of V.*

In particular, in a 2-clique free coloring of G adjacent nodes cannot receive the same color. A 2-clique free coloring is commonly referred as a legal or well coloring of G. In a 3-clique free coloring of G the nodes of a triangle in G cannot receive the same color.

Proposition 4.1. *If G has an s-clique free coloring with r colors, then $\omega(G) \leq r(s-1)$.*

Proof. Let Δ be a k-clique in G and suppose that $C_1, \ldots, C_r$ are the color classes of an s-clique free coloring of G with r colors. Note that Δ can have at most $s-1$ nodes in each of the color classes $C_1, \ldots, C_r$. It follows that $k \leq r(s-1)$ and therefore $\omega(G) \leq r(s-1)$. □

We implemented two different algorithms for obtaining s-clique free colorings. One is based on Brelaz' DSatur algorithm. Here the problem arose from defining the saturation of a node. The other algorithm is derived from the Culberson's iterative coloring scheme. Note that combining any $s-1$ color classes of a legal node coloring we get an s-clique free coloring. So we started from a legal coloring scheme and started the iterative recoloring proposed by Culberson, where the color classes were defined as above. Sadly the results were not satisfactory. While for some tiny graphs we could get better results for moderate and big graphs as our proposed test graphs we never get any better result for upper estimate for $\omega(G)$ as the legal coloring. So we omit here the table of our non-conclusive results.

4.2.4 Edge coloring

Edge coloring can also provide a good upper bound for the clique number. We consider an *edge coloring* of a graph G with k colors an assignment of color numbers to the edges of G such that:

1. Each edge of G receives exactly one color.
2. If x, y, z are distinct nodes of a 3-clique in G, then the edges $\{x, y\}$, $\{y, z\}$, $\{x, z\}$ must receive three distinct colors.
3. If x, y, u, v are distinct nodes of a 4-clique in G, then the edges $\{x, y\}$, $\{x, u\}$, $\{x, v\}$, $\{y, u\}$, $\{y, v\}$, $\{u, v\}$ must receive six distinct colors.

Note, that this edge coloring *differs* from the one usually found in the graph literature. Comparable to node coloring, edge coloring can also be used as an upper bound for the clique number of G, base on the following property:

Property 4.1. *Let Δ be an l-clique in a graph G, and let G be edge-colorable with k colors. Then $l(l-1)/2 \leq k$ holds.*

Proof. A legal edge coloring of G must also provide a legal edge coloring of $\Delta]$. Since any legal edge coloring of Δ must contain at least $l(l-1)/2$ colors, then $l(l-1)/2 \leq k$, as required. □

The procedure to color the edges of a graph G is to use an auxiliary graph $\Gamma = (W, F)$. Each edge of G is represented by a node in Γ. We connect the nodes of Γ according the rules above, that is two nodes in Γ should be connected if the corresponding edges in G forming a 3- or a 4-clique. It is easy to see that any legal coloring of the nodes of Γ represents a legal edge coloring of G. The auxiliary graph Γ can be quite large, but greedy coloring procedure like the Brélaz' Dsatur can still be used. We constructed the auxiliary graph Γ from all our test graphs and tried to run first the Brélaz' Dsatur coloring procedure, then using its output the Culberson's iterative coloring algorithm on these auxiliary graphs.

4.2.5 Lovász number

The last bound considered is the *Lovász number* of a graph, a real number that is an upper bound on the Shannon capacity of the graph [Lov1976, Karg1998, Hrg2019]. It is also known as *Lovász theta function* and is commonly denoted by $\vartheta(G)$. Lovász theta is actually an upper bound for the maximum independent set. There are several formulations of this number, Knuth composed an extended list [Knu1994].

4.2.6 The partial MaxSAT bound

The bound based on partial MaxSAT was first described in [Li2010a], and referred to as UB_{SAT}. It reduces the maximum clique problem for a k-colored graph G to a partial maximum satisfiability problem, and employs typical Boolean constraint propagation techniques to prove that no k-cliques exists in G. This bound is at least as good as the bound coming from coloring. In the literature we find that it has been successfully applied as bounding function to determine the clique number of a graph in the algorithms MaxCLQ [Li2010b] and IncMaxCLQ [Li2013].

The input of the algorithm is an independent set partition $C_1, C_2, \ldots, C_k$ of the vertices of the G, that is a coloring of the nodes of the graph. The algorithm greedily looks for r *conflicting* subsets of independent sets $I = \{I_1, I_2, \ldots, I_r\}$ to reduce the upper bound for the clique number from the original k to $k - r$. A conflicting subset I_j in I is such that $\omega(G[I_j]) < |I_j|, (1 \leq j \leq r)$, where $G[I_j]$ is the graph induced by the vertices in I_j, and $|I_j|$ denotes the number of independent sets. For this step the algorithm uses unit clause propagation from SAT. It stores all unit independent sets in a queue Q, and repeatedly dequeues each one and assumes that its single node v is part of a new clique. In the remaining independent sets, all vertices that do not belong to v's neighbor set $N(v)$ are removed, which, in turn, may lead to fresh independent sets added to Q. The procedure ends when either Q is empty, or an independent set becomes empty.

A node v is called *failed* if the application of UP driven by $Q = \{v\}$ leads to an empty independent set. If this is the case, and v belongs to a unit independent set f, a *conflict* I_v is determined by the set of independent sets that participated in the UP chain (including the final one that became empty) together with f. Another possible conflict derives from a non unit f set with all its vertices failed. In this case, $I_f = \bigcup_{v \in f} I_v$.

The second way to find disjoint conflicts with overlapping independent sets is by using the clause relaxation method employed in maximum Boolean satisfiability (MAX-SAT) [Fu2006]. Once a conflicting set of cardinality s

is found, a fresh node $w_i, (1 \leq i \leq s)$ is added to each independent set in the conflict such that it is connected to all the other vertices in G with two exceptions. Those vertices that belong to its same independent set, and the other fresh vertices $w_j, (j \neq i)$. Each added node and its enlarged independent set are denoted *relaxed*. It is easy to see that for a given conflict I_l, the set of $|I_l|$ relaxed vertices thus defined cover all possible cliques of size $|I_l|$ in I_l. Once a relaxed independent set $C_j, (1 \leq j \leq k)$ from I_l becomes unit and is inserted into the UP queue Q, the remaining sets in $I_l \setminus C_j$ can take part in future disjoint conflicts.

4.2.7 Numerical experiments

The listed auxiliary algorithms are of different complexity thus have big variation in running time. The upper bounds derived from them are also different. For some we do not know how good the bound is, for others we have some theoretical comparison, see [Karg1998, God2001]:

$$\omega(G) \leq \chi_v(G) \leq \vartheta(\bar{G}) \leq \chi_f(G) \leq \chi(G) \tag{4.3}$$

We performed extended measurements on a carefully selected data set of 35 graphs, and reported the results in Table 4.1. The first 11 graphs in the Table come from various error correcting code problems [Sloan][1]. The next 19 graphs are taken from the 2nd DIMACS Challenge [Hass1993][2]. The next 3 graphs are reformulated problems of monotonic matrices [Szab2013][3], and the last 2 are from the so-called EVIL instances [Szab2019a][4]. For the reported graphs, the clique number is extremely hard to compute, and in some cases these problems are still open. There are also instances where $\omega(G)$ is known but which cannot be determined by state-of-the-art solvers. Examples of these are the EVIL graphs, where $\omega(G)$ is known by construction, and a subset of the *johnson* graphs. The latter are derived from code theory – values for $\omega(G)$ available at `https://www.win.tue.nl/~aeb/codes/Andw.html#d4`.

Table 4.1 reports the sizes of the 35 selected graphs, along with their clique number, or alternatively the best known bounds. Also in the Table, the columns *Dsatur color* and *IC color* show the best upper bound for $\omega(G)$ considering the Dsatur and the IC coloring heuristics respectively. The IC coloring is the obtained after 1000 iterations, taking the Dsatur coloring as

[1]`https://oeis.org/A265032/a265032.html`

[2]`http://iridia.ulb.ac.be/~fmascia/maximum_clique/DIMACS-benchmark`

[3]`http://mathworld.wolfram.com/MonotonicMatrix.html`

[4]`http://clique.ttk.pte.hu/evil`

starting point. For the experiments conerning UB_{SAT} the coloring obtained from IC was the starting point.

We used the CSDP program from Brian Borchers [Bor1999, Bor2007] for calculating the value of Lovasz' theta, $\vartheta(\bar{G})$.

We omitted the result on 3-free coloring, as we could not design a heuristic algorithm that would give a better bound then legal node coloring. This task is yet to be done.

A "*" in any cell refers to a bound that could not be computed due to memory or time limitations. Table 4.1 also reports some previous, not published, results by the author for some problems using a supercomputer. These results are also indicated by a "*" and the bound is given in parenthesis. Those calculations used up to several hundred of cores, up to half terabyte of memory, and they run sometimes for weeks.

Table 4.1 reports the clique bound provided by Dsatur, and the best bound obtained by the iterative Culberson's method for legal coloring, the Lovasz' theta function over the complement graph, the partial MaxSAT bound, the Culberson's iterative method for the auxiliary edge graph and the 5-fold node coloring. From the table, the best results were always provided by the Lovasz' theta function. The second best bound came from different algorithms, but mostly from the 5-fold coloring.

4.3 Kernelization

In the present section we would like to list some possible preconditioning tools that can aid the maximum and k-clique solvers. Although there is no strict definition the term "kernelization" usually used for such algorithm which reduces the problem instance [Cyg2015]. For clique search we usually would like to delete some nodes or edges, or transform the graph into a smaller one. Our list is not complete, for other kernelization methods in this area see [Aki2016, Hes2020].

4.3.1 Structions

In this section we rely on publications [Ebe1984, Ale2003]. We introduce some notations. Let $V = \{1, \ldots, n\}$ be the set of nodes of G. We may assume that node 1 is the pivot node, that is, the node which plays a pivotal role in the struction construction. This is only a matter of renaming the nodes of the graph G.

Let $A = \{a_1, \ldots, a_r\}$ be the set of neighbors of the pivot node 1 and let $B = \{b_1, \ldots, b_s\}$ be the set of non-neighbors of the pivot node 1. Using the

	$\|V\|$	$\omega(G)$	Dsatur color	Iterated color	$\vartheta(\bar{G})$	pMax SAT	Edge color	5-fold color
1dc.512-c	512	52	83	74	53.03	136	65	56.8
1dc.1024-c	1024	94	152	137	95.98	136	*	103
1dc.2048-c	2048	172–174	304	268	*(174.73)	266	*	190.2
1et.1024-c	1024	171	225	215	184.23	209	*	194.4
1et.2048-c	2048	316	436	404	342.03	399	*	358.6
1tc.1024-c	1024	196	241	229	206.3	225	*	217.2
1tc.2048-c	2048	352	450	426	374.64	422	*	389.8
1zc.512-c	512	62	104	93	68.75	92	84	74.4
1zc.1024-c	1024	112–117	201	177	128.67	176	*	138
2dc.1024-c	1024	16	34	30	*	29	*	22.2
2dc.2048-c	2048	24	65	54	*	53	*	38
brock800_2	800	24	134	118	*	117	79	107
brock800_4	800	26	136	118	*	117	79	106.4
C1000.9	1000	68–	305	255	*	246	*	236.6
C250.9	250	44	92	78	56.24	71	70	76
C500.9	500	57–	164	140	84.2	132	123	131.8
hamm10-4	1024	40	85	74	*	73	*	55.8
johns-10-4-4	210	30	48	41	30	40	37	32
johns-11-4-4	330	35	71	61	41.25	60	53	45
johns-11-5-4	462	66	97	88	66	87	81	66.4
johns-12-4-4	495	51	99	86	55	85	73	60.8
johns-12-5-4	792	80	161	138	99	137	128	99
johns-13-4-4	715	65	133	116	71.5	115	*	80.4
johns-13-5-4	1287	123–	248	212	143	211	*	143
keller5	776	27	61	31	*	31	42	31
keller6	3361	59	141	63	*	63	*	63
MANN_a45	1035	345	369	360	356.05	359	*	360
MANN_a81	3321	1100	1153	1134	1126.62	1133	*	1134
p_hat1500-3	1500	94	270	265	*	263	*	244.8
p_hat700-3	700	62	143	134	*	131	105	125
monoton-9	729	28	53	47	*(34.41)	46	46	42.6
monoton-10	1000	32–	71	60	*(41.83)	59	59	53.2
monoton-11	1331	37–	84	72	*(49.96)	71	73	64.2
evil-N330	330	60	109	100	71.99	85	90	90.2
evil-N500	500	80	165	140	100	119	121	128.2

Table 4.1: The summary for the upper limit of $\omega(G)$ by different methods. The sign $*$ indicates that the bound cannot be computed due to time or memory limit.

set B we construct a new set C. The elements $b_1, \dots, b_s$ of B are listed such that

$$b_1 < \cdots < b_s \tag{4.4}$$

holds. The ordered pair (b_α, b_β) is an element of C whenever the unordered pair $\{b_\alpha, b_\beta\}$ is an edge of G and $\alpha < \beta$. We denote such an ordered pair by $c(b_\alpha, b_\beta)$.

The set of nodes of the struction graph G' is $A \cup C$. Two distinct elements a_i and a_j from A are adjacent in G' if the unordered pair $\{a_i, a_j\}$ is an edge of G. Otherwise a_i and a_j are not adjacent in G'.

Two distinct elements $c(b_\alpha, b_\beta)$ and $c(b_\gamma, b_\delta)$ from C are adjacent in G' if $b_\alpha = b_\gamma$ and the unordered pair $\{b_\beta, b_\delta\}$ is an edge of G. Otherwise $c(b_\alpha, b_\beta)$ and $c(b_\gamma, b_\delta)$ are not adjacent in G'.

An element $c(b_\alpha, b_\beta)$ from C and an element a_i from A are adjacent in G' if the unordered pairs $\{b_\alpha, a_i\}$ and $\{b_\beta, a_i\}$ are edges of G. Otherwise the elements $c(b_\alpha, b_\beta)$ and a_i are not adjacent in G'.

Lemma 4.1. *If Δ' is a clique in G', then there is a clique Δ in G, where $|\Delta| = |\Delta'| + 1$*

Proof. If Δ' contains no node from C, then $\Delta = \Delta' \cup \{1\}$ is such a required clique.

If Δ' contains nodes from C, they must be of the form $c(i, b_\alpha)$, $c(i, b_\beta)$, $c(i, b_\gamma), \dots$, and the nodes $i, b_\alpha, b_\beta, b_\gamma, \dots$ are the nodes of a clique in G. Then $\Delta = \Delta' \setminus \{c(i, b_\alpha, c(i, b_\beta), c(i, b_\gamma), \dots\} \cup \{i, b_\alpha, b_\beta, b_\gamma, \dots\}$ is a clique of G. $\square$

Lemma 4.2. *If Δ is a clique in G, then there is a clique Δ' in G', where $|\Delta'| = |\Delta| - 1$*

Proof. Let Δ be a clique in G, and set $\Delta_0 = \Delta \cap (A \cup \{1\})$.

If $|\Delta_0| = 0$ then removing any node from Δ results a required clique Δ' in G'.

If $|\Delta_0| = 1$ then $\Delta' = \Delta \setminus \Delta_0$ is a required clique.

If $\Delta_0 = \{i_1, i_2, \dots, i_r\}$, where $r \geq 2$ and $i_1 < i_2 < \cdots < i_r$, then $\Delta' = (\Delta \setminus \Delta_0) \cup \{c(i_1, i_2), c(i_1, i_3), \dots, c(i_1, i_r)\}$ is a required clique. $\square$

The following result is a corollary of Lemma 4.1 and 4.2.

Theorem 4.1. $\omega(G) - 1 = \omega(G')$

Before embarking on a large scale clique search it is advisable to carry out a thorough inspection of the original graph to detect deletable nodes and edges. We will use the notation $N(a)$ for the set of neighbors of the node a.

4.3.2 Color indices

Suppose that the nodes of a finite simple graph G are legally colored using k colors and $C_1, \ldots, C_k$ are the color classes of this coloring.

Definition 4.3. *The color index of a node v of G (with respect to a legal coloring of the nodes of G) is the number of color classes C_i that contains at least one node adjacent to v.*

Note that if the color index of a node v is less than $k - 1$, then v cannot be a node of a k-clique in G. In other words if the colors index of v is at most $k - 2$, then v can be deleted from G without loosing any k-clique.

Definition 4.4. *The color index of an edge $\{u, v\}$ of G (with respect to a legal coloring of the nodes of G) is the number of color classes C_i that contains at least one node adjacent to u and v simultaneously.*

Note that if the color index of an edge $\{u, v\}$ is less than $k - 2$, then the edge $\{u, v\}$ can be deleted from G when one is looking for a k-clique in G. (We do not delete the nodes u or v.)

4.3.3 Dominance

Definition 4.5. *Let G be a graph and let a, b be distinct nodes of G. We say that node b dominates node a if a and b are not adjacent and $N(a) \subseteq N(b)$.*

The basic observation is that a dominated node can be dropped from the graph during the search for a k-clique. Note that we may loose k-cliques during this reduction. But we are not going to loose all of them.

Definition 4.6. *Let G be a graph and let a, u, b be distinct nodes of G such that $\{a, u\}$, $\{u, b\}$ are edges of G. We say that edge $\{u, b\}$ dominates edge $\{a, u\}$ if $b \notin N(a) \cap N(u)$ and $N(a) \cap N(u) \subseteq N(u) \cap N(b)$.*

If edge $\{u, b\}$ dominates edge $\{a, u\}$, then the edge $\{a, u\}$ can be canceled from G when we are deciding if G contains a k-clique. (We do not delete the nodes a or u.)

Definition 4.7. *Let G be a graph and let x, y, u, v be distinct points of G such that $\{x, y\}$, $\{u, v\}$ are edges of G. We say that edge $\{u, v\}$ dominates edge $\{x, y\}$ if $\{u, x\}$ or $\{u, y\}$ is not edge of G, and $\{v, x\}$ or $\{v, y\}$ is not edge of G, and $N(x) \cap N(y) \subseteq N(u) \cap N(v)$.*

If edge $\{u, v\}$ dominates edge $\{x, y\}$, then the edge $\{x, y\}$ can be canceled from G when we are deciding if G contains a k-clique. (We do not delete the nodes x or y.)

Chapter 5

New method for k-clique search and its extension to a maximum clique solver

In the last years the scientific viewpoint about NP-complete and NP-hard problems has shifted. First, today we have a more detailed analysis which can distinguish between subclasses. By doing so we know that although two problem are being in the same NP-hard class can be quite different in difficulty. Second, the introduction of parameterized algorithms showed that some problems happen to be much easier than the conservative worst case prediction made by being in a certain class. In these cases some extra information can guide the algorithm to solve the problem more efficiently. Finally, the approach of parameterized algorithms sometimes able to deal with the more complex problems by dividing the problem into an easier and a harder part. Solving the easy parts as a preprocessing step we are left with the hard part and so reducing the size of the original problem.

In this chapter, which is based mostly on [Szab2018a], we would like to propose an approach driven by these ideas. The problem in question is the NP-hard combinatorial optimization problem of maximum clique search in simple graphs. There are many different algorithms and proposition for solving it, and most of the proposed algorithms are being refinements of the Carraghan-Pardalos algorithm. (Patric Östergård's cliquer is being as an exception.) Our contribution is based on the fact that the NP hard maximum clique optimization problem can be replaced by a series of NP-complete k-clique decision problems. The structure of a k-clique search algorithm is simpler than the maximum clique problem. In addition the combined search space of the k-cliqe problems is significantly smaller than the original maximum clique problem. Basically the parameter k drives the search and thus we are able to reduce the size of the search tree.

This approach turns out to be more efficient in several cases. Using simple methods, our program can keep up with the much more sophisticated programs and even beat them on several instances.

5.1 Background

The Carraghan-Pardalos algorithm [Carr1990, Wu2015] forms the base for many of the exact clique search algorithms. In this sense it occupies a special position among the clique search procedures. The Carraghan-Pardalos algorithm divides a given clique search instance into smaller instances by choosing nodes into a prospective clique and repeats this process. We can see this algorithm as a good example of the well known Branch and Bound algorithm family.

In the present chapter we would like to introduce a different approach for the maximum clique search optimization problem. As known from the literature (see [Cyg2015]) one can often build a more efficient parameterized algorithm for the k-clique search problem. So we followed this path and started to build our own program, the $k$clique, which instead of solving the optimization problem of maximum clique deals with the decision problem of k-clique. Based on this program we could build a very simple and yet efficient maximum clique search program, which we will call $k$clique-sequence. We will first detail the key features of building a k-clique search program. Second we will shortly describe the maximum clique search program. Last, we will present results by a large scale numerical experiments. We will test our procedures using well established test graphs, and compare these results to those of the other programs. Finally, we will evaluate the results and make comments and remarks on our implementation.

The program was implemented in `C++` language, and we used dynamic bitsets from the Boost Template Library.

5.2 Nuts and Bolts for k-clique search

The main idea behind our program is the strong reduction of the search tree. For both branching and bounding the choice of searching for k-clique helps us to reduce the search tree. It is also worth noting, that the k-clique approach can help to make an efficient parallel program as well as the reader can learn in Chapter 8.

5.2.1 Branching and Bounding

Other implementations for maximum clique use the biggest yet found clique as a bounding condition. It is more than obvious, that such bounding will be dependent on the early finding of a big clique. Thus, those programs that find the final maximum clique early run faster. That is possibly why the heuristics of ordering nodes by node degree in *descending* order plays such an important role in these algorithms. And certainly that is why parallel implementations of maximum clique search [McCr2015] find strange speed-up results – sometimes even superlinear speed-ups – in the literature, as they are dependent on this early big clique finding. In our case, by choosing a different problem formulation, we always can bound by the value of k.

Also, for branching we can use the value of k. As we describe later, we use coloring in our implementation. It is well known, that the coloring gives us an upper limit for the clique size. Thus if given the value of k and a coloring with c colors ($c \geq k$), then we can choose the smallest $c - (k - 1)$ color classes, and use those nodes in them for branching – as a *branching rule.* (As a terminology later we will call these nodes the k-clique covering node set – KCCNS –, as introduced in the Chapter 7.) The importance of this comes from the nature of the Branch and Bound algorithms. These algorithms sort out the nodes already examined, meaning that they are not taken into account in the future search. Thus if all these nodes are eliminated, then the remaining nodes can be colored with $(k - 1)$ colors, so there cannot be any k-clique present. Note, that without the value of k one cannot make this branching rule, and need to branch on all nodes. This method is the base of our Branch and Bound algorithm. The size of the k-clique covering node set is the branching factor, and finding the smallest possible of such set can aid us in bounding the size of our search tree.

Algorithm 1 summarizes our $k$clique algorithm based on this branching rule.

Algorithm 1 $k$clique

```
Require: $G = (V, E), P = V$
1: function $k$CLIQUE($P, k$)
2:     if $k = 1$ then return true
3:     KCCNS $\leftarrow$ construct a $k$-clique covering node set
4:     for all vertex $p \in$ KCCNS do
5:         if $k$CLIQUE($P \cap N(p), k - 1$) then return true
6:         $P \leftarrow P \setminus \{p\}$
7:     return false
```

5.2.2 Efficient coloring

It is well known from the literature, that coloring of the nodes can speed-up the clique search. Two methods are widely used in the literature. One is a simple sequential greedy algorithm, where in sequence we place the nodes in the first possible color class. The second is a coloring procedure named Dsatur presented by Daniel Brélaz [Brel1979], which always places the least "suitable" node into a color class. The Brélaz's algorithm usually gives us better approximation to the chromatic number than the simple sequential coloring algorithm, but it costs more in time complexity.

Designing a clique search algorithm one usually decides over the first, especially when coloring takes place not only at the top of the search tree. Using Dsatur at all levels reduces the size of the search tree, but costs in time – as we learned from our preliminary test runs. Actually we need not to choose between these two algorithms. An efficient algorithm can use a costly DSatur coloring at the top of the search tree, and later it can switch to the cheaper sequential greedy coloring.

Although Dsatur gives us a good coloring it is often quite far from the optimum. So we used in addition another technique, the Iterated Coloring presented by Culberson [Culb1992]. This technique uses reordering the color classes and using a sequential coloring several times. The result cannot be worse than the previous coloring in terms of the number of colors, but it can be better. Thus we started from a Dsatur coloring and performed iterated coloring. Our stopping criteria was if the number of colors did nod decreased after 1000 iterations, and we used it on the top of the search tree.

5.2.3 Recoloring the nodes

Some researchers proposed methods that would use the color classes from the previous level of the search tree and use a recoloring procedure to improve the actual coloring. [Nik2015] Here we present a similar approach.

As it was described by Culberson [Culb1992], the sequential greedy coloring has a special property, namely that it does not increase the number of colors if it is applied to suitable orderings of the nodes. If this procedure given a graph sequenced by color classes of a best possible coloring in which exactly so many color classes used as the chromatic number of the graph, then it will produce a coloring with the same number of color classes. If this procedure given a sequence ordered by any color classes, then it will result with a coloring at least as good as the previous one. One may see this procedure as repacking the color classes. Each node is moved forward as far as possible in the already given color classes, and never backwards.

So during the Branch and Bound procedure, when there are less and less nodes as we go down on the search tree, we can use the coloring of the previous level, and use the repacking feature of the sequential greedy coloring. We sort the color classes by their size, and start a greedy sequential coloring from the biggest color class. As the k-clique covering node set is actually the set of smallest color classes, the nodes from them moved ahead to the bigger color classes. So this procedure directly reduces the size of the k-clique covering node set and so the branching factor. Our tests showed us, that using this method the size of the search tree is comparable with that when we would use a DSatur coloring at each level while reducing the running time.

As this coloring is performed on each node of the search tree, we needed a fast implementation of the sequential greedy coloring. Our original version was of $O(c|V|)$ for c colors, and proved quite fast. Later we implemented this coloring using bitsets, similarly to the algorithm described in [Kom2015]. This method led to an even faster algorithm.

5.2.4 Rearranging branching nodes

From previous results [Zav2014a, Zav2014b, Zav2015] on parallel clique search algorithms we concluded, that the branching is even more important than it was thought before. It seems that the sequence of the nodes by which we proceed in the branch has a big effect on the search tree size if pruning is present. This was shown for SAT problems [Ouy1998], and could be shown for clique search problems as well. This effect is used by our algorithm, and so it reduces the search space. Note though, that this effect is different from the effect of finding a big clique early, where the sequence by decreasing node degrees proved useful!

We use a very basic reordering rule. We proceed with the nodes with the smallest degree in the remaining subgraph. That is we ordered the nodes by node degree in *increasing* order. By doing this we solve first the more easy problems and reduce the size of the later ones. Although simple and "cheap" this approach had quite a good effect on the size of the search tree.

5.3 Numerical results for maximum clique

The structure of our maximum clique problem is extremely simple. First we find an upper bound for the size of the maximum clique. Although we tried more sophisticated methods simply using the number of colors from the coloring of the graph was good enough and the fastest. Note, that we started with the Dsatur coloring and used Culberson's Iterated Coloring scheme till the number of colors did not changed for 1000 iterations.

We set k equal to the obtained number of colors and run our $k$clique program with this parameter. If the result was that the graph did not contain a k-clique we decreased the value of k by one. Doing these procedure as a sequence our program finally founds the biggest value of k for which there is a k-clique present. Thus $\omega(G) = k$. We called this program "$k$clique-sequence down", see Algorithm 2. Note, that the program calls Algorithm 1 several times.

```
Require: $G = (V, E)$
  function MAIN
      $k \leftarrow$ an upper bound by coloring
      $k$CLIQUE-SEQ-DOWN
      Print $k$ as the size of the maximum clique
```

Algorithm 2 $k$clique-sequence down

```
1: function $k$CLIQUE-SEQ-DOWN
2:     FOUND $\leftarrow$ false
3:     while $\neg$FOUND do
4:         FOUND $\leftarrow$ $k$CLIQUE$(V, k)$
5:         if $\neg$FOUND then
6:             $k \leftarrow k - 1$
7:     return k
```

Our opinion was that the most time consuming part of maximum clique search is the last but one step, namely to prove the non-existence of a clique of size $\omega(G)+1$. Our experiments confirmed this view in most of the cases. We also included the running time of this particular subproblem in the Tables 5.1 and 5.2 under column "$k$clique, $k = \omega(G) + 1$". Also McCreesh and Prosser pointed out so in the Section 3.3 in their work [McCr2015], and our results confirmed this. This observation confirms our approach of using the k-clique search in this sequential way as opposed for example to a more complex binary search.

Though, one can question the decision of using a top-down approach, and instead propose using a bottom-up sequence. We also implemented the second algorithm as well, and it is built the following way. First we find a lower bound for the size of the maximum clique. This is done by a simple greedy clique search algorithm. We set k equal to the obtained number plus one and run our $k$clique program with this parameter. If the result was that the graph do contain a k-clique we increased the value of k by one. Repeating this procedure as a sequence our program finally finds the smallest value of k for which there is no k-clique present. Thus $\omega(G) = k - 1$. We call this program "$k$clique-sequence up", see Algorithm 3.

```
Require: $G = (V, E)$
  function MAIN
      $k \leftarrow$ a lower bound by greedy clique search
      $k \leftarrow k + 1$
      $k$CLIQUE-SEQ-UP
      Print $k - 1$ as the size of the maximum clique
```

Algorithm 3 $k$clique-sequence up

```
function $k$CLIQUE-SEQ-UP
    FOUND $\leftarrow$ true
    while FOUND do
        FOUND $\leftarrow$ $k$CLIQUE$(V, k)$
        if FOUND then
            $k \leftarrow k + 1$
    return k
```

5.3.1 Test graphs

We performed extended measurements on a carefully selected data set of different graphs and various maximum clique search programs. As there are no challenges performed nowadays – the last well known is being the 2nd DIMACS Challenge more than 25 years ago – we used test graphs and programs that have been published about exact maximum clique search recently. Our program intends to solve previously infeasible or extremely hard problems, thus we choose appropriate problems. Most of these test problems can be solved only in minutes, if not in several hours. In the present chapter we chose 50 examples out of all together 80 graphs tested in our extended experiment. Those that were not chosen were either too easy, that is solvable under a couple of second, or too hard, that is could not be solved in 12 hours by any program including ours. We also omitted some examples that were too repetitive in kind and results.

The Table 5.1 list graph from various sources. The first 7 graphs are taken from the 2nd DIMACS Challenge [Hass1993][1]. The next 2 graphs are

[1]`http://iridia.ulb.ac.be/~fmascia/maximum_clique/DIMACS-benchmark`

reformulated problems of monotonic matrices [Szab2013][2], and the next 2 graphs in the table come from various error correcting code problems[Sloan][3]. Last 13 graphs are problems of Erdős–Renyi type random graphs [Erd1959],

In the second table, Table 5.2, we first list instances from benchmark tests collected in the BHOSLIB library.[4] Next in the table are graphs from the so-called EVIL instances [Szab2019a][5].

5.3.2 Results

For comparison we choose the most known and best state-of-the-art programs for exact maximum clique search.

We compared our program with the programs by Östergård[6] [Öst2002], Li[7] [Li2010a, Li2013], Konc[8] [Konc2007], Prosser[9] (who implemented Tomita's algorithm [Tom2003]) and San Segundo[10] [Seg2011, Seg2013, Seg2014, Seg2015]. We indicated not only the running times but the size of the search tree as well. (For this purpose a small modification was made to the `cliquer` program, and the type of the counter in `mcqd` needed to be changed to `long long` from `int`.)

For our $k$clique program we indicated not only the time and search tree size of the whole maximum clique search, but also the time and the search tree size for the $k = \omega(G) + 1$ step, that is where our program proved the nonexistence of cliques of size one bigger than $\omega(G)$.

Columns "$k$clique-seq up" and "$k$clique-seq down" show result from our programs, $|V|$ the size of the graph, % the density of the graph and $\omega(G)$ stands for the clique size of the graph.

The hardware used for comparison was a Xeon E5-2670 v3 machine at 2.30GHz clock speed with 128GB of RAM, and we used a 12 hour time limit.

5.3.3 Evaluation

As the reader can see in Tables 5.1 and 5.2, our simple approach of running the k-clique program with various k values gives time result that are usually comparable or even better than most of the other programs. It is also worth

[2]`http://mathworld.wolfram.com/MonotonicMatrix.html`
[3]`https://oeis.org/A265032/a265032.html`
[4]`http://www.nlsde.buaa.edu.cn/~kexu/benchmarks/graph-benchmarks.htm`
[5]`http://clique.ttk.pte.hu/evil`
[6]`http://users.aalto.fi/~pat/cliquer.html`
[7]`http://home.mis.u-picardie.fr/~cli/EnglishPage.html`
[8]`http://www.sicmm.org/konc/maxclique/`
[9]`http://www.dcs.gla.ac.uk/~pat/maxClique/distribution/`
[10]`https://www.biicode.com/pablodev/examples_clique`

	$\lvert V\rvert$	%	$\omega(G)$	$k$clique, $k = \omega(G)+1$	$k$clique-seq down	$k$clique-seq up	BBMC (S.Seg.)	BBMC-R (S.Seg.)	BBMC-L (S.Seg.)	BBMC-X (S.Seg.)	Max-CLQ 10 (Li)	Max-CLQ 13 (Li)	mcqd (Konc)	mcqd-dyn (Konc)	MCR (Tomita)	cliquer (Österg.)
brock800-3	800	65	25	1955s	8837s	11171s	3297s	4614s	3539s	2452s	4580s	5561s	8883s	4290s	23013s	25459s
				654M	2505M	3361M	234M	138M	253M	91M	218M	18M	1545M	851M	1884M	38753M
brock800-4	800	65	26	1543s	7277s	14385s	2262s	2294s	2476s	1787s	4370s	7037s	6144s	3072s	22349s	6038s
				471M	1866M	4612M	149M	88M	164M	59M	229M	25M	1040M	564M	1915M	9554M
latin-	900	76	90	131s	213s	136s	>12h	>12h	>12h	>12h	47s	>12h	>12h	1180s	>12h	>12h
-square10				7M	11M	7M					448k			50M		
keller5	776	75	27	46s	53s	53s	>12h	>12h	>12h	>12h	6848s	238s	>12h	18098s	>12h	>12h
				8M	9M	8M					243M	249k		2271M		
MANN-a45	1035	99	345		>12h	>12h	148s	67s	155s	82s	25s	123s	2384s	2058s	3692s	>12h
							1M	118k	1M	118k	70k	22k	5339k	4647k	2852k	
sanr200-0.9	200	90	42	55s	141s	86s	27s	31s	30s	21s	7s	2s	123s	30s	913s	>12h
				15M	38M	25M	4M	1370k	4M	850k	442k	8k	28M	6M	62M	121156M
sanr400-0.7	400	70	21	140s	419s	157s	113s	135s	114s	105s	112s	117s	232s	110s	942s	3593s
				67M	169M	69M	15M	9M	17M	6M	8M	477k	82M	38M	102M	5535M
monoton-7	343	79	19	3s	12s	11s	49s	38s	42s	31s	29s	6s	151s	72s	249s	1354s
				1M	5M	5M	8M	5M	10M	5M	618k	28k	51M	20M	27M	2288M
monoton-8	512	82	23	577s	846s	733s	17919s	23327s	17219s	15049s	4131s	1279s	>12h	19272s	>12h	>12h
				181M	251M	197M	2663M	1760M	3344M	1577M	89M	6M	16125M	3386M		
1dc.256	256	88	30	2s	8s	19s	3s	2s	3s	2s	6s	5s	15s	22s	65s	375s
				558k	2M	6M	382k	218k	491k	181k	156k	22k	2463k	3133k	4114k	545M
2dc.1024	1024	68	16	112s	178s	135s	29s	36s	30s	40s	182s	199s	67s	146s	139s	3060s
				19M	28M	21M	2M	1509k	2M	1345k	2M	307k	8M	23M	8M	2768M
rand200-9	200	90	44	58s	204s	158s	33s	40s	36s	30s	12s	3s	225s	88s	1268s	>12h
				15M	52M	43M	7M	3M	9M	2M	494k	7k	42M	14M	84M	
rand300-7	300	70	20	7s	28s	15s	5s	8s	5s	6s	10s	5s	17s	11s	40s	95s
				4M	13M	8M	1440k	842k	1566k	583k	593k	26k	6M	4M	7M	152M
rand300-8	300	80	28	313s	984s	615s	162s	216s	160s	142s	154s	115s	758s	274s	2756s	29261s
				130M	379M	207M	33M	16M	37M	11M	9M	370k	201M	72M	271M	52065M
rand300-9	300	90	48		>12h	>12h	25333s	29220s	28071s	17574s	6645s	5894s	>12h	9800s	>12h	>12h
							3288M	1256M	4083M	802M	316M	9M		2641M		
rand500-5	500	50	13	2s	7s	2s	1s	2s	1s	1s	4s	4s	3s	3s	7s	11s
				1M	3M	1M	224k	143k	239k	102k	262k	18k	1110k	844k	1211k	14M
rand500-6	500	60	17	24s	86s	40s	22s	36s	24s	20s	57s	32s	61s	39s	136s	197s
				11M	35M	15M	4M	2M	4M	1669k	3M	165k	17M	11M	20M	283M
rand500-7	500	70	23	712s	2356s	997s	886s	1267s	662s	638s	968s	929s	1978s	928s	6456s	16244s
				313M	898M	378M	106M	60M	115M	40M	47M	2M	549M	227M	577M	26668M
rand800-4	800	40	12	2s	11s	5s	1s	2s	1s	1s	7s	6s	3s	3s	7s	11s
				901k	4M	2M	137k	81k	137k	60k	377k	34k	951k	863s	1078k	14M
rand800-5	800	50	14	45s	166s	49s	31s	51s	35s	31s	113s	107s	75s	59s	177s	297s
				21M	60M	21M	4M	3M	4M	2M	5M	491k	23M	17M	25M	348M
rand800-6	800	60	19	859s	3161s	1232s	1123s	1465s	1072s	736s	1571s	1657s	2523s	1496s	6546s	8398s
				397M	1151M	399M	89M	54M	96M	38M	77M	5M	509M	285M	609M	12397M
rand1000-4	1000	40	12	9s	39s	11s	7s	9s	6s	6s	36s	20s	16s	14s	32s	24s
				4M	12M	4M	734k	459k	780k	328k	1440k	64k	4M	3M	5M	27M
rand1000-5	1000	50	15	172s	679s	198s	125s	216s	129s	152s	526s	356s	364s	265s	795s	784s
				71M	216M	73M	16M	10M	17M	7M	21M	1486k	87M	65M	94M	1176M
rand1000-6	1000	60	19	7519s	26398s	9942s	9858s	13019s	9552s	7164s	16558s	17093s	19683s	9833s	>12h	>12h
				3258M	9441M	3366M	812M	491M	873M	335M	687M	57M	4622M	4641M		

Table 5.1: DIMACS, coding theory and random instances. Running time results in seconds. The “>12h” sign indicates that the running times are exceeding the 12 hour limit.

	$\lvert V\rvert$	%	$\omega(G)$	$k$clique, $k=$	$k$clique-seq	$k$clique-seq	BBMC	BBMC-R	BBMC-L	BBMC-X	Max-CLQ	Max-CLQ	mcqd	mcqd-dyn	MCR	cliquer
				$\omega(G)+1$	down	up	(S.Seg.)	(S.Seg.)	(S.Seg.)	(S.Seg.)	10 (Li)	13 (Li)	(Konc)	(Konc)	(Tomita)	(Österg.)
frb30-15-1	450	82	30	0s	0s	0s	1611s	1645s	1694s	1613s	575s	0s	2735s	2541s	3673s	0s
				0	13k	16k	256M	129M	316M	105M	25M	0k	514M	474M	288M	455k
frb30-15-2	450	82	30	0s	0s	0s	1010s	1094s	1191s	1047s	921s	0s	3329s	4155s	905s	2s
				0	30k	45k	159M	86M	205M	68M	40M	0k	668M	878M	73M	5M
frb35-17-1	595	84	35	0s	2s	3s	>12h	>12h	>12h	>12h	>12h	1s	>12h	>12h	34983s	6231s
				0	322k	437k						0k			1723M	12630M
frb35-17-2	595	84	35	0s	5s	11s	>12h	>12h	>12h	>12h	>12h	1s	>12h	>12h	>12h	33533s
				0	1M	2M						0k				89673M
frb40-19-1	760	86	40	0s	16s	29s	>12h	>12h	>12h	>12h	>12h	1s	>12h	>12h	11589	>12h
				0	2M	3M						0k			403M	
frb40-19-2	760	86	40	0s	9s	19s	>12h	>12h	>12h	>12h	>12h	0s	>12h	>12h	>12h	>12h
				0	1M	2M						0k				
frb45-21-1	945	87	45	0s	827s	1216s	>12h	>12h	>12h	>12h	>12h	119s	>12h	>12h	>12h	>12h
				0	111M	126M						11k				
frb45-21-2	945	87	45	0s	175s	253s	>12h	>12h	>12h	>12h	>12h	72s	>12h	>12h	>12h	>12h
				0	22M	26M						5k				
frb50-23-1	1150	88	50	0s	1287s	7227s	>12h	>12h	>12h	>12h	>12h	764s	>12h	>12h	>12h	>12h
				0	125M	632M						46k				
frb50-23-2	1150	88	50	0s	3828s	5532s	>12h	>12h	>12h	>12h	>12h	363s	>12h	>12h	>12h	>12h
				0	428M	466M						19k				
frb53-24-1	1272	88	53	0s	44086s	>12h	>12h	>12h	>12h	>12h	>12h	4771s	>12h	>12h	>12h	>12h
				0	3883M							252k				
frb53-24-2	1272	88	53	0s	36514s	>12h	>12h	>12h	>12h	>12h	>12h	190s	>12h	>12h	>12h	>12h
				0	3309M							10k				
frb59-26-1	1534	89	59		>12h	>12h	>12h	>12h	>12h	>12h	>12h	>12h	>12h	>12h	>12h	>12h
chv12x10	120	92	20	1840s	1897s	1912s	4s	5s	4s	1s	1s	0s	4759s	48s	700s	0s
				3450M	3518M	3450M	2M	2M	3M	236k	58k	17k	17092M	87M	477M	0k
myc11x11	121	93	22	233s	238s	234s	8s	12s	7s	2s	1s	0s	1097s	85s	1081s	239s
				347M	350M	347M	6M	5M	6M	241k	60k	10k	2537M	174M	845M	4860M
myc5x30	150	97	60	47s	63s	47s	1s	1s	1s	0s	0s	0s	10s	2s	47s	42042s
				11M	15M	11M	194k	43k	210k	7k	1k	0k	2M	326k	5M	235514M
s3m25x6	150	90	24	258s	279s	267s	192s	278s	195s	186s	4s	8s	64s	92s	128s	0s
				285M	298M	285M	219M	176M	258M	94M	234k	127k	49M	142M	49M	0k
chv12x15	180	94	30		>12h	>12h	26019s	34161s	26045s	7796s	1235s	184s	>12h	>12h	>12h	0s
							20118M	12990M	21790M	2366M	103M	3M				3k
myc23x8	184	90	16	623s	645s	645s	115s	165s	112s	88s	215s	90s	1138s	1390s	>12h	>12h
				892M	906M	892M	46M	45M	85M	24M	21M	2M	3051M	3192M		
myc11x17	187	95	34		>12h	>12h	40109s	>12h	33957s	5056s	2378s	2375s	>12h	>12h	>12h	3159s
							25910M		26338M	1223M	196M	38M				52113M
s3m25x8	200	92	32		>12h	>12h	46253s	>12h	44843s	38987s	181s	487s	22778s	18148s	40089s	0s
							44840M		50665M	16230M	10M	4M	16184M	10915M	12675M	581k
myc5x42	210	98	84	4006s	6048s	4067s	26s	15s	25s	4s	0s	0s	443s	36s	1414s	>12h
				573M	819M	583M	5M	591k	5M	94k	6k	0k	51M	3M	95M	
myc11x20	220	95	40		>12h	>12h	>12h	>12h	>12h	>12h	>12h	38519s	>12h	>12h	>12h	>12h
												366M				
myc23x10	230	91	20		>12h	>12h	>12h	>12h	>12h	38104s	26210s	7545s	>12h	>12h	>12h	>12h
										12201M	2083M	216M				
myc5x48	240	97	96		>12h	>12h	23s	22s	25s	13s	0s	0s	319s	39s	3316s	>12h
							4M	616k	4M	266k	8k	0k	33M	3M	90M	
s3m25x10	250	93	40		>12h	>12h	>12h	>12h	>12h	>12h	6980s	>12h	>12h	>12h	>12h	18s
											344M					104M

Table 5.2: BHOSLIB and EVIL instances. Running time results in seconds. The ">12h" sign indicates that the running times are exceeding the 12 hour limit.

noting, that our program achieves the main goal, and it reduces the search space considerably. It also clear, that although the “down” and “up” methods are different, they are very close and none of them is superior to the other. This follows from the fact that in our approach the proving of non-existence of a $(\omega(G) + 1)$-clique takes the most time, and that part is common to the two methods.

Evaluating the numerical results the reader can see, that our simple approach of running the k-clique program with decreasing or increasing k values gives running time result that are usually comparable with, or even better than most of the best state-of-the-art programs. In 4 cases our simple approach is clearly better than more sophisticated programs that using subchromatic bounds – the keller5 graph, the monoton-8 graph and two BHOSLIB instances. It is also at second place in 10 cases – the monoton-7, the latin square and all of the BHOSLIB instances.[11] It is also worth noting, that our program achieves the main goal, and it reduces the search space considerably. And it is very bad on one instance, the MANN-a45. But one would expect exactly such a mixed behavior for a program that is based on a very different approach. Note, that the EVIL graph instances were designed especially *against* programs using coloring, so it is not surprising that programs using subchromatic approach from San Segundo and Li perform better than our program.

Also, we would like to mention, that the proposed program is highly tunable, as one can use any suitable method to find the k-clique. We suspect that fractional coloring or other means of finding a better bound – see Section 4.2 – than the chromatic number may increase the efficiency of our program.

5.4 PACE competition

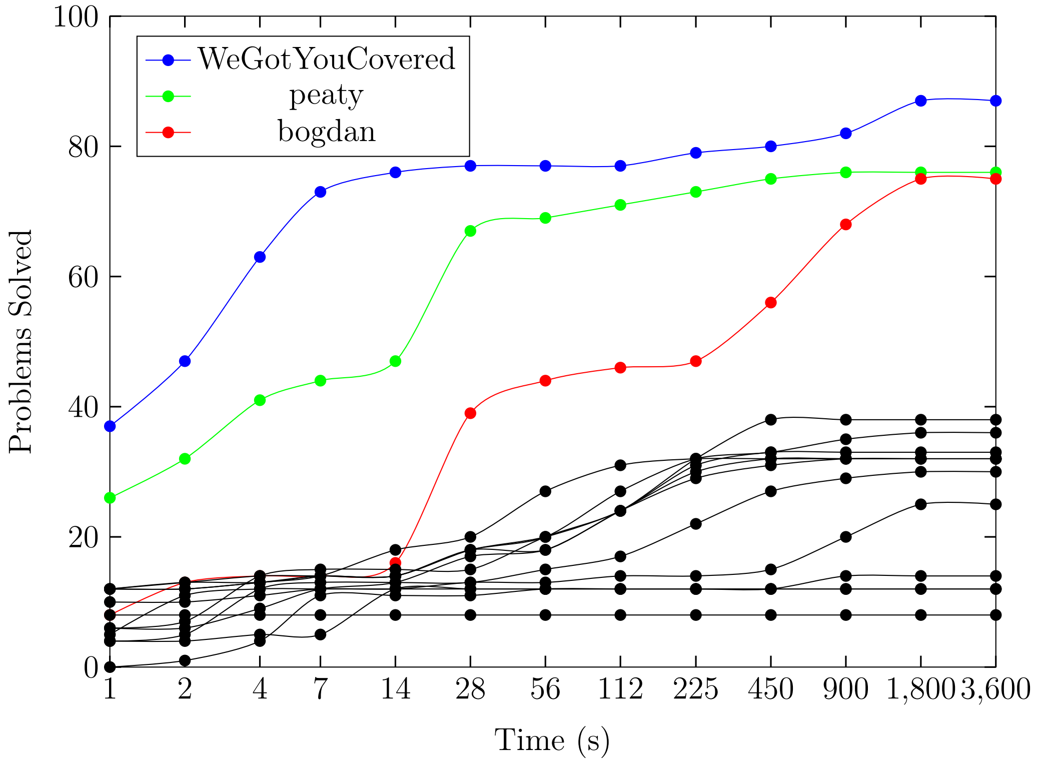
Perhaps not the best way to compare programs, but stil a good indicator is the results of the 1a track of the 2019 PACE competition, `https://pacechallenge.org/2019/`. The competition was about finding the minimum vertex cover, and was open for contestants Worldwide. The results proved that the best practice to this problem is strong kernelization and good maximum clique solver. The first place (solved 87 instances from 100) and second place (solved 77 instances from 100) teams used a maximum clique solver from Chu Min Li [Li2017], while the author of this work, result-

[11]Although the BHOSLIB instances are widely used for exact maximum clique search testing we do not consider them as really good tests. This is because of the fact that in each case $\omega(G) = \chi(G)$, and it is easy to find a coloring using $\chi(G)$ colors. The good behavior of our program is clearly because of this fact.

ing in third place (solved 76 instances from 100), used the solver described in this chapter [Dzu2019]. Other competitors could not solve even 40 instances. The results in detail depicted on Figure 5.1.

As kernelization[12] – see Section 4.3 –, played a huge role in the results one cannot directly conclude the performance of the underlying clique search program. But indirectly we can state, that our version of the clique search program is competitive to other state-of-the-art programs and among the best ones at the present.

Figure 5.1: PACE 2019, Exact Vertex Cover track. Number of solved instances in given time limit – the medalists compared to others.



[12]Our experiments, not detailed here, showed that reduction played little to no role in solving the problems listed in Section 5.3.3, explaining why such reductions are usually not implemented in maximum clique search programs.

Chapter 6

Concepts on parallelization

The main idea behind any parallelization algorithm is to divide the original problem into several subproblems [Braw1989, Gram2003, Karn2003, Matt2005, McCo2012, Pet2004, Var2013]. Then the problem of algorithmization lies in two key questions: Is the problem dividable? Can we easily construct the final answer from the sub-answers? For discrete optimization problems usually and for the clique problems particularly the division of the problem is quite straightforward. Also one can trivially give the final answer knowing the sub-answers.

In case of combinatorial optimization problems the main obstacle is the uneven distribution of problem hardness [McCr2015]. So if there are a few algorithms in this area, they are still not scaling well. For example the algorithm in [Chen2012] lists all maximal cliques, but it only scales for 32 processors.

Another possible usage of parallel architectures are for using a portfolio of algorithms [Gom1997].

Although the question lies outside of our work we must mention one special parallelization approach to combinatorial optimization. Namely, the parallel local search approach, which speeds up well for heuristic search of combinatorial structures. Although such methods were at our knowledge not used in clique search they were adopted to other similar problems like the already mentioned Costas Array Problem [Diaz2012b, Diaz2012a, Can2015].

6.1 The problem of even distribution

In the case of large discrete problems the unevenness of problem size and distribution makes scaling extremely problematic. Even in the most recent research using specialized MPI communication protocols researchers could achieve no better scaling as 2-4 thousand cores for the Breadth-first-search problem from the Graph500 test suite and only up to 5 hundred cores in the Unbalanced Tree Search problem [Dang2016]. Obviously the clique search problems we dealing with fall into these categories.

It is important to emphasize the importance of even distribution. The groundbreaking achievement of complete resolution of the Keller's Conjecture using graph representation and clique search of a 64 machine cluster [Deb2011] was possible by using methods that concluded in even distribution: "The computation took 109 days in this environment. This means of dividing the work resulted in a very even split. All machines finished on the same day (after 109 days of computation!) with the longest job (measured by the last update times for the output files) taking just 17 hours longer than the shortest one." Although we must note, that further scaling was less effective, as a 8192 processor supercomputer with 64 times more processors shortened the computation time less than 10 times.

Let us see an example. Given a graph $G = (V, E)$, we search for the answer if there is a k-clique in the graph. We pick an arbitrary node, and can construct two subproblems depending whether this node is part of the searched k clique or not. Let $u \in V$ be a node of graph G. We can construct two sub-graphs representing two subproblems as follows. Let G' be the spanned graph on $V' = V \setminus u$; and let G'' the spanned graph on the nodes neighboring u, that is $V'' = N(u)$. The first subproblem is to find k-clique in G' (the node u is not part of the searched clique), and the second subproblem is to find $(k-1)$-clique in G'' (the node u is part of the searched clique). The final answer will be 'yes', if at least one of the answers for the subproblems is 'yes', and 'no', if both answers are 'no'.

This method trivially can be extended by dividing the subproblems into even smaller subproblems, thus dividing the original problem into arbitrary many subproblems. Also one can create subproblems by using instead of a given node a given edge $\{u, v\}$. In this case we delete this edge from one graph ($E' = E \setminus \{u, v\}$), and constrain the other graph by the neighborhood of these two nodes ($V'' = N(u) \cap N(v)$). Finally, as an edge can be seen as a 2-clique, one can use 3-cliques or even any z-cliques for such division. The problem with such division that it will be too complex to describe the subproblems, and one need to use taboo lists in the search for excluding such cliques in some subproblems.

After the successful parallel algorithmization of the problem one still faces the problem of parallel efficiency. This depends on two key questions: What is the ratio of communication between the processes dealing with subproblems compared to the work to solve these subproblems? And is the distribution of the subproblems even?

The first issue is usually unimportant in case of discrete optimization. One can clearly see, that in the above proposed k-clique problem division absolutely no communication is needed. When more complex scheduling is used – we describe them in the next section – a little more communication is needed. Still, the amount of this communication is negligible to the complexity of the subproblems in case of mid sized and hard problems.

The other issue is the problem of even distribution. This is the real problem in discrete optimization. Again, the reader should think about the previously proposed division of the problem. Clearly the two subproblems won't be of equal size. So if one would have two processors to solve these problems, then one processor would finish early, while the other one would proceed with the calculations much, much longer.

The team of two BSc students on the National Student Competition (OTDK) with the author as advisor showed in their work, that the difference between the shortest and longest subproblem solution may be as much as 10^4 times for different hard problems [Har2015].

6.2 Effects on speedup

The main goals of the parallelization is to achieve more speed-up. In order to achieve this we need to take care of three separate problems.

The first one is to distribute the work among the processes (running on different processors) more evenly. If the distribution is uneven, then this will cause some processes run shorter time while the whole running time will be dominated by the slower processes. Let us look at an example. Take a problem which is decomposed into 100 subproblems, and we will use 10 processors and so 10 processes. If the subproblems are even and we give each process 10 subproblems, that is one tenth of the whole problem, then each process will finish its work roughly in one tenth of the original time. (We do not consider here other tasks connected to parallelization yet.) The speed-up is 10. Now, if we give one process 20 subproblems and the others only 9 or 8, then the processes given less tasks will roughly finish in less then half time as the one given 20 subproblems. The whole system will have a running time of the longest process, and that would lead to a speed-up of 5, which is clearly a worse case. Obviously, if we know that the subproblems are equal we will assign them to processes evenly. But in case of uneven subproblems we cannot do so, and we either must correctly guess the size of the subproblems to assign them evenly, or still end at an uneven distribution. In this chapter we deal with this question in details.

The second problem is the locality of the data. For shared memory systems this question is a serious one, as the possibility of reading and writing any data at any time hides away the real problem of how far the data is. Failing to notice this problem may cause a shared memory algorithm run *slower* by magnitudes! Also, there is a huge problem of cache usage, as nowadays multiprocessor computers are cache coherent, but achieving cache coherency may take quite a good time from the computation. For detailed analysis on cache coherency the reader may see the book [Sima1997]. For the question of algorithmic design considering cache usage of the book [Braw1989] produces very good examples.

In the case of distributed systems with message passing communication this problem is easier and harder at the same time. It is easier, because there is no possible way of directly reading data from the other computer, so the algorithmic design must always take this into consideration. But it is a harder problem, for the same reason, as there is no easy way of getting the data. One must always design it, consider it, and reconsider several times. And the main drawback is not the hardness of writing correct programs, but the communication overhead. If we are sending back and forth data many times, our program may be slower than the sequential version! At the end, we will always face a special problem, when adding more processors does not make the program faster, but even slows it down. In this case the execution time of the program is dominated by the communication time. So, when designing our problem decomposition we must take care of data locality, hence minimising the intercommunication between different processes and make them work mostly on their local data.

The third problem is the overhead of the algorithm. This one consists of that part which must be done in serial way and the cost of the parallelization itself. The serial part cannot be overridden. There will be always some parts – mostly setting up the problem, distributing the original data, collecting the sub-results, making tests on correctness –, which cannot be distributed. Moreover, we may add to these the start-up time of an MPI system. Starting several hundreds of processes on a distributed supercomputer may take some seconds. If the problem itself is solvable on a PC in some minutes, there is no reason for using a supercomputer, which will be slower, clumsier and overloaded most of the time.

6.2.1 Problem of decomposition

When we need to construct a parallel algorithm we usually want to decompose the problem into subproblems. For some cases this can be done straightforward, as the main problem itself consists of independent tasks or the algorithm deals with a set of data which can be split and processed independently. While there are still some interesting questions of submitting the split problems to different processes, this case counts as a good example for parallelization.

But this is not always possible. In case of quite a few problems we cannot divide the whole problem into independent subproblems, as they would not be independent. Many discrete algorithms fall into this category.

With some luck we can still deal with this case if we assign subproblems to parallel tasks, and combine the solution of subproblems by a sequential thread. One clear example can be the problem of sorting. We can assign a sub-domain of the elements to the parallel threads for sorting, and at the end combine these already sorted subsets into the final solution. This can be done by merging, so this scheme would be the parallel merge sort. By other means we can first divide the whole set into subsets such as the value of elements in each set is greater than in the previous set. In other words divide roughly the elements by value into baskets of different magnitude. After sorting these elements parallelly and routing back the solution to the master thread we are done, as each subset follows the previous one, and is sorted by itself. This scheme could be called the parallel Quicksort. But dividing the whole set into subsets raises a special problem, so a tuned version of subdivision is needed. By this subdivision is called this algorithm Samplesort.

6.2.2 Possible decomposition methods

As we have seen one of the main components of parallel algorithm design is the decomposition of the problem. We are given a large problem and our task is to split it up into parts so that these parts can be assigned to different computing resources. This task is not straightforward, and we can only outline some methods that can be helpful.

First, we should note, that one problem of the decomposition is the actual number of parts we would like to split the problem into. Actually we should construct our algorithm in that way that we do not know the actual number of processes in advance, as algorithms can be applied to different cases. The actual number will be decided in runtime. It seems that choosing the same number of parts as processes available is desired, but it is not so. If the parts cannot be constructed in equally computational expensive way then using the same number of parts will lead to unequal load balance. So actually using *more* subproblems as processes is more useful, as we can deal better with inequalities. Obviously using much-much more subproblems is also undesirable, as accounting and scheduling them will cost us resources comparable to those we use to do useful work. This means that using more subproblems as processes but not too much more is usually the goal. More detailed see [Mar1998].

6.2.3 Division by the branching tree

In the introduction of this chapter we described the basic step for dividing a problem into two subproblems. Now we would like to detail the possibilities for efficient division of a clique problem.

The most simple and widely used method is the division of the problem according to the search tree. This means that when we run a sequential algorithm it searches over a search tree in the problem space. We can divide the problem by using the top levels of the search tree. Namely, on the first level there will be problems represented by nodes. On the next level the problems will be represented by edges, and so on.

The problem with this approach, as it turns out, that its efficiency is lost after the second-third level [McCr2015]. The unevenness of the problems becoming so big, that we cannot gain by subdividing the problems further.

6.2.4 Fixed and dynamic distribution

The most simple method is clearly the fixed distribution. One would divide the problem into several subproblems without any extra knowledge of the evenness of the distribution itself and use a master–slave or by other name a work-pool distribution of the problems. In this case we would rather hope for even distribution than exactly knowing it. While it seems unreasonable, it has some advantages. Mainly, it is the simplest method, so it is easy to implement, which also leads to robust and reliable program code. The drawback of this method is obviously lays in the possible uneven solving time of the subproblems, as we already pointed out the possibility in the introduction of this chapter. In Chapter 8, which describes the “Las Vegas Parallelization Method” we will propose a possible approach to deal with this problem.

The usual question arising during this method is the number of created subproblems. Clearly, if we would have less subproblems than processing units then the parallel program would be inefficient, as there would be processors that would do no job at all. In some methods the number of subproblems are naturally concludes from the method itself. In this case the number of subproblems will give an upper limit of possible number of parallel processes. The methods of “disturbing structures” described in Chapter 7 falls into this category.

If the method of dividing the problem gives us more freedom to choose the number of subproblems, there are also two possibilities. First possibility would be to choose a number of subproblems be near the magnitude of the number of processors, say 1.5–3 times more. This will lead to a little more even distribution because the master–slave scheduling will give more work to those processors, which would finish early. Second possibility would be to choose number of subproblems to be several magnitudes more than the number of processors. The proposed name of this method is "Embarrassingly Parallel Search", which would go even further in the previously proposed way [Reg2013]. Authors of this method propose 30 times more subproblems as number of processors, but we suspect that in case of hard clique problems with 10^4 difference in problem solving times there should be even more of these subproblems. Alas if we make too many subproblems then the time of starting the solver for these subproblems on a supercomputer may be comparable to the problem solving itself. This issue needs more research.

In our opinion the work sharing by self scheduling is a good dynamic method. Other literature give other examples as well. On of them is called Work Stealing. It propose a static distribution in the beginning. When a process runs out of its assigned jobs, then it steals unstarted jobs from other, more occupied processes.

While the static distribution is the beginning may have real advantages, the more complex programing of the stealing makes this method harder to program.

6.3 Evaluation of scalability

6.3.1 Problematic case

A special problem of evaluating the speed-up may appear using varying processor count algorithm. Some algorithms, may use different processor counts in the different steps of the procedure. Adding to this one may use a normal PC for preconditioning an algorithm – mainly do some coloring. This step does not need a supercomputer and it is faster than being reasonable to use a parallel algorithm for. So if an algorithm using 1 processor in the beginning, thousand processors in the middle, and 64 at the end, then how could we calculate the speedup of the algorithm and compare the running times of this algorithm to other algorithms? This question is open, and needs arguments from the supercomputing community.

6.4 Framework for parallel implementations

There is an increasing demand for computational power. One core could not manage this need for over a decade. The speed of one core is limited. The computers of today reached a maximum frequency of 3-5GHz, which is quite flat in the last 15 years. Intel introduced its 3.8GHz Pentium4 processor 15 years ago.

There is a need of parallel computing, for multiprocessor systems. Later we used multi core processors and distributed systems. Today, we use literally millions of computing cores (`www.top500.org`). We can see, that we can have exponentially increasing computational capacities. But this raises new problems, namely heat dissipation and energy cost of supercomputers. The 3-5 years cost of energy supersedes the cost of the system itself.

We have also the problem of too many cores. Many algorithms do not scale properly over hundred thousands of cores.

6.4.1 Parallel architectures

Today's computer architectures vary in many ways. After we have seen the urge of computer power for large computations, we shall show the classification of nowadays' computers.

The PCs, the standalone computers at home or at the office are multi-core machines now. They have one CPU, but the CPU has more cores, typically 2 to 8, but there are processors with 16-32 cores, as well.

The next scale are bigger computers – these are the servers, usually multiprocessor systems. This later means they have more processors, usually 2 to 8. (Obviously these processors are multi-core, so such systems can have even 64 cores present.) Also specialized acceleration processors with hundreds or thousands of processors are in use, like Xeon Pi, Nvidia Tesla GPGPU card or the Sunway 260 processor.

For HPC, High Performance Computing much much bigger computers or computer systems are used. These provide thousands or millions of cores to their users. The SMP (Symmetric Multiprocessor) systems are shared memory systems, where the programs can access the whole memory. These systems nowadays are so called ccNUMA systems, which means that only part of the memory is close to the processor, while other – bigger – part of the memory is further away. This means more memory latency which is balanced by the usage of cache, that is why the system is *cc*, cache coherent. The programming of such systems is usually made by openMP programs, but bigger systems can run MPI as well.

The biggest supercomputers of our time are distributed systems or clustered computers. They consist of separate computers connected by a fast interconnect system, These are programed with MPI language extensions.

Other architectures of today's computers are video cards, which can be programed for different purposes and even used in supercomputers. This paradigm is called the General Programming GPU, the GPGPU, and programed with languages of for example CUDA or OpenCL.

6.4.2 Scheduling

Scheduling of subproblems can be done statically or dynamically. The problem with static assignment that it is hard to achieve even work distribution. So one would like to make some dynamic assignment.

One best known method to do this is the master-slave work processing or in other words post office parallelization.

There is a Master process, who takes care of and accounts for the work to be done. The Slave processes ask for a job to do, and report back when they are ready. After sending the results, they ask for another job. When the job pool is empty, in other words no job (to be done) has been left, the Master process tells the Slaves to terminate.

The basic – but yet incomplete – program part would be like this:

```
//N: number of elements
//id: the rank of the process
//a[]: the array of work descriptors

if(id==0){//Master process
  int id_from;
  unit_result_t ANSWER;
  for(int i=0;i<N;++i){
    //recive from anybody, tag=1:
      MPI_Rec(&id_from, 1, MPI_INT, MPI_ANY_SOURCE, 1,...);
    //recieve answer, tag=2:
    MPI_Rec(&ANSWER, 1, MPI_datatype, id_from, 2,...);
    //send to slave who asked for job, tag=3:
    MPI_Send(&a[i], 1, MPI_datatype, id_from, 3, ...);
  }
 }else{//Slave process
  unit_result_t ANSWER;
  unit_of_work_t QUESTION;
  while(true){
    //send our id
    MPI_Send(&id, 1, MPI_INT, 0, 1, ...);
    //send the answer
    MPI_Send(&ANSWER, 1, MPI_datatype, 0, 2, ...);
    //ask for the next question and calculate the answer
    MPI_Rec(&QUESTION, 1, MPI_datatype, 0, 3,...);
    ANSWER=do_work(QUESTION);
  }
 }

```

We can notice the incompleteness, as there is no start and no end in this question-answer conversation. These parts must be written separately. Also the reader must note the strict ordering of send and receive commands. If they would be in other order deadlock may occur, as discussed earlier in chapter three.

It is also important to make distinction between the different message types, and this is done by assigning different tags to messages of different functionality. It is also important in order to make the program error free.

6.4.3 Problems arising of parallelization

When we write a parallel program we obviously would like to achive faster computation. But apart from being fast it is an interesting question how fast it is. One would like to measure the goodness of the parallelization. Obviously, for different purposes different measures can be done. For some cases even the slightes reduction in running time may be important as a problem may be of a kind that it must be solved in some time limit. Others may look at the question from economical point of view and compare the income from faster computation with the invested money (and perhaps time). There cannot be an overall perfect measurement, but still there is one which is accepted the widest. This is called the speed-up, and it is calculated the following way. For a given problem we measure the running time of the best known sequential program and the running time of our parallel program. The ratio of them will give us the speed-up number, which by this definition will be dependent of processors or computers we use. One would like to develop and use a parallel program, that will achive a speed-up of n with n

processors, or in other words with twice as many processors half the running time.

In most cases, because the problem cannot be divided into independent subproblems, we should find an alternative algorithm, which is more complex than the original one. This algorithm would run on one thread slower than the original one. So adding more processors we speed up the modified algorithm, which means we are far from reaching speed-up of number of processors compared to the original algorithm, namely we are unable to reach linear speed-up. Still, there can be a good usage of such algorithms, as even with sub-linear speed-up and with the aim of many computers (or a bigger cluster or supercomputer) we can solve much bigger problems than with the original sequential program. One may say, that gaining speed-up of 1.5 with twice as many processors is a satisfying result.

6.4.4 Amdahl's law and Gustavson's law

At this point we need to mention two historical notes on the problem of speed-up and its limits. These notes called the Amdahl's law and the Gustavson's law.

The law by Gene Amdahl was formed in the late 60's, and states, that every parallel program has a limit in speed-up independently of the number of processors used. This phenomena is caused by the fact, that every program has a part which cannot be done in parallel. This usually includes the start-up of the program and reading the initial values; the starting of parallelization; the collecting the data by the main thread; and the ending of parallelization. These parts cannot be done in parallel, so no matter how many processors we use, and speed-up the parallelizable part, this part will run the same time. Say 1% of the program is like this, and we use infinite number of processors to speed up the remaining 99%, which will finish so immediately. The running time of the whole program will be 1/100-th of the original time, the running time of the non-parallelizable part, so we gain a total speed-up of 100. No bigger speed-up is possible in this case.

We also should note, that in real problems we need communication between the different threads or processes. The 'cost' of this communication is not independent of the size of the cluster or the supercomputer. Namely, if we use more and more processors the communication will take more and more time. Also, the load balancing between different processors may be unequal causing longer execution time as some processors may be unoccupied for some time. So in real world the problem is even more pronounced, as adding more and more processors the program cannot even reach the theoretical speed-up, but will produce slower running time because of the communication taking more and more time and unbalanced problems becoming more and more unbalanced.

Although not denying the law of Amdahl John L. Gustafson and Edwin H. Barsis noted that the fraction which cannot be parallelized becoming less and less if we increase the size of the problem itself. Because the main interest is always solving problems and not finding theoretical speed-up limits, this means that with bigger computers and computer clusters we can assign them and solve bigger problems because we can achieve greater speed-up in the end. With the case of communication overhead and unbalanced decomposition we can assume mostly the same.

6.4.5 Superlinear speed-up

The theoretical expected speed-up of a parallel program is the number of processors we use. But there can be an exception, where we gain more speed than the added number of processors. This phenomenon is called the super-linear speed-up, which occurs rarely, but not extremely rarely. There may be two causes for such speed-up. One is the accumulation of the cache or memory in the system. If the whole problem cannot fit into memory (or cache) it slows down. But using so many processors that the chunk of the data can fit into the memory (or cache) the problem will speed up extremely. So the speed-up at this very point will be super-linear.

Other possible occurrence of super-linear speed-up may be observed in backtracking algorithms, where the result of one branch may give information to cut other branches [Pro2015].

Chapter 7

Parallelization by disturbing structures

In this section, based on [Szab2017, Szab2018b], we would like to outline a general method of creating subproblems for parallel computation. As we will show some of the already known methods fall into this category. The parallelization methods which use the top levels of the search tree are fall into this category, as several methods described by Szabo in [Szab2011]. We also would like to classify this method and show some possible expansions to it. The most important result of this general method that new practical methods can be implemented by its usage. We will show some of such examples.

7.1 Disturbing structures

One of the most convenient ways to divide a k-clique search problem into subproblems is to partition the node set V of the graph G into two subsets: A and $B = V \setminus A$. Let G_A be the graph that is spanned by the set of nodes A. The goal of our division is to prove that $\omega(G_A) < k$, and we shall find such an A for which this problem can be easier or trivially solved. That is, if we remove the nodes of B from the graph G, the remaining graph will admit to have no k-clique in it and this information is easier to determine than in the original problem. In the original Carraghan–Pardalos algorithm [Carr1990], this separation is made according to the number of nodes. Trivially, if we take out $(k-1)$ nodes from V, the induced subgraph on these nodes cannot contain a k-clique. Thus, we need to remove, also eliminate, the remaining $|V| - (k-1)$ nodes in order to solve the problem. Moreover, this elimination can be done one node at a time, producing an easier problem each time.

More formally, let $V = \{v_1, v_2, \ldots, v_n\}$. We then consider the following subproblems: $V_1 = V, V_2 = V_1 \setminus v_1, V_3 = V_2 \setminus v_2, \ldots, V_{n-k+1} = V_{n-k} \setminus v_{n-k}$. Clearly, solving the k-clique problem in G is equivalent to determining the existence of a $(k-1)$-clique in the induced subgraphs defined by the node sets: $N(v_1) \cap V_1, N(v_2) \cap V_2, \ldots, N(v_{n-k+1}) \cap V_{n-k+1}$ where $N(v_i)$ denotes the neighbor set of v_i. Note, that all these subproblems are independent and can be solved in parallel.

Instead of the size of the graph, tighter bounds for $\omega(G)$ may be employed. The most frequently used bound is some form of greedy coloring algorithm, as in [Brel1979, Culb1992], since computing the chromatic number is impractical for large problems. Given any legal coloring of the nodes, the graph spanned by the subset of nodes determined by any $(k-1)$ color class cannot have a k-clique. Consequently, the separation is done in exactly the same way as before, but considering now the nodes of any partial $k-1$ coloring of the nodes. This method was first described, with a number of minor modifications, in [Bal1986]. In this Subsection we shall analyze the described division of the problem into subproblems and propose some simple algorithms to assist this goal.

7.1.1 k-clique covering node set

The separation method described above can be generalized as follows. Let $G = (V, E)$ be a finite simple graph and let k be a positive integer. Let $W \subseteq V$. If each k-clique in G has at least one node in W, then we call W a k-clique covering node set of G (see Figure 7.1).

Let $W = \{w_1, w_2, \ldots, w_n\}$ be a k-clique covering node set in G. Consider the subgraph H_i of G denote the graph induced by the neighbor set $N(w_i)$ in G for each i, $1 \leq i \leq n$. Let Δ be a k-clique in G. The definition of W states that w_i must be a node of Δ for some i, $1 \leq i \leq n$. Consequently, the subgraph H_i contains exactly $k-1$ nodes of the clique Δ. This observation has a clear intuitive meaning: the problem of determining the existence of a k-clique in G can be reduced to a list of smaller problems of determining the existence of a $(k-1)$-clique in the subgraphs H_i for each i, $1 \leq i \leq n$.

Moreover, the smaller the n, the fewer the subproblems to be analysed become. In other words, starting with a k-clique covering node set with a minimum number of nodes could save computational resources. However, finding an optimal k-clique covering node set is again an NP-hard problem. Clearly it is as hard as the k-clique decision Problem 1.2: a 'yes' in the latter results in a non-empty k-clique covering node set; a 'no' gives an empty cover.

As explained previously, by computing a legal coloring of the nodes of the graph we can parittion the nodes of the graph into two sets: a first set consisting of nodes from the biggest $(k-1)$ color classes, and a second set

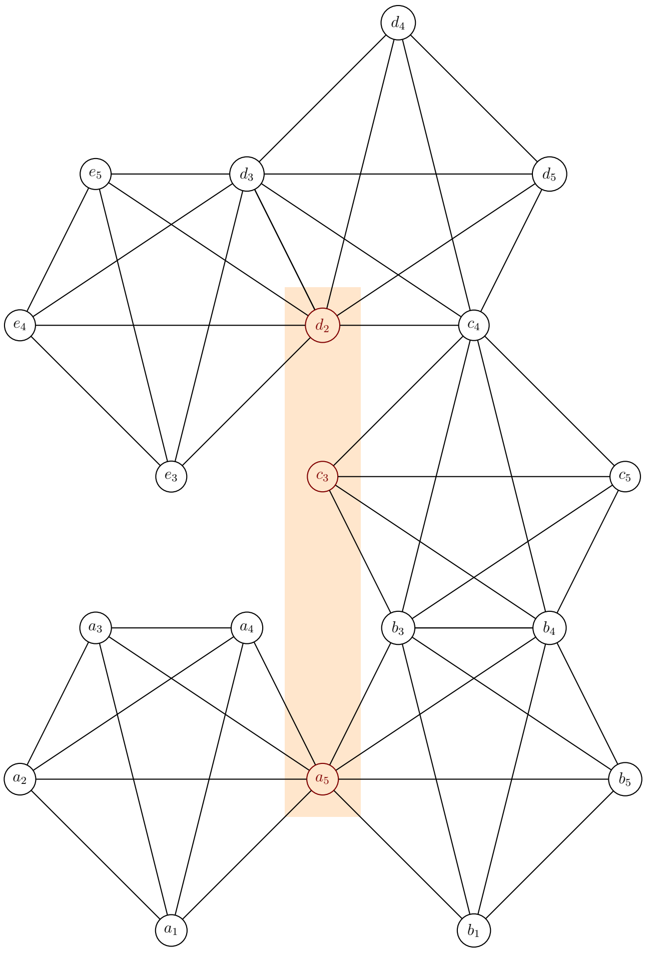


Figure 7.1: A 5-clique covering node set – $\{d_2, c_3, a_5\}$.

with the remaining nodes. It is easy to see that the latter set must be a k-clique covering set, as any k-clique in the graph must have at least one node in this set. Thus we arrive to the partitioning method described in the previous Subsection. Obviously, any other k-clique covering set can be used in the same manner.

7.1.2 k-clique covering s-clique set

More specifically, we are interested in searching for a *k-clique covering edge set*. This is an extension of the node cover described previously to other structures, such as s-cliques. Let $G = (V, E)$ be a finite simple graph and let k be a positive integer. Let F be a subset of all s-cliques in G. If each k-clique in G has at least one s-clique in F, then we call F a *k-clique covering* *s-clique set* of G. In particular, when $s = 2$, then F is an *k-clique covering edge set.* Figure 7.2 depicts an example of an edge covering of all the 5-cliques in a graph, that is, $s = 2$ and $k = 5$.

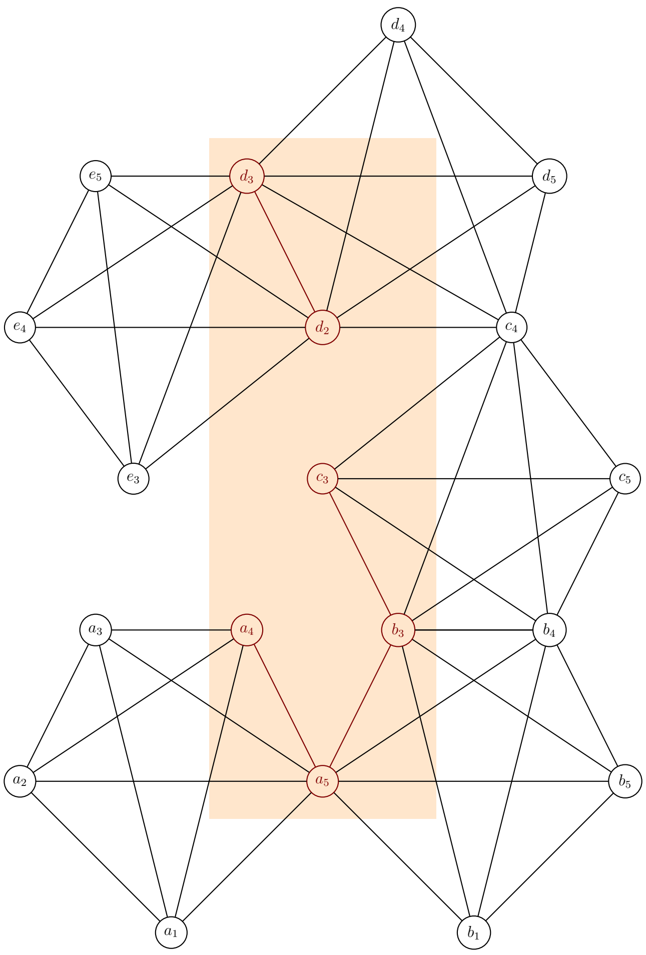


Figure 7.2: A 5-clique covering edge set – $\{\{d_3, d_2\}, \{c_3, b_3\}, \{b_3, a_5\}, \{a_4, a_5\}\}$.

Let F be an s-clique cover of all the k-cliques in G, and let

$$c_i = \{u_{i,1}, u_{i,2}, \ldots, u_{i,s}\}, 1 \leq i \leq |F|$$

be all the s-cliques in F. Also, let H_i be the subgraphs spanned by the sets of nodes

$$H_i = \bigcap_j N(u_{i,j}) \quad 1 \leq j \leq s. \tag{7.1}$$

and let Δ be a k-clique in G. According to the definition of F, there must be a c_i that is an s-clique of Δ for some i, $1 \leq i \leq |F|$. Consequently, the subgraph H_i contains exactly $k-s$ nodes of Δ. This observation has a clear intuitive meaning: the problem of determining the existence of a k-clique in G can be reduced to determining the existence of a $(k-s)$-clique in a series of graphs spanned by each of the subgraphs H_i, $1 \leq i \leq |F|$.

Rather than determining if a $(k-s)$-clique exists in each of the H_i subproblems, it would be preferable to examine the subproblems of finding a $(k-s)$-clique in $G'_i(H'_i, E'_i)$ graphs derived from the previously described general methodology. Specifically, we consider subproblems on the node set $H'_i = \bigcap_j N(u_{i,j})$ with some suitable E'_i edge set that takes into account the sequence of eliminated (examined) problems. For $s = 2$, when s-cliques are edges of the graph, we can easily construct these E_i subsets as follows. Let

$$E_1 = E, E_2 = E_1 \setminus \{u_{1,1}, u_{1,2}\}, E_3 = E_2 \setminus \{u_{2,1}, u_{2,2}\}, \ldots$$

Then let E'_i be E_i on the node set of H'_i. That is we consider only those edges, that have endpoints in this node set. However, for higher values of s we cannot subtract each s-clique from its corresponding V_i or E_i set in the same way. We will return to this issue in Subsection 7.2.

In what follows, we will refer to the k-clique covering node, edge and s-clique sets as *disturbing structures*, because we need to eliminate them to obtain simplified subproblems and to solve the original problem in the end.

7.2 Partitioning the k-clique problem for parallel architectures

In order to design a well-balanced parallel algorithm we now retake the discussion started in Subsection 7.1 concerning disturbing structures. We are interested in finding a set of such structures so as to partition the k-clique problem into several subproblems.

Recall that given any legal coloring of the nodes of a graph, it is possible to choose $(k-1)$ color classes (namely the largest ones) and divide the k-clique problem using the *remaining* nodes. More formally, let $C_1, C_2, \ldots, C_r, (r \geq k)$ be the color classes of a legal coloring of the nodes of $G = (V, E)$. Let

$$V' = C_1 \cup C_2 \cup \cdots \cup C_{k-1} \quad \text{and} \quad V'' = C_k \cup C_{k+1} \cup \cdots \cup C_r.$$

Also let $p = |V''|$ and $V'' = \{v_1, v_2, \ldots, v_p\}$. The set V'' is a k-clique covering node set. To partition the problem space, it suffices to define the subsets of nodes

$$V_1 = V, V_2 = V_1 \setminus v_1, V_3 = V_2 \setminus v_2, \ldots, V_p = V_{p-1} \setminus v_{p-1},$$

which incrementally *eliminate* the nodes of V'' and decrease in size as i increases. With the help of V_i, the k-clique search can be partitioned into finding a $(k-1)$-clique in each one of the subgraphs spanned by the node sets $N(v_1) \cap V_1, N(v_2) \cap V_2, \ldots, N(v_p) \cap V_p$ respectively. These problems are independent and can be solved in parallel.

7.2.1 k-clique covering node set partitioning

It can be easily seen that some of the proposed bounding methods can also be used for the same purpose, with minor changes. The PMAX-SAT method described in Subsection 4.2.6 can be employed in a straightforward manner: we need to determine the largest possible color classes in V', such that the bound is below k. This can be achieved by solving a small integer linear program. The remaining nodes will be the branching set V'', which is a k-clique covering node set.

The s-clique free coloring can be used for partitioning the problem space in much the same way as a legal coloring of the nodes. To note that the bound provided by the coloring is $k - s$.

The b-fold coloring may also be used for such construction with the help of the following simple algorithm. First we count and store the number of nodes inside each color class C_i. We will denote this number by l_i. Then we place the nodes that belong to the smallest C_i in V''. By definition of b-fold coloring, these nodes can be in other color classes as well, so we subtract 1 from the total number of nodes count in each color class C_j, $i \neq j$, that is $l_j = l_j - 1$, for each node shared with C_i. We next choose the smallest color class in $V \setminus C_i$, that is the smallest l_j, and proceed in the same manner. The process is repeated until the number of remaining non-empty color classes falls below $b \times k$. The resulting V'' is a k-clique covering node set.

7.2.2 Partitioning using the Lovász number

Using the Lovasz' theta function described in Subsection 4.2.5 for partitioning the k-clique problem is more computationally demanding than in the previous case, because we need to calculate several theta functions. We start from a subgraph $S \subset G$ which does not contain a k-clique, for example because a legal $k-1$ coloring of the nodes exists for S. Then, at each step of the process, we add a new node $v \in V \setminus S$ and compute the theta bound of the complement of the graph spanned by the node set $S \cup \{v\}$. If this bound is still under k, the node is *accepted* and the partitioning algorithm takes as reference set $S \leftarrow S \cup \{v\}$ in future iterations; else v is not added to S. The procedure ends when all vertices have been examined. The set $V'' = V \setminus S$ is a suitable k-clique covering set.

7.2.3 Partitioning by k-clique covering edge set

As shown in Subsection 7.1, edges, instead of nodes, can also be used as disturbing structures. The edge coloring method described in Subsection 4.2.4 is a straightforward way to determine these edges. We choose the largest $(k(k-1)/2)-1$ color classes, as the graph spanned by the union of the edges in these color classes by themselves cannot form a k-clique. We denote by E' the set of edges inside these color classes. The edges in $E'' = E \setminus E'$, form a k-clique covering edge set. From E'' we can construct the subproblems as described in Subsection 7.1.

Moreover, finding such disturbing edges is not restricted to edge coloring. For example, starting from a legal node coloring we can take the largest $(k-1)$ color classes and then move all the nodes from the other color classes at will into these color classes. We will then arrive to an improper node coloring in the general case, because some nodes in a color class will be adjacent. These *disturbing* edges will form a k-clique covering edge set and can be used to produce subproblems as described above (see [Zav2014b]).

It is easy to see the connection of this method to "subtree of distance 2 from the root" using the term from [McCr2015]. If we put all the nodes into the color class this node has the smallest number of neighbors we will get the same result as by making the parallelization over the top two levels of the search tree constructed the method prescribed in the Chapter 5. But other methods of placing these nodes into color classes will result in different set of subproblems, and some of those may prove better.

7.2.4 Parallelization by s-free quasi coloring

With Szabo we introduced a variation of graph coloring problem, the s-clique free coloring [Szab2012], also see Subsection 4.2.3. In this problem we try to create color classes not edge free as prescribed in the original legal coloring, but s-clique free classes. This method can be used for parallelization the same way as we described in the previous chapters. We can construct quasi s-clique free colorings, where some color classes are not perfect, and may contain cliques of size s or even bigger. Again we may point out some disturbing edges or even s-cliques that constrain us from constructing a legal s-clique free coloring. The parallelization method is the same as described previously.

One method would use the simulated annealing technique, and move nodes from one color class into another while concluding in small number of disturbing structures. Other method would start from an s-clique free coloring of the graph, and placing the nodes from some smallest color classes into bigger ones construct quasi s-clique free color classes.

The actual construction of subproblems also may be done different ways. One method would point out disturbing edges, and construct subproblems using these edges. The parallelization can then be made exactly as in the previous methods.

Other method would use the s-cliques from the quasi s-clique free color classes, and construct the subproblems by constraining the G graph to the common neighborhood of all the nodes from the s-clique. As we pointed out in the introduction of this chapter this method needs taboo listing to be effective, as we cannot easily delete these s-cliques from the adjacency matrix.

7.2.5 Parallelization by quasi coloring

This parallelization technique was introduced by Szabo in [Szab2011]. We try to color the nodes of a graph by $k-1$ colors. If we can do so, there cannot be any k-cliques in the graph, so we could solve the problem of k-clique in one single step. If we cannot do a $k-1$ coloring we produce a quasi $k-1$ coloring, which means there will be some disturbing edges in the color classes. The proposed algorithm by Szabo is a simulated annealing technique. We move nodes from one color class into another if the by this move we can reduce the number of disturbing edges. As this method is strictly monotonic in therms of the number of disturbing edges it will stop. The actual algorithm proved to be very fast.

The parallelization technique is the following. We take the disturbing edges one by one, and inspect, whether they can be an edge of a k-clique or not. We do this step as follows. Let's note the disturbing edge as $\{u, v\} \in E$. We construct the common neighborhood of this edge, thus $N(u) \cap N(v)$, and search for a $(k-2)$-clique in the resulting graph. If we find one, then we found the answer to the original question, because the resulting $(k-2)$-clique can be extended by u and v to produce a k-clique in the G graph. If no such clique present in the subproblem, then this edge can be deleted from the graph. As inspection of the edges are independent problems this leads us to a good parallel technique.

7.3 Increasing and modifying the subproblems

To achieve an even load in large scale parallelization, more subproblems than the number of available processing cores are required. A reasonable ratio ranges between 3 and 10$\times$ more subproblems than processing units. The proposed k-clique covering node set will not meet this constraint in the general case: their size frequently range between 30 and 100 nodes, while a modern supercomputer may contain between 1000 and 100 000 cores. Note that the k-clique covering edge set is more appropriate in this case, as it typically generates from 500 to 2000 subproblems.

To overcome this limitation, we propose the following strategy. Let v, u be the first two nodes of one such k-clique covering set, such that $u \in N(v)$, that is $\{u, v\}$ an edge. The corresponding two subproblems following the general partition methodology are searching for a $(k-1)$-clique in each of the two subgraphs spanned by the node sets $N(v) \cap V$ and $N(u) \cap (V \setminus v)$. We propose to construct new subproblems using the edge $\{v, u\}$. Namely, we search for a $(k-2)$-clique in the graph spanned by node set $N(\{v, u\}) \cap V$, and search for a $(k-1)$-clique in each of the subgraphs spanned by node sets $N(v) \cap (V \setminus u)$ and $N(u) \cap (V \setminus v)$ respectively. Clearly solving these latter subproblems is equivalent to solving the former ones. This method may be trivially extended to all the other edges in the k-clique covering node set, and thus increment the number of subproblems by an order of magnitude.

We further discuss a problem that concerns the elimination of disturbing structures, once the corresponding subproblem has been examined, for the k-clique covering s-clique set strategy, where $s > 2$. In this case, it is not possible to directly remove the *disturbing* s-cliques from the node sets $V_i \subseteq V$, or edge sets $E_i \subseteq E$, where each i corresponds to an s-clique element of the cover. A possible solution is to consider additional edge subproblems instead of s-cliques. For $s = 3$ it works as follows. Let Δ be a 3-clique in the given k-clique covering 3-clique set, with nodes x, y, z and edges $e = \{x, y\}, f = \{y, z\}, g = \{x, z\}$. After failing to find a $(k-3)$-clique in the graph spanned by the node set $N(\Delta)$, we cannot delete it from the graph directly. The situation changes if it is possible to prove that there are no $(k-2)$-cliques in the graphs spanned by the node sets $N(e) \cap (V \setminus z), N(f) \cap (V \setminus x)$ and $N(g) \cap (V \setminus y)$ respectively. It is easy to see that by examining these additional subproblems, the nodes in Δ can be removed in all ensuing subproblems without losing completeness. If all the s-cliques of the cover are divided in this way, the general elimination methodology can be extended to these edge-based structures (see Subsection 7.1).

7.3.1 Refinement by usage of edge weights

We can construct the quasi coloring classes in several different ways. Note that the resulting parallel problems may be quite different depending on the subproblems represented by the disturbing edges.

This recognition leads us to a variation of the previously described algorithm. We can inspect the edges prior to the quasi coloring algorithm, and by some heuristics guess the hardness of the subproblem represented by them. With this information we can construct a different quasi coloring which results in easier problems of bigger set. The algorithm runs as follows. All edges assigned a proportional weight which represent the time that the subproblem defined by this edge would take us to solve. Then we follow the previously described simulated annealing technique. Only this time the objective function should be the sum of the weights of the disturbing edges and not the mere number of these edges. So we move a node from one quasi color class into another one if by this move the sum of weights over the disturbing edges reduced. Again, it is easy to see, that this method is strictly monotonic, and will stop.

The gain of this method is that even if we would have more subproblems in the end they may be solved in shorter time altogether. The second gain may come from the multitude of the subproblems, as we use many cores and it is desirable to have more problems than few.

Chapter 8

The Las Vegas method for parallelization

One goal of the present work is to find parallel, and even massively parallel algorithms for the k-clique problem. In this chapter, based on the theory detailed in the previous chapter, we shall detail such algorithms and evaluate it using numerical experiments. The chapter is based on previous results [Zav2015]. As the computations in question used 512 cores of a supercomputer – the Sisu computer in Finland –, we did not repeat the tests but used our previous results. This means that the sequential program inside the parallel one was a previous version of our program detailed in Chapter 5, and thus the results for one core run cannot be directly compared to the running times of the new program version. The main message though is the speed-up, which is independent from the sequential program version.

We should point out one more effect of discrete optimization problems. The usual Branch-and-Bound method is sensitive to the sequence of subproblems in a branch. This was shown for SAT problems [Ouy1998], and could be shown for clique search problems as well. This effect will be used by my algorithm, as it eventually finds a better sequence for solution, and thus reduces the search space. Those are the cases where the solution of a subproblem alters the problem space of another one.

Another important effect is based on restarting the same subproblems. This results of changed run time, as the same subproblem starts from different starting points, thus may have a better preprocessing optimization [Moor2004]. Also this method can help to employ the free processors. If the original subproblem would finish early, then we have the advantage of running two or more instances of the same subproblem, which can cause that one would finish earlier then the other.

8.1 Implementation of a massively parallel algorithm

The Las Vegas algorithms, first described by Babai [Bab1979], is a variation of the Monte Carlo randomized algorithm. Formally, we call an algorithm a Las Vegas algorithm if for a given problem instance the algorithm terminates returning a solution, and this solution is guaranteed to be a correct solution; and for any given problem instance, the run-time of the algorithm applied to this problem is a random variable.

The variance in the running time of a Las Vegas algorithm led Truchet, Richoux and Codognet to implement an interesting way of parallelization of the heuristic algorithms for some NP-complete discrete optimization problems [Tru2012]. The authors note that the algorithm implementation for those problems heavily depends on the "starting point" of the algorithm, as it starts from a random incorrect solution and constantly changes it to find a real solution. Depending on the incorrect starting solution the convergence of the algorithm may be very fast or slow. The idea behind the Las Vegas parallel algorithm was to start several instances of the sequential algorithm from different starting points and let them run independently. The first instance that finds the solution shuts down all the other instances and the parallel algorithm terminates. As the running time of the different instances vary, some will terminate faster, thus ending the procedure in shorter time. The article describes the connection of the variance of the running times and the possible speedup when using k instances and found that for some problems a linear speedup could be achieved.

We may use these results in another way. We can alter the sequence of the subproblems not knowing their complexity in prior, but in the course of the computations.

8.1.1 Parallel Las Vegas algorithms

We choose, as an example algorithm, a parallel algorithm for k-clique problem proposed in [Szab2011]. The basic step of this algorithm is the removing of an edge from the graph. Given an edge v_i, v_j. If one can prove that this edge is not part of any k-clique, then this edge can be freely deleted from the graph without altering the answer to the k-clique question. The proof takes the subgraph of G spanned by the nodes $N(v_i) \cap N(v_j)$, and examines whether there is a $(k-2)$-clique in it. ($N(v)$ denotes the neighboring nodes of node v.) If the answer is 'Yes', then we found a 'Yes' answer for the original question. If the answer is 'No', then we can delete this edge, as it cannot be an edge of a k-clique.

The algorithm uses the concept of disturbing edges. Given some set of edges, if we can delete them, then the original question can be answered by 'No', and this answer is trivial. The actual algorithm enumerates disturbing edges by a quasi coloring, and by deleting these edges we get a graph which can be colored by $k-1$ colors, thus there can be no k-clique in it. Actually this method of reducing the graph by taking the neighbors of *two* nodes is closely related to the two level branching detailed in [Pro2015].

The subproblems, which can be denoted by an edge, are independent, but overlapping. In order to eliminate the overlapping parts let us consider a fixed sequence of the disturbing edges. Should we solve the problems in a sequential manner, then after solving the first one we can delete this edge from all the other problems to be solved. And so on for all the edges, one by one. The resulting problems are free of overlapping. Because these problems can still be solved independently, we can run them parallelly, deleting these edges *a priory* even before solving the specific problem.

We propose two other methods. First, which we call the Las Vegas edge deletion, starts the original overlapping problems without any edge deletion parallelly. The problems will be of different complexity, thus some of them will run fast, and others slowly. If one problem (a fast one) finishes, then the edge denoted by this problem shall be deleted from all the other problems *in the course they are being solved.* (Obviously this can be done only if the sequential program can allow this.) This method will do some surplus calculation because of overlapping search spaces, but can profit by reordering the sequence of edge deletion. As we already pointed out previously the sequence in which the problems are solved can affect the size of the search space. The structure of the program:

Master	**Slave**
Get report / get request / asked for deleted edges	Report / request for new task (edge)
IF found: exit IF not found: delete the edge IF asked: give deleted edges IF requested: give new task (edge)	Construct the task from edge Solve the task (repeatedly ask for deleted edges)

Second version, which we call the Las Vegas edge deletion with restart, do the same as the previous one, but uses an other technique, as well. If no work left to be given out (the number of subproblems fall below the number of processors), then we give out an edge, which is already given out to another process, and so two threads do the same calculation. It may seem redundant, but the restarting process can start from scratch, and with some already deleted edges, can produce better preconditioning of the subproblem. This method is well known in SAT solving community, but my proposal is somehow different, as given spare processors we run the original, half solved subproblem, along with the newly started. On the other hand, if the already long-running solver is near to the finish, we do not need to throw it out.

As it will be seen from the evaluation, both methods are really strong. Also, there is an interesting effect of reproducibility. While different runs will delete edges in different order, one can save the actual order of a given run. It is clear, that given this special order of edges we can run the original *a priory* edge deletion algorithm with the same, or even better speed.

We would like to summarize the three approaches:

1. The *a priory* deletion sequence, where we list edges in a fixed way, and delete the edges representing an earlier (in the sequence) subproblem from a later (in the sequence) problem.

2. The Las Vegas method, where we do not delete any edge from the subproblems. We start the solvers in parallel manner, and if a subproblem finished we delete the edge representing this subproblem from all subproblems – either waiting for starting to be solved either if a solver presently working on it.

3. The Las Vegas method with restarting is at first the same as the previous one. The difference comes into play at the very end, when we solved most of the subproblems, and the remaining (running) subproblems are less then the number of processing elements. In this case we restart a subproblem – keeping the same subproblem being solved at the same time –, so there will be two instances of solvers solving the same subproblem. One have the advantage of being half through solving it, the other have the advantage of being restarted, and so having a better preconditioning.

From theoretical point of view we can compare the proposed Las Vegas parallelization to the work share – work steal methodology. As in our case we have a fixed distribution it does not fall into the second category. But because the work to be done changes dynamically it is not in the first category either. We call this approach "work help", as an early finishing subproblem helps other solvers to solve their job. Note, that this approach connected to the technique of parallel blackboard system [Clea1991, Clea1992].

8.1.2 Other possible usage

The method described in this section can be used for parallelization of different combinatorial optimization problems such as SAT, Integer Linear Programming or Constraint Programming. In order to utilize the algorithm it is important, that the original problem can be divided into several independent (and actually overlapping) subproblems. Also, the result of a subproblem should be usable to help other subproblems – it will reduce the search tree by the overlapping part.

The usage can be done in three different ways. First, the "help" can be given *a priory* in a chosen sequence – in all cases we can add additional constrains to the problem description. In fact, this is the usual method for parallelizing discrete optimization problems. Second, start the independent subproblems, and if one finished give this information to all the other threads. This is my method of Las Vegas edge deletion. Third, if some threads have no work to do, than restart a subproblem, but with the information gained from the previous results. The third algorithm uses this technique together with the edge deletion.

8.1.3 Tests

We programmed the described algorithms in `c++` and MPI, and performed several tests on different graph sets. Table 8.1 below summarize the running times for sequential and different parallel algorithms for $k = \omega + 1$. The columns labeled as follows. "N" denotes the size of the graph; "%" denotes the edge density; "ω" denotes the size of the maximum clique. The "parts" indicates the number of disturbing edges, thus the number of different subproblems that will be started. The rest of the columns denote the number of processors we used, where "1" means the sequential run time. The "seq" denotes the original algorithm where the edges deleted in the given sequential order, *a priori.* The "lv" denotes the Las Vegas method where the edge deletion is performed after a subproblem was solved. The "rest" denotes that, apart from the Las Vegas edge deletion, the problems also restarted when free processors were available, so the same problem run on several processors. The running times are in seconds, and for really big figures we used "k" for denoting thousand. All tests, including the sequential runs, were performed on the same supercomputer.

The first set consists of random graphs, the second set is the DIMACS set of graph problems [Hass1993], the third set is graphs of hard problems of monotonic matrices [Szab2013, Öst2019] and of deletion code [Sloan]. The data presented here is partial, as we left only those instances, where the run-times are big enough to be of any interest.

The time limit for sequential and 16 processor runs was 12 hours, while for 64 and 512 processor runs 72 hours. The symbol "*" denotes run times exceeding the time limit, and "-" means that we have not run the test.

	N	%	ω	parts	1	16	16	64	64	64	512	512	512
						seq	lv	seq	lv	rest	seq	lv	rest
rand 200p9	200	90	40	152	480	27	35	22	30	20	22	30	32
rand 300p8	300	80	29	540	754	41	45	13	16	17	12	19	19
rand 300p9	300	90	47	341	*	*	*	*	*	*	*	16k	14k
rand 500p6	500	60	17	2478	48	9	9	2	2	2	0	0	0
rand 500p7	500	70	22	2231	3069	167	179	40	45	46	9	15	15
rand 500p8	500	80	32	1664	*	*	*	15k	17k	17k	5703	3543	3377
rand 800p5	800	50	14	7296	71	26	26	6	7	7	1	1	1
rand 800p6	800	60	19	6345	2371	158	166	38	41	41	5	6	6
rand 800p7	800	70	25	5587	*	*	*	5999	6571	6578	830	968	967
rand 900p5	900	50	15	8729	132	40	41	10	10	10	3	2	2
rand 900p6	900	60	19	8215	6493	398	423	96	103	104	13	14	15
rand 1000p5	1000	50	15	10955	282	65	66	16	16	16	2	2	2
rand 1000p6	1000	60	20	9823	14k	798	847	192	206	207	25	30	29
brock800_3	800	65	25	4888	6383	357	373	86	91	91	12	14	15
brock800_4	800	65	26	4592	4899	280	291	68	71	71	9	11	11
latin_sq_10	900	76	90	380	4053	91	121	40	61	59	40	55	54
keller5	776	75	27	420	4071	369	382	118	198	166	115	114	133
sanr200_0.9	200	90	42	128	297	15	25	14	33	33	14	29	28
sanr400_0.7	400	70	21	1408	351	22	23	5	6	6	1	2	2
p_hat700-2	700	50	45	826	740	36	59	27	48	44	27	82	84
p_hat300-3	300	74	36	297	198	9	15	7	14	14	7	15	15
p_hat500-3	500	75	50	657	*	*	*	9645	6020	4960	9653	4328	2253
monoton-8	512	82	23	590	2123	367	266	367	165	145	367	154	114
monoton-9	729	84	28	932	*	-	-	137k	39k	23k	137k	27k	6377
deletion-9	512	93	52	375	*	-	-	-	-	-	*	*	67k

Table 8.1: Test runs

	N	%	ω	parts	1	64	64	64	512	512	512
						seq	lv	rest	seq	lv	rest
monoton-9	729	84	28	932	~1150k	137k	39k	23k	137k	27k	6377
hours:					~320h	38h	11h	6h	38h	8h	2h
speed-up:					1x	8x	30x	50x	8x	43x	180x
average run:									1232	1717	909
minimum run:									1	11	3
maximum run:									137k	27k	5k

Table 8.2: Test runs for `monoton-9`

8.1.4 Evaluation

The test runs lead to several conclusions. First, it is clear, that the original idea of disturbing edges by Sandor Szabo enables quite good parallel speedups even for large number of cores. Second, for some (harder) problems there is a limit for the original algorithm. For other problems, such as random graphs, the speedup is nearly linear. Actually, in my opinion, this indicates not as much the goodness of the algorithm, as it is rather shows the problem of testing with random graphs.

The Las Vegas methods are also performing well, and while being a bit behind for smaller and easier problems, they make a big difference for bigger and harder problems. Actually the lower performance for the easy problems are less interesting: one would not use a supercomputer for those problems! The difference between the simpler edge deletion Las Vegas algorithm and the restarting algorithm is similar. For very small problems the second can be a little slower – it takes time to shut down the threads that are not needed. For most problems they run for the same time, and for really hard problems the restarting version makes one more huge leap in performance.

We would point out the problems of `monoton-9` and `deletion-9`. These problems are extremely hard, and only few achieved solving them with the aid of reducing the problem by finding symmetries, and in the second case, with the aid of semi definite programming. My method is using none of these.

Details of one problem.

Let us take a close look at one special problem, the `monoton-9`. Table 8.2 presents the `monoton-9` problem alone, and we indicated also the running times in hours, the speedup and the average run time for the subproblems as well. The sequential run time is calculated by summing up the run times of the 512-seq subproblems. This is obviously not the same, as we would run the sequential program, but it is as if the *a priory* edge deleted problems would be run in sequential order. This figure, in my opinion, should be close enough to the real one.

Evaluating the results, one can clearly see, that the *a priory* edge deletion method is dominated by one subproblem, and that is why it cannot sped up using more processors. The edge deletion Las Vegas method is much better, but it also has some limits of speedup. The restarting Las Vegas method on the other hand scales very well. A scale up is usually considered good if by doubling the cores the running time reduced by a factor of 1.5. In case of this hard combinatorial optimization problem we could achieve a better scale up using 512 processors. This indicates that the method possibly can be scaled up for several thousands processors.

Also, the average run times of the subproblems are quite interesting. The edge deletion Las Vegas method has a bigger average run time than the *a priory* edge deletion – and one would expect this –, as this method is certainly making more calculations in order to eliminate the dominating subproblem. More interesting is that the average time of the restarting algorithm is less. That indicates, that this method even possibly can achieve super linear speedups in some lucky problems one day. Obviously then the whole question of super linearity should be examined and rethinked, because it depends on which sequential algorithm we compare it to. There certainly will be a faster sequential algorithm, but we cannot find it without using the Las Vegas randomization method.

Finally, we would present the graphs of actual running times of the subproblems. In the first graph on Figure 8.1 we reordered the problems by the magnitude of the times. Be aware, that the time scale is logarithmic, so the actual differences are of several magnitudes. In this graph one can see, that all three algorithms are dominated by the longest subproblem, although the restarting Las Vegas can smooth out this problem the best, reducing the variance of running times by more then 2 magnitudes.

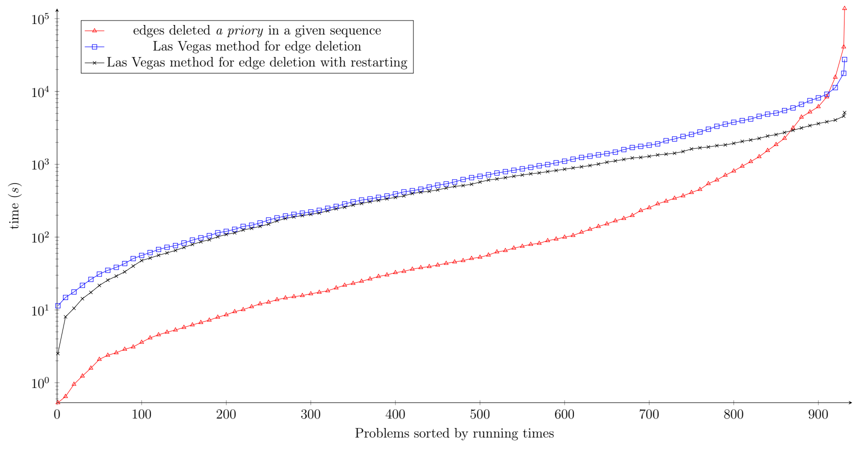


Figure 8.1: The sorted running times of the `monoton-9` subproblems.

The second graph on Figure 8.2 is the running times sorted by finishing times. To the left the time passes while we run the parallel algorithm, and the finished subproblem times denoted on the x axis. The graph is smoothed, to have less 'noise' of big variance of running times. It could clearly be seen, that the restarting Las Vegas algorithm helps at the very end by reducing the dominating problems exactly where it needs it the most.

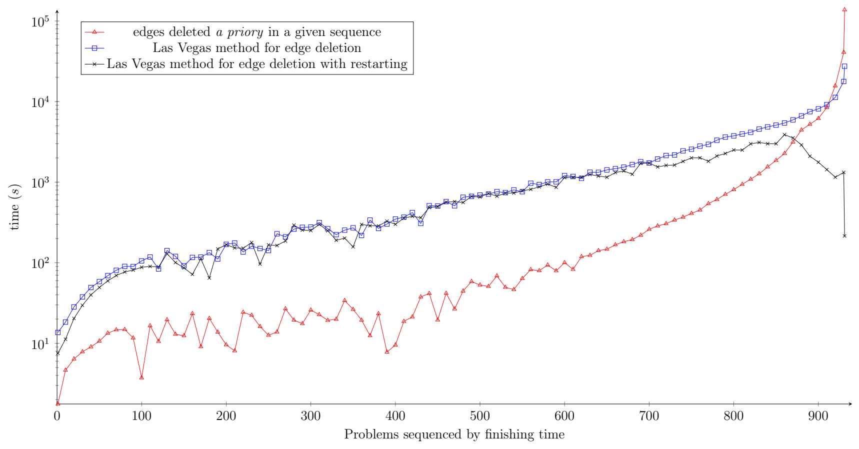


Figure 8.2: The time sequence of running times of the `monoton-9` subproblems.

8.2 Further possible usage

The Las Vegas approach detailed in this chapter shows not only its usefulness but also its potential for further development. Backed up with the idea that different subproblems may be used as work-help to solve the whole problem we can construct different methods that can exploit this approach. The actual implementation and evaluation of these methods lay outside of the scope of our work.

8.2.1 Anything goes

With this method some interesting variations can be implemented mostly because there is an uncertainty which edge will be disturbing. This may lead us to a method where we start to solve more problems than desirable.

First proposed algorithm goes as follows. We consider *all* possible subproblems defines by nodes. That is we take each node v in the graph, compute the neighborhood $N(v)$ and take the subgraph spanned by these nodes. We start all subproblems at once – or as much as we have processing elements. If any calculation finishes we delete that node from all other subproblems. This way we do not need to build a branching set, and guessing which node is representing an easier problem just done by parallel run. After a while the problems become trivial and the algorithm will finish. Or one can run periodically a sub procedure, for example a coloring, to check if the remaining graph still can have the prescribed k-clique or not.

The proposed method trivially can be extended to use edges instead of nodes, that is using the neighborhood $N(\{u, v\})$ for each edge. Maybe a full algorithm to wait for each subproblem represented by an edge would not be feasible, but we can use this method for some preconditioning. This means if we stop after some edges were deleted the result will be an easier graph instance. As the number of edges would be much bigger then the number of processing elements, one may consider to use a time limit for calculation a certain instance. This would mean, that we skip the hard subproblems but finish with the easy ones helping to reduce the whole instance.

8.2.2 Combined with disturbing structures

The previous idea may be combined with the idea of disturbing structures as well.

Construct a sequence of disturbing edges by a proposed method. Start the Las Vegas parallel algorithm on the subproblems defined by these disturbing edges. After some time limit stop the work. Some edges will be deleted, some will remain. We call the remaining edges are un-nice, since they represent hard problems.

Run the method for constructing disturbing edges again. But this time modify the procedure, and search for a sequence of disturbing edges that contain as little as possible un-nice edges. Start the Las Vegas parallel algorithm again. We expect the constructed series of subproblems to be easier each time we restart the procedure. At the end the problem will became so easy, that it will finish in the prescribed time limit.

For example if we constructing the sequence of disturbing edges using a coloring then we try to put nodes from the small color classes into bigger ones. So if one insertion proves to be hard, then we can construct different disturbing edges by trying to insert such a node into another color class which will produce different disturbing edges.

8.2.3 Las Vegas search for disturbing structures

We can directly use the Las Vegas idea for constructing the disturbing structures themselves. This is done the following way.

Make a legal coloring. Take the nodes in the smallest $c - (k - 1)$ color classes. We would like to put them into the bigger $k - 1$ classes, but there are disturbing edges for any class.

Start the k-clique search program for *all* disturbing edges of a node at once. That is for each edge this node would produce in any color class if it would be put there. One will stop early as there is no $(k - 2)$-clique in the neighborhood of that edge. Delete that edge. If for a node and a given color class all disturbing edges were deleted we can put the node into that color class – and stop all other instances.

This approach would help us in two ways. First, it would amplify the number of subproblems. Sometimes the disturbing method gives us much less problems as the number of processors we have, so amplifying them means that we can use all processing elements. Second, it would be quite hard to predict which disturbing edge concerning one node is the easiest one. And we cannot know the best elimination sequence of those disturbing edges. The Las Vegas method solves both unknowns at the same time.

Chapter 9

Summary and conclusions

The aim of this work was to synthesize knowledge about the k-clique problem. We detailed the modeling expressivity by showing several problems that can be modeled and solved by k-clique and maximum clique search. We detailed the up-to-date methods of solving this problem, and discussed the possibility of parallelization of it, which opens the possibility of using supercomputers to solve it.

9.1 Theses

9.1.1 1st thesis

In our work we showed that several problems can be modeled by graphs and solved using a k-clique solver. These problems arise from various fields. First, we showed reformulation for Latin square, Sudoku game, the problem of non attacking queens, Costas Arrays and combinatorial problems arising from coding theory. Second, we detailed a group of real life problems that connected to subgraph isomorphism, like protein docking, molecule search, fingerprint and image recognition. Third, we detailed the reformulation for scheduling problems, namely open shop, flow shop and job shop problems. Finally, we described how clique search can be of use in network analysis using market graphs or brain graphs.

In a few cases we did also some numerical experiments. The goal in that was not to beat other methods, but to show that these reformulations can solve non trivial problems of these kind in comparable manner then other methods and solvers.

9.1.2 2nd thesis

We choose one problem class for extended demonstration of modeling power of graphs and k-cliques, namely various graph coloring problems. We detailed reformulation for the problem of k coloring of the nodes of a graph, 3-clique free coloring and coloring of hypergraphs. With the aid of the last one we could solve some hard hypergraph coloring problem and aid an open question by Voloshin. The question asks about colorability of hypergraph having C edges – nodes cannot receive all different colors –, and D edges – nodes cannot receive all the same color. The construction is interesting, because there are hypergraphs that cannot be colored at all. We performed a series of calculation from witch one can conclude about probability of edge C and D where this transition from colorable to uncolorable happens.

9.1.3 3rd thesis

After listing different sequential maximum clique solvers we look for the most helpful auxiliary algorithms that can help us to solve these problems. First, we compared algorithms that produce upper bounds on clique size, and as such can be used inside a maximum or k-clique solver as a bounding or cutting function. Our measurements help to see how good is the produced bound and how long it takes to calculate it. Second, we showed some kernelization steps that can help in reducing the size of the graph.

9.1.4 4th thesis

We designed and implemented a specialized k-clique search program, and detailed its building blocks. Using this program we built a maximum clique solver, which could be compared to other state-of-the art solvers. Our approach proved successful, and we could show that our program is among the best ones.

9.1.5 5th thesis

After detailing the background on parallelization we introduced the concept of disturbing structures. This concept helped us to build the sequential k-clique search algorithm. It also gives a method on parallelizing algorithms for this problem.

9.1.6 6th thesis

Based on the previous concept we implemented a massively parallel program for k-clique search. We examined the importance of the deletion sequence, that is the ordering of subproblems. Driven by this we constructed the so called Las Vegas method of parallelization, which helps us find a better ordering and thus reduce the search place. Among other data we also could show that the running times for different subproblems may differ in 4 magnitudes, and this is exactly where the Las Vegas method and solver instance restarting helps.

9.2 Future work

A work like the present one is never finished. There are and will be still open questions, algorithms to be implemented and compared. In the future we would like to concentrate on further development of the sequential k-clique program. We would like to include subchromatic coloring methods like b-fold coloring.

We also would like to implement parallelization techniques detailed in Chapter 7.

As we showed ordering by increasing subproblem hardness can lead to very efficient program. We would like to implement an algorithm which estimates the size of the backtrack tree [Knu1974, Purd1977, Kilb2006, Mar1998, Corn2006]. This can prove useful in ordering the nodes of the branching, and also in parallelization.

9.3 Author's own results

The question that may be open at this point to the reader is that which of the results in the present work are considered common knowledge or results from other researchers, and which of these are the achievements of the author. So we shall list my own results.

1. The formulation of the Costas Array problem by graphs in Section 2.1.4 is my own result. It is not yet published.

2. The formulation of the flow shop, open shop and job shop problems in Section 2.3 is the joint work with Sandor Szabo. It is not yet published.

3. The k-clique approach of the 3-free coloring in Section 3.2 is my own result. It was published in [Szab2016b].

4. The k-clique approach to hypergraph coloring including its application to Voloshin's problem Section 3.3 is my own result. It was published in [Szab2019b].

5. Measurements and comparison of different upper bounds in Section 4.2, including sequential and parallel implementation of DSatur and Iterated Coloring algorithms, and also its application to edge and b-fold coloring is my own result. It was published in [Szab2017, Marg2019].

6. The sequential k-clique program in Chapter 5 was first developed by Sandor Szabo and later extendedly developed by myself including addition of Culberson's Iterative Coloring, node rearrangement, parallelization by bitsets and transformation to a maximum clique search algorithm. It was published in [Szab2018a].

7. The theoretical background of parallelization which we call "disturbing structures" in Chapter 7 is mostly my own result. It was published in [Szab2018b].

8. The parallel algorithm and program detailed in Chapter 8 is based on the idea from Sandor Szabo, while the Las Vegas approach and the measurements proving its effectiveness was done by myself. It was published in [Zav2014a, Zav2014b, Zav2015].

Appendix A

Összefoglaló

A jelen munka célja, hogy szintetizálja a k-klikk keresés tudományos kutatásának eredményét. Bemutattuk, hogy a gráfokkal való modellezés, és ezekben a gráfokban k-klikk illetve maximum klikk keresés milyen módon tud modellezni legkülönfélébb problémákat, azaz bemutattuk a k-klikk keresés modellező erejét. Bemutattuk a maximum klikk keresés legjobb programjait és azok hátterét, illetve bemutattuk a probléma párhuzamosíthatóságát. Ezzel megnyílt a lehetőség, hogy superszámítógépet is használhassunk ezen problémák megoldásához.

A.1. Tézisek

A.1.1. Első tézis

Bemutattunk problémákat, melyek hatékonyan fogalmazhatók meg mint klikk-keresési feladat. Többek között a latin négyzetek és sodoku, a sok királynő feladat, Costas téblázatok és kódelmélet témaköréből. Bemutattuk, hogy a részgráf izomorfizmus is visszevezethető klikk feladatra, és rámutattunk, hogy a kémia és alakfelismerés témakörének feladatai így megoldhatóak. Bemutattuk, hogy ütemezési feladatokat is meg tudunk fogalmazni klikk feladatként, és hogy ezek hatékonyan is számolhatóak.

A.1.2. Második tézis

Kiválasztottunk a problémáknak egy osztályát, a gráfok és hipergráfok színezésének problémáját, és részletesen bemutattuk, hogy miképp lehet ezeket ugyancsak viszavezetni klikkeresésre. Az átfogalmazás segítségével meg tudtunk támogatni egy nyitott kérdést, melyet Volosin tett fel a hipergráfok színezhetősége kapcsán.

A.1.3. Harmadik tézis

Miután bemutattuk, hogy milyen klikk-keresők vannak, ezek segédalgoritmusait is bemutattuk. Külön részleteztük, hogy milyen felső becslők vannak a klikkszámra, és ezek eredményeit – mind jóság, mind az algoritmus tapasztalati hatékonysága tekintetében – mérések segítségével összehasonlítottuk.

A.1.4. Negyedik tézis

Megterveztünk és implementáltunk a saját k-klikk kereső programunkat, illetve részleteztük annak hátterét. A program segítségével maximum klikk keresőt is implementáltunk, melyet mérésekkel összehasonlítottunk más, modern klikkeresőkkel, melynek alapján kijelenthetjük, hogy a klikkeresőnk a legjobbak között van.

A.1.5. Ötödik tézis

Miután bemutattuk a párhuzamosítás problémakörét bevezettük a zavaró struktúrák elvét. Ezen elv segítségével elméleti szinten tudtuk megalapozni a párhuzamos algoritmusainkat.

A.1.6. Hatodik tézis

A fenti elvek alapján implementáltunk egy nagyléptékben párhuzamos k-klikk kereső algoritmust. Megvizsgáltuk a részproblémák sorrendjének jelentőségét. Ezen alapulva bevezettük a Las Vegas párhuzamosítási elvet, amely jelentősen csökkentette a keresőteret.

A.2. A szerző saját eredményei

1. A Costas táblázat átfogalmazása a 2.1.4 fejezetben saját eredmény. Még nem publikált.
2. Az ütemezési feladatok átfogalmazása a 2.3 fejezetben közös eredmény Szabó Sándorral. Még nem publikált.
3. A 3-free színezés k-klikk megközelítése a 3.2 fejezetben saját eredmény. A [Szab2016b] lett publikálva.
4. A hipergráfok színezés k-klikk megközelítése, illetve Voloshin által felvetett problémának a számolása a 3.3 fejezetben saját eredmény. A [Szab2019b] lett publikálva.

5. A klikk méretének felső becsléseinek összehasonlítása a 4.2 fejezetben, beleértve a soros és párhuzamos DSatur és ismételt színező algoritmusok implementálását, illetve az él és b-fold színezésekhez való használata saját eredmény. A [Szab2017, Marg2019] lett publikálva.

6. A 5 fejezetben bemutatott soros k-klikk keresőt először Szabó Sándor fejlesztette ki, de később a szerző jelentősen kibővítette az ismételt színező, csúcsátrendezés illetve bitsetekkel való párhuzamosítás fejlesztésével. Továbbá a szerző szerkesztette ennek segítségével a maximum klikk kereső algoritmust és programot. A [Szab2018a] lett publikálva.

7. A „zavaró struktúrák" elméleti háttere és az ezen alapuló párhuzamosítási elvek a 7 fejezetben főképp a szerző eredménye. A [Szab2018b] lett publikálva.

8. A 8 fejezetben részletezett párhuzamos program Szabó Sándor ötletén alapulva a szerző fejlesztése. A Las Vegas párhuzamosítási elv, illetve a mérések, melyek a megközelítést megalapozták saját eredmény. A [Zav2014a, Zav2014b, Zav2015] lett publikálva.

Appendix B

Publications related to this thesis

Book chapters

1. Margenov, S., Rauber, T., Atanassov, E., Almeida, F., Blanco, V., Ciegis, R., Cabrera, A. Frasheri, N., Harizanov, S., Kriauzien, R., Rünger, G., San Segundo, P., Starikovicius, A., Szabo S. and **Zavalnij** B. *Applications for ultrascale systems.* In: Ultrascale Computing Systems. Eds.: Carretero, J., Jeannot, E., Zomaya, A.Y. Institution of Engineering and Technology (IET), London, (2019) pp. 189–244.

2. Varady G. and **Zavalnij** B. *Introduction to MPI by examples.* TypoTEX. 2013. 304 p. ISBN: 978 963 279 378 8

Journal papers

3. Szabo S. and **Zavalnij** B. "Benchmark Problems for Exhaustive Exact Maximum Clique Search Algorithms." *Informatica (Ljubljana).* 43 : 2 (2019) pp. 177–186.

4. Szabo S. and **Zavalnij** B. "Reducing hypergraph coloring to clique search." *Discrete Applied Mathematics.* 264. (2019) pp. 196–207.

5. Szabo S. and **Zavalnij** B. "Decomposing clique search problems into smaller instances based on node and edge colorings." *Discrete Applied Mathematics.* 242 (2018) pp. 118–129.

6. Szabo S. and **Zavalnij** B. "The gulf between the clique number and its upper estimate provided by fractional coloring of the nodes." *Serdica Mathematical Journal.* 43:2 pp. (2017) 111–126.,

7. Szabo S. and **Zavalnij** B. “Edge coloring of graphs, uses, limitation, complexity.” *Acta Univ. Sapientiae, Informatica.* 8, 1 (2016) 63—81.

8. Szabo S. and **Zavalnij** B. “Reducing Graph Coloring to Clique Search.” *Asia Pacific Journal of Mathematics.* 3 (2016), 64–85.

9. **Zavalnij** B. “Three Versions of Clique Search Parallelization.” *Journal of Computer Science and Information Technology.* Vol. 2: (No. 2) pp. 9–20. (2014)

10. Szabo S. and **Zavalnij** B. “Coloring the edges of a directed graph.” *Indian Journal of Pure and Applied Mathematics.* April 2014, Volume 45, Issue 2, pp 239–260.

11. Szabo S. and **Zavalnij** B. “Coloring the nodes of a directed graph.” *Acta Univ. Sapientiae, Informatica.* 6, 1 (2014) 117—131.

12. Szabo S. and **Zavalnij** B. “Greedy algorithms for triangle free coloring.” *AKCE International Journal of Graphs and Combinatorics.* 9:(2) pp. 169–186. (2012)

Proceedings

13. Szabo S. and **Zavalnij** B. *Combining algorithms for vertex cover and clique search.* In: Middle-European Conference on Applied Theoretical Computer Science (MATCOS 2019). Proceedings of the 22nd International Multiconference INFORMATION SOCIETY – IS 2019 pp. 71–74.

14. Szabo S. and **Zavalnij** B. *Splitting partitions and clique search algorithms.* In: Middle-European Conference on Applied Theoretical Computer Science (MATCOS 2019). Proceedings of the 22nd International Multiconference INFORMATION SOCIETY – IS 2019 pp. 75–78.

15. Szabo S. and **Zavalnij** B. *A different approach to maximum clique search.* In: IEEE Computer Society, 20th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing. (SYNASC2018) IEEE Proceedings, (2018) pp. 310–316.

16. Depolli, M., Konc, J., Szabo S. and **Zavalnij** B. *Usage of hereditary colorings of product graphs in clique search programs.* In: Middle-European Conference on Applied Theoretical Computer Science (MATCOS 2016). Proceedings of the 19th International Multiconference INFORMATION SOCIETY - IS 2016. pp. 40–43

17. **Zavalnij** B. *Speeding up Parallel Combinatorial Optimization Algorithms with Las Vegas Method.* In: Lecture Notes in Computer Science, 10th International Conference, Vol. 9374. Springer, 2015. pp. 258–266.

18. Bóta A., Krész M. and **Zavalnij** B. *Adaptations of the k-means algorithm to community detection in parallel environments.* In: IEEE Computer Society, 17th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing. (SYNASC2015) IEEE Proceedings, (2015) pp. 299–302.

19. **Zavalnij** B. *The Las Vegas method of parallelization.* In: Proceedings of Information Society 2014 – IS 2014: Volume A; Intelligent Systems. pp. 105–108. (Best paper award.)

Conference talks

20. Szabo Sandor, **Zavalnij** Bogdan. *Using sequential kclique solver for the Vertex Cover problem.* 14th International Symposium on Parameterized and Exact Computation (IPEC 2019 / ALGO 2019). Munich, Germany. 2019.09.11–13. Poster.

21. **Zavalnij** Bogdan *On the auxiliary algorithm of coloring.* SWORDS 2014: Szeged WORkshop on Discrete Structures. Szeged, Hungary, 2014.10.09–10.

22. **Zavalnij** Bogdan. *Discrete optimization in supercomputing environment.* 9th International PhD and DLA Symposium. October 21–22, 2013, Pécs.

23. Szabo Sandor, **Zavalnij** Bogdan. *Selected topics in combinatorial optimization.* Sok-processzoros rendszerek a mérnöki gyakorlatban: TÁMOP–4.1.2/A/1–11/1–2011–0063. October 16–18, 2013, Harkány.

24. **Zaválnij** Bogdán. *Different Coloring Methods in Maximum Clique Search.* 5th Veszprém Optimization Conference: Advanced Algorithms. Veszprém, Hungary, December 11-14, 2012.

25. **Zaválnij** Bogdán. *Let's Gamble! Casinoes and Parallelization.* Mathematics in Educating Civil-Engineering and Architecture Conference. Pécs, 2011.

Bibliography

[Abe1999] Abello, J., Pardalos, P.M. and Resende, M.G.C. *On Maximum Clique Problems In Very Large Graphs.* In: External Memory Algorithms. AMS. 1999. pp. 119–130.

[Ada1988] Adams, J., Balas, E., Zawack, D. “The shifting bottleneck procedure for job shop scheduling.” *Management Science.* **34.3**, 391–401, 1988.

[Ahm2012] Ahmed, S. “Applications of Graph Coloring in Modern Computer Science.” *International Journal of Computer and Information Technology* VOLUME 03, ISSUE 02. 2012.

[Aki2016] Akiba, T. and Iwata, Y. “Branch-and-reduce exponential/fpt algorithms in practice: A case study of vertex cover.” *Theoretical Computer Science.* 609:211–225, 2016.

[Ale2003] Alexe, G., Hammer, P. L., Lozin, V. V. and de Werra, D. “Struction revisited.” *Discrete applied mathematics.* 132(1-3):27–46, 2003.

[Bal1986] Balas, E. and Yu, Ch. S. “Finding a maximum clique in an arbitrary graph,” *SIAM J. Comput.* Vol 15, No 4, November 1986.

[Bab1979] Babai, L. *Monte-Carlo algorithms in graph isomorphism testing.* Université de Montréal, D.M.S. No.79–10. (1979)

[Bat2013] Batsyn, M., Goldengorin, B., Maslov, E. and Pardalos, P.M. “Improvements to MCS algorithm for the maximum clique problem.” *Journal of Combinatorial Optimization.* February 2014, Volume 27, Issue 2. pp. 397–416.

[Berg1973] Berge, C. *Graphs and Hypergraphs.* North-Holland, 1973.

[Bogd2001] Bogdanova, G. T. and Östergård, P. R. J. "Bounds on codes over an alphabet of five elements." *Discrete Mathematics.* 240, 1-3, pp. 13–19. 2001.

[Bogi2003] Boginski, V., Butenko, S. and Pardalos, P.M. *On Structural Properties of the Market Graph.* In: Innovations in Financial and Economic Networks. Ed. Nagurney, A. Edward Elgar Publishing Inc. 2003.

[Bogi2006] Boginski, V., Butenko, S. and Pardalos, P.M. "Mining Market Data: A Network Approach." *Computers and Operations Research.* Volume 33 Issue 11, November 2006. pp. 3171–3184

[Bogi2014] Boginski, V., Butenko, S., Shirokikh, O., Trukhanov, S. and Lafuente, J.G. "A network-based data mining approach to portfolio selection via weighted clique relaxations." *Annals of Operations Research.* May 2014, Volume 216, Issue 1. pp. 23–34.

[Bom1997] Bomze, I., Pelillo, M. and Giacomini, R. *Evolutionary Approach to the Maximum Clique Problem: Empirical Evidence on a Larger Scale.* In: Developments in Global Optimization. Ed. Bomze, M., Csendes T., Horst, R., Pardalos, P.M. Springer 1997. pp. 95–108.

[Bom1999] Bomze, I., Budinich, M., Pardalos, P.M. and Pelillo, M. *The Maximum Clique Problem.* In: Handbook of Combinatorial Optimization. Supplement Volume A. Eds. Du, D-Z.,Pardalos, P.M. Kluwer Academic Publishers 1999. pp. 1–74.

[Bor1999] Borchers, B. "CSDP, A C library for semidefinite programming." *Optimization methods and Software.* 11:1–4. pp. 613–623. 1999.

[Bor2007] Borchers, B. and Young, J. G. "Implementation of a primal–dual method for SDP on a shared memory parallel architecture." *Computational Optimization and Applications.* 37:3. pp. 355–369. 2007.

[Bota2014] Bóta A. *Methods for the description and analysis of processes in real-life networks.* PhD Dissertation. 2014.

[Bota2015] Bóta A., Krész M. and Zavalnij B. *Adaptations of the k-means algorithm to community detection in parallel environments.* In: IEEE Computer Society, 17th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing. (SYNASC2015) IEEE Proceedings, (2015) pp. 299–302.

[Braw1989] Brawer, S. *Introduction to Parallel Programming.* Academic Press. 1989.

[Brel1979] Brélaz, D. "New Methods to Color the Vertices of a Graph." *Communications of the ACM.* 1979. Volume 22. Number 4. pp. 251–257.

[Bret2013] Bretto, A. *Hypergaph Theory: An Introduction.* Springer 2013.

[Bron1973] Bron, C. and Kerbosch, J, "Algorithm 457: finding all cliques of an undirected graph." *Commun. ACM.* 16 (9): 575–577. 1973.

[Bull2009] Bullmore, E. and Sporns, O. "Complex brain networks: graph theoretical analysis of structural and functional systems." *Nature Reviews Neuroscience.* 10. pp. 186–198. 2009.

[But2002] Butenko, S., Pardalos, P., Sergienko, I., Shylo, V. and Stetsyuk, P. *Finding Maximum Independent Sets in Graphs Arising from Coding Theory.* In: Proceedings of the 2002 ACM Symposium on Applied Computing. Madrid 2002. pp. 542–546.

[But2009] Butenko, S., Pardalos, P., Sergienko, I., Shylo, V. and Stetsyuk, P. *Estimating the size of correcting codes using extremal graph problems.* In: Optimization. Structure and Applications. Ed. Pearce, Ch. and Hunt, E. Springer. 2009. pp. 227–243.

[Can2015] Caniou, Y., Codognet, P., Richoux, F. et al. "Large-scale parallelism for constraint-based local search: the costas array case study." *Constraints.* 20, 30–56. 2015.

[Carm2012] Carmo, R. and Züge, A. "Branch and bound algorithms for the maximum clique problem under a unified framework." *Journal of the Brazilian Computer Society* . June 2012, Volume 18, Issue 2. pp. 137–151.

[Carr1990] Carraghan, R. and Pardalos, P.M. “An exact algorithm for the maximum clique problem.” *Operation Research Letters.* **9** (1990), 375–382.

[Chan2014] Chan, J., Lam, S. and Hayes, C. “Generalised blockmodelling of social and relational networks using evolutionary computing.” *Soc. Netw. Anal. Min.* (2014) 4:155

[Chap2019] Chapuis, G., Djidjev, H., Hahn, G. et al. “Finding Maximum Cliques on the D-Wave Quantum Annealer.” *J Sign Process Syst.* 91, 363–377 2019.

[Chen2012] Cheng, J., Zhu, L., Ke, Y. and Chu, S. *Fast Algorithms for Maximal Clique Enumeration with Limited Memory.* In: Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. Beijing, 2012. pp. 1240–1248.

[Chi2014] Chiang, S. and Haneef, Z. “Graph theory findings in the pathophysiology of temporal lobe epilepsy.” *Clinical Neurophysiology.* 2014 Jul. 125(7). 1295–1305.

[Clea1991] Clearwater, S.H., Huberman, B.A. and Hogg, T. “Cooperative Solution of Constraint Satisfaction Problems.” *Science.* 1991 Nov 22;254(5035):1181–1183.

[Clea1992] Clearwater, S.H., Hogg, T. and Huberman, B.A. *Cooperative Problem Solving.* In: Computation: The Micro and the Macro View. Ed. Huberman B.A. World Scientific 1992. pp. 33–77.

[Coop2006] Cooper, K.D. and Dasgupta, A. *Tailoring Graph-coloring Register Allocation For Runtime Compilation.* In: Proceedings of the 2006 International Symposium on Code Generation and Optimization (CGO’06), New York. New York, 2006.

[Corn2006] Cornujols, G., Karamanov, M. and Li, Y. “Early Estimates of the Size of Branch-and-Bound Trees.” *INFORMS Journal on Computing.* Volume 18 Issue 1, Winter 2006. pp. 86–96.

[Corn2008] Cornaz, D. and Jost, V. “A one-to-one correspondence between colorings and stable sets.” *Operations Research Letters.* 36 (2008) pp. 673–676.

[Corn2016] Cornaz, D., Furini, F., Malaguti, E. “Solving vertex coloring problems as maximum weight stable set problems.” *Discrete Applied Mathematics.* Volume 217, Part 2, (2017), pp. 151–162.

[Cost1965] Costas, J.P. *Medium constraints on sonar design and performance.* Class 1 Report R65EMH33, G.E. Corporation. 1965.

[Cost1984] Costas, J. P. “A Study of Detection Waveforms Having Nearly Ideal Range-Doppler Ambiguity Properties.” In: Proc. IEEE 72, 996-1009, 1984.

[Culb1992] Culberson, J.C. *Iterated Greedy Graph Coloring and the Difficulty Landscape.* Technical Report. University of Alberta. 1992.

[Cyg2015] Cygan, M., Fomin, F.V., Kowalik, Ł., Lokshtanov, D., Marx D., Pilipczuk, M., Pilipczuk, M., Pilipczuk, M. and Saurabh, S. *Parameterized algorithms.* Springer. 2015.

[Czaj2009] Czajkowski, G. *Large-scale graph computing at Google.* 2009. `http://googleresearch.blogspot.hu/2009/06/large-scale-graph-computing-at-google.html`

[Dang2016] Dang, H.-V., Snir, M. and Gropp, W. *Towards millions of communicating threads.* EuroMPI 2016.

[Deb2011] Debroni, J., Eblen, J.D., Langston, M.A., Myrvold, W., Shor, P. and Weerapurage, D. *A complete resolution of the Keller maximum clique problem.* In: Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms. 2011. pp. 129–135.

[Diaz2012a] Diaz. D., Richoux, F., Codognet, P., Caniou, Y. and Abreu, S. “Parallel Local Search for the Costas Array Problem,” 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum, Shanghai, 2012, pp. 1793–1802.

[Diaz2012b] Diaz. D., Richoux, F., Codognet, P., Caniou, Y. and Abreu, S. “Constraint-Based Local Search for the Costas Array Problem.” In: Hamadi Y., Schoenauer M. (eds) Learning and Intelligent Optimization. LION 2012. Lecture Notes in Computer Science, vol 7219. Springer, 2012.

[Dzu2019] Dzulfikar, M.A., Fichte, J.K. and Hecher, M. "The PACE 2019 Parameterized Algorithms and Computational Experiments Challenge: The Fourth Iteration." In: 14th International Symposium on Parameterized and Exact Computation (IPEC 2019). Leibniz International Proceedings in Informatics (LIPIcs). 25:1–25:23. 2019.

[Ebe1984] Ebenegger, Ch. Hammer, P.L. and de Werra, D. "Pseudo-boolean functions and stability of graphs." *North-Holland mathematics studies.* volume 95, pp. 83–97. 1984.

[Ebl2012] Eblen, J. D., Phillips, C. A., Rogers G. L. and Langston, M. A. "The maximum clique enumeration problem: algorithms, applications, and implementations." *BMC Bioinformatics.* 2012;13

[Erd1959] Erdős P. "Graph theory and probability." *Canad. J. Math.* **11** (1959), 34–38.

[Foul1992] Foulds, L.R. *Graph Theory Applications.* Springer. 1992.

[Fu2006] Fu, Z. and and Malik, S. *On solving the partial MAX-SAT problem.* In: International Conference on Theory and Applications of Satisfiability Testing – SAT2006. Lecture Notes in Computer Science. pp. 252–265. 2006.

[Gare2003] Garey M.R. and Johnson, D.S. *Computers and Intractability: A Guide to the Theory of NP-completeness*, Freeman, New York, 2003.

[God2001] Godsil, Ch. and Royle, G. *Algebraic Graph Theory.* Springer, 2001.

[Gom1997] Gomes, C.P. and Selman, B. "Algorithm Portfolio Design: Theory vs. Practice." In: Proceedings of the Thirteenth Conference on Uncertainty in Artificial Intelligence. UAI'97. 1997.

[Gram2003] Grama, A., Gupta, A., Karypis, G. and Kumar, V. *Introduction to Parallel Computing.* 2nd ed. Addison Wesley. 2003.

[Greg2012] Gregori, E., Lenzini, L. and Mainardi, S. "Parallel k-Clique Community Detection on Large-Scale Networks." *IEEE Transactions on Parallel and Distributed Systems.* Volume: 24, Issue: 8, Aug. 2013. pp. 1651–1660.

[Har2015] HARANGOZÓ P. and SZABÓ I. *Egy nehéz gráfelméleti probléma megoldása szuperszámítógép segítségével.* (In English: *Solving a hard graph problem using supercomputers.*) National Scientific Students' Associations Conference 2015.

[Hass1993] HASSELBERG, J., PARDALOS, P.M. and VAIRAKTARAKIS, G. "Test case generators and computational results for the maximum clique problem." *Journal of Global Optimization* **3** (1993), 463–482. `http://www.springerlink.com/content/p2m65n57u657605n`

[Hay2006] HAYES, B. "Connecting the Dots." *American Scientist.* Volume 94. pp. 400–404.

[Hes2020] HESPE, D., LAMM, S., SCHULZ, Ch. and STRASH, D. "WeGotYouCovered: The Winning Solver from the PACE 2019 Implementation Challenge, Vertex Cover Track." SIAM Workshop on Combinatorial Scientific Computing 2020, February 11–13, 2020, Seattle.

[Hrg2019] HRGA, T. and POVH, *Accelerated Direction Augmented Lagrangian Method for Semidefinite Programs.* In: Proceedings of the 15th International Symposium on Operations Research. SOR'19. Slovenia. pp 155-160. 2019.

[Iva2013] IVÁNYI P. and RADÓ J. *Előfeldolgozás párhuzamos számításokhoz* Tankönyvtar, 2013.

[Jain1999] JAIN, A.S. and MEERAN, S. "Deterministic job-shop scheduling: Past, present and future." *European Journal of Operational Research.* **113**, 390–434. 1999.

[Karg1998] KARGER, D., MOTWANI, R. and SUDAN, M. "Approximate graph coloring by semidefinite programming," *J. ACM*, 45, 2 (March 1998), pp. 246–265.

[Karn2003] KARNIADAKIS, G.E. and Kirby, R.M. *Parallel Scientific Computing in C++ and MPI.* Cambridge UP. 2003.

[Karp1972] KARP, R.M. *Reducibility Among Combinatorial Problems.* In: Complexity of Computer Computations. Eds: R. E. Miller and J. W. Thatcher. New York. Plenum. pp. 85–103.

[Kilb2006] Kilby, P., Slaney J., Thiébaux, S. and Walsh, T. *Estimating Search Tree Size.* In: Proceedings of the 21st National Conference on Artificial Intelligence - Volume 2. 2006. pp. 1014–1019.

[Kivi2016] Kiviluoto, L., Östergård, P. R. J. and Vaskelainen, V. P. "Algorithms for finding maximum transitive subtournaments." *Journal of Combinatorial Optimization.* 31, 2, pp. 802–814. 2016.

[Knu1974] Knuth, D. E. *Estimating the Efficiency of Backtrack Programs.* Technical Report. Stanford University. 1974.

[Knu1994] Knuth, D. E. "The Sandwich Theorem ." *Electronic Journal of Combinatorics.* 1. 1994. #A1

[Kom2015] Komosko, L., Batsyn, M., San Segundo, P. and Pardalos, P.M. "A fast greedy sequential heuristic for the vertex colouring problem based on bitwise operations." *Journal of Combinatorial Optimization.* March 2015.

[Konc2007] Konc, J. and Janezic, D. "An improved branch and bound algorithm for the maximum clique problem." *MATCH Commun. Math. Comput. Chem.* 2007, 58, 569–590.

[Konc2010] Konc, J. and Janezic, D. "ProBiS algorithm for detection of structurally similar protein binding sites by local structural alignment." *Bioinformatics.* 26(9), 03. pp. 1160–1168. 2010.

[Konc2012] Konc, J., Depolli, M., Trobec, R., Rozman, K. and Janezic, D. "Parallel-ProBiS: Fast parallel algorithm for local structural comparison of protein structures and binding sites." *Journal of Computational Chemistry.* 33(27):2199–2203, 2012.

[Krai1953] Kraitchik, M. *Mathematical Recreations.* 2nd ed. Dover Publications. New York. 1953.

[Kre2002] Krebs, V. "Mapping Networks of Terrorist Cells." *Connections.* 24(3), 2002. pp. 43–52.

[Kuml2005] Kumlander, D. *Some Practical Algorithms to Solve The Maximum Clique Problem.* PhD Thesis. Tallin, 2005.

[Kuml2020] Kumlander, D. and Porošin, A. *Reversed Search Maximum Clique Algorithm Based on Recoloring.* In: Le Thi H., Le H., Pham Dinh T. (eds) Optimization of Complex Systems: Theory, Models, Algorithms and Applications. WCGO 2019. Advances in Intelligent Systems and Computing, vol 991. Springer. 2020.

[Kump2008] Kumpula, J.M., Kivelä, M., Kaski, K. and Saramäki, J. "A sequential algorithm for fast clique percolation." *Phys. Rev.* E 79, 026109 (2008), arXiv:0805.1449

[Lam2016] Lamm, S. and Sanders, P. and Schulz, C. and Strash, D., Werneck, R.F. "Finding Near-Optimal Independent Sets at Scale." In: Proceedings of the 16th Meeting on Algorithm Engineering and Exerpimentation (ALENEX'16)

[Leš2020] Lešnik, S. and Konc, J. *In Silico Laboratory: Tools for Similarity-Based Drug Discovery.* In: Labrou N. (eds) Targeting Enzymes for Pharmaceutical Development. Methods in Molecular Biology, vol 2089. Humana, 2020.

[Li2010a] Li, C.-M. and Quan, Z. *An Efficient Branch-and-Bound Algorithm Based on MaxSAT for the Maximum Clique Problem.* In: Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence. (AAAI-10), pp. 128–133.

[Li2010b] Li, C.-M. and Quan, Z. *Combining graph structure exploitation and propositional reasoning for the maximum clique problem.* In: 22nd IEEE Internat ional Conference on Tools with Artificial Intelligence (ICTAI). IEEE. pp. 344–351. 2010.

[Li2013] Li, C.-M., Fang, Z. and Xu, K. *Combining MaxSAT Reasoning and Incremental Upper Bound for the Maximum Clique Problem.* In: Proceedings of the 2013 IEEE 25th International Conference on Tools with Artificial Intelligence. (ICTAI2013), pp. 939–946.

[Li2017] Li, C.-M., Jiang, H. and Manya, F. "On Minimization of the Number of Branches in Branch-and-Bound Algorithms for the Maximum Clique Problem." *Computers & Operations Research.* 84: 1–15. 2017.

[Lov1976] Lovasz L. "On the Shannon Capacity of a Graph," *IEEE Transactions on Information Theory.* Volume 25 Issue 1, January 1979 pp. 1–7.

[Madd2007] Madduri, K., Bader, D.A., Berry, J.W., Crobak, J.R. and Hendrickson, B.A. *Multithreaded Algorithms for Processing Massive Graphs* In: Petascale Computing: Algorithms and Applications. Ed.: Bader, D.A. Chapman & Hall/CRC Computational Science. 2007.

[Marg2019] Margenov, S., Rauber, T., Atanassov, E., Almeida, F., Blanco, V., Ciegis, R., Cabrera, A. Frasheri, N., Harizanov, S., Kriauzien, R., Rünger, G., San Segundo, P., Starikovicius, A., Szabo S. and Zavalnij B. *Applications for ultrascale systems.* In: Ultrascale Computing Systems. Eds.: Carretero, J., Jeannot, E., Zomaya, A.Y. Institution of Engineering and Technology (IET), London, (2019) pp. 189–244.

[Marx2004] Marx D. "Graph colouring problems and their applications in scheduling." *Periodica Polytechnica, Electrical Engineering.* 48 (1—2), pp. 11—16. 2004.

[Mar1998] Marzetta, A. *ZRAM: A Library of Parallel Search Algorithms and Its Use in Enumeration and Combinatorial Optimization.* PhD dissertation, Swiss Federal Institute of Technology Zurich. 1998.

[Math2017] Mathew, K. A. and Östergård, P. R. J. "New lower bounds for the Shannon capacity of odd cycles." *Design Codes and Cryptography.* 84, 1-2, pp. 13–22. 2017..

[Matt2005] Mattson, T.G., Sanders, B.A. and Massingill, B.L. *Patterns for Parallel Programming.* Addison-Wesly. 2005.

[McCo2012] McCool, M., Robison, A.D. and Reinders, J. *Structured Parallel Programming. Patterns for Efficient Computation.* Elsevier. 2012.

[McCr2015] McCreesh, C. and Prosser, P. "The Shape of the Search Tree for the Maximum Clique Problem, and the Implications for Parallel Branch and Bound." *ACM Transactions on Parallel Computing.* Volume 2 Issue 1, May 2015. Article No. 8.

[Milo2002] MILO, R., SHEN-ORR, S., ITZKOVITZ, S., KASHTAN, N., CHKLOVSKII, D. and ALON, U. “Network motifs: simple building blocks of complex networks.” *Science.* 298(5594):824–827 (2002)

[Moor2004] van MOORSEL, A. and WOLTER, K. “Optimal restart times for moments of completion time.” IEE Proceedings – Software(2004),151(5). pp. 219–223.

[MPI2012] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard. Version 3.0.* 2012.

[Nagh2019] NAGHSH, Z., JAVAD-KALBASI, M. and VALAEE, S. “Digitally Annealed Solution for the Maximum Clique Problem with Critical Application in Cellular V2X.” ICC 2019 - 2019 IEEE International Conference on Communications (ICC), Shanghai, China, 2019, pp. 1-7.

[Nik2015] NIKOLAEV, A., BATSYN, M. and SAN SEGUNDO, P. “Reusing the Same Coloring in the Child Nodes of the Search Tree for the Maximum Clique Problem.” *Learning and Intelligent Optimization.* LNCS Volume 8994, 2015, pp. 275–280.

[Öst2002] ÖSTERGÅRD, P.R.J. “A fast algorithm for the maximum clique problem.” *Discrete Applied Mathematics.* 120 (1–3): 197–207. 2002.

[Öst2019] ÖSTERGÅRD, P. R. J. and PÖLLÄNEN, A. “New Results on Tripod Packings.” *Discrete and Computational Geometry.* 61, 2, pp. 271–284. 2019.

[Öst2020] ÖSTERGÅRD, P. R. J. and SZÖLLŐSI, F. “Constructions of maximum few-distance sets in euclidean spaces. ” *Electronic Journal of Combinatorics.* 27, 1, P1.23. 2020.

[Ouy1998] OUYANG, M.: “How Good Are Branching Rules in DPLL?” *Discrete Applied Mathematics.* 89 (1–3): 281–286. 1998.

[Papa1994] PAPADIMITRIOU, C.H. *Computational Complexity*, Addison-Wesley Publishing Company, Inc., Reading, MA 1994.

[Patt2013a] PATTILLO, J., YOUSSEF, N. and BUTENKO, S. “On clique relaxation models in network analysis.” *European Journal of Operational Research.* Volume 226, Issue 1, 1 April 2013. pp. 9–18.

[Patt2013b] Pattillo, J., Veremyev, A., Butenko, S. and Boginski, V. "On the maximum quasi-clique problem." *Discrete Applied Mathematics* . Volume 161, Issues 1–2, January 2013. pp. 244–257.

[Pet2004] Petersen, W.P. and Arbenz, P. *Introduction to Parallel Computing. A practical Guide with Examples in C.* Oxford UP. 2004.

[Pol2019] Polak, S. C. and Schrijver, A. "New lower bound on the Shannon capacity of C7 from circular graphs." *Information Processing Letters.* Volume 143, March 2019, pp. 37–40.

[Purd1977] Purdom, P.W. *Tree Size by Partial Backtracking.* Technical Report. Indiana University. 1977.

[Pro2015] McCreesh, C. and Prosser, P. "The Shape of the Search Tree for the Maximum Clique Problem, and the Implications for Parallel Branch and Bound." *ACM Transactions on Parallel Computing.* Volume 2 Issue 1, May 2015 Article No. 8.

[Reg2013] Régin, J. C., Rezgui, M. and Malapert, A. *Embarrassingly Parallel Search.* In: Schulte C. (eds) Principles and Practice of Constraint Programming. CP 2013. Lecture Notes in Computer Science, vol 8124.

[Rob2013] Robinson, I., Webber, J. and Eifrem, E. *Graph Databases.* O'Reilly Media. 2013.

[Ross2014] Rossi, R.A. and Ahmed, N.K. "Coloring large complex networks." *Soc. Netw. Anal. Min.* 4, 228. 2014.

[Rub2009] Rubinov, M. and Sporns, O. "Complex network measures of brain connectivity: uses and interpretations." *NeuroImage.* 2010 Sep. 52(3). pp. 1059–1069.

[Seg2010] San Segundo, P., Rodríguez-Losada, D., Matía, F. et al. "Fast exact feature based data correspondence search with an efficient bit-parallel MCP solver." *Appl Intell.* 32, 311–329 2010.

[Seg2011] San Segundo, P., Rodriguez-Losada, D. and Jimenez, A. "An exact bit-parallel algorithm for the maximum clique problem." *Computers & Operations Research.* 38(2), 2011, 571–581.

[Seg2012] San Segundo, P. "A new DSATUR-based algorithm for exact vertex coloring." *Computers & Operations Research.* 30:7, 2012, 1724–1733.

[Seg2013] San Segundo, P., Matia, F.; Rodriguez-Losada, D. and Hernando, M. "An improved bit parallel exact maximum clique algorithm." *Optimization Letters.* 7(3), 2013, 467–479.

[Seg2014] San Segundo, P. and Tapia, C. "Relaxed approximate coloring in exact maximum clique search." *Computers & Operations Research.* 44, 2014, 185–192.

[Seg2015] San Segundo, P., Nikolaev, A. and Batsyn, M. "Infra-chromatic bound for exact maximum clique search." *Computers & Operations Research.* Volume 64, December 2015, pp. 293—303.

[Schl2016] Schlauch, W.E. and Zweig, K.A. "Motif detection speed up by using equations based on the degree sequence." *Social Network Analysis and Mining.* (2016) 6: 47. `doi:10.1007/s13278-016-0357-6`

[Sew1996] Sewell, E. *An improved algorithm for exact graph coloring.* In. M.A. Trick, D.S. Johnson (Eds.), Cliques, coloring, and satisfiability. Proceedings of the second DIMACS implementation challenge, vol. 26. American Mathematical Society, 1996, 359–373.

[Sima1997] Sima, D., Fountain, T. and Kacsuk, P. *Advanced Computer Architectures: A Design Space Approach.* Addison Wesley. 1997.

[Sloan] Sloane, N.J.A. *Challenge Problems: Independent Sets in Graphs.* `http://neilsloane.com/doc/graphs.html`

[Szab2011] Szabo, S. "Parallel algorithms for finding cliques in a graph." *Journal of Physics: Conference Series Volume 268, Number 1.* 2011 J. Phys.: Conf. Ser. 268 012030 doi:10.1088/1742-6596/268/1/012030

[Szab2012] Szabó S. and Zaválnij B. "Greedy Algorithms for Triangle Free Coloring." *AKCE Int. J. Graphs Comb.,* 9, No. 2 (2012), pp. 169–186.

[Szab2013] SZABÓ, S. "Monotonic matrices and clique search in graphs." *Annales Univ. Sci. Budapest., Sect. Comp.* **41** (2013), 307–322.

[Szab2014a] SZABÓ S. and ZAVÁLNIJ B. "Coloring the edges of a directed graph." *Indian Journal of Pure and Applied Mathematics.* April 2014, Volume 45, Issue 2, pp 239–260.

[Szab2014b] SZABÓ S. and ZAVÁLNIJ B. "Coloring the nodes of a directed graph." *Acta Univ. Sapientiae, Informatica.* 6, 1 (2014) 117—131.

[Szab2014c] SZABÓ S. and ZAVÁLNIJ B. "Estimating clique size via coloring edges in graphs." *AKCE Int. J. Graphs Comb.* (submitted.)

[Szab2016b] SZABÓ S. and ZAVÁLNIJ B. "Reducing Graph Coloring to Clique Search," *Asia Pacific Journal of Mathematics*, 3 (2016), pp. 64–85.

[Szab2017] SAN SEGUNDO, P., SZABÓ S. and **ZAVÁLNIJ** B. "Parallelization of the clique search problem using sub-chromatic bounds." NESUS. Technical Report. January 17, 2017.

[Szab2018a] SZABO S. and **ZAVALNIJ** B. *A different approach to maximum clique search.* In: IEEE Computer Society, 20th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing.
(SYNASC2018) IEEE Proceedings, (2018) pp. 310–316.

[Szab2018b] SZABO S. and ZAVALNIJ B. "Decomposing clique search problems into smaller instances based on node and edge colorings." *Discrete Applied Mathematics.* 242 (2018) pp. 118–129.

[Szab2019a] SZABO S. and ZAVALNIJ B. "Benchmark Problems for Exhaustive Exact Maximum Clique Search Algorithms." *Informatica (Ljubljana).* 43 : 2 (2019) pp. 177–186.

[Szab2019b] SZABO S. and ZAVALNIJ B. "Reducing hypergraph coloring to clique search." *Discrete Applied Mathematics.* 264. (2019) pp. 196–207.

[Tor2017] TORRES-JIMENEZ, J., PEREZ-TORRES, J.C. and MALDONADO-MARTINEZ, G. "hClique: An exact algorithm for maximum clique problem in uniform hypergraphs." *Discrete Mathematics, Algorithms and Applications.* December 2017, Vol. 09, No. 06

[Tom2003] TOMITA, E. and SEKI, T. “An Efficient Branch-and-Bound Algorithm for Finding a Maximum Clique.” *Discrete Mathematics and Theoretical Computer Science, Lecture Notes in Computer Science 2731.* Springer-Verlag, pp. 278–289. 2003.

[Tru2012] TRUCHET, C., RICHOUX, F. and CODOGNET, P. “Prediction of Parallel Speed-ups for Las Vegas Algorithms.” `http://arxiv.org/abs/1212.4287`

[Val1990] VALIANT, L.G. “A bridging model for parallel computation.” *Communications of the ACM.* Volume 33 Issue 8, Aug. 1990.

[Var2013] VÁRADY G. and ZAVÁLNIJ B. *Introduction to MPI by examples.* TypoTEX Kiadó, 2013.

[Volo2002] VOLOSHIN, V.I. *Coloring Mixed Hypegraphs: Theory, Algorithms and Applications.* AMS. 2002.

[Weisst] WEISSTEIN, E.W. “Monotonic Matrix.” In: MathWorld–A Wolfram Web Resource. `http://mathworld.wolfram.com/MonotonicMatrix.html`

[Wood1997] WOOD, D.R. “An algorithm for finding a maximum clique in a graph.” *Operations Research Letters.* Volume 21, Issue 5, 12 January 1997. pp. 211–217.

[Wu2015] WU, Q. and HAO, J.-K. “A review on algorithms for maximum clique problems.” *European Journal of Operational Research.* 242:3. 693–709. 2015.

[Zav2014a] ZAVÁLNIJ B. “Three Versions of Clique Search Parallelization.” *Journal of Computer Science and Information Technology.* Vol. 2:(No. 2) pp. 9–20. (2014)

[Zav2014b] ZAVALNIJ, B. “The Las Vegas method of parallelization.” In: Rok Piltaver, Matjaž Gams (eds.) Information Society 2014 – IS 2014: Volume A; Intelligent Systems. Ljubljana, Slovenia, 2014.10.07–2014.10.08. Ljubljana: pp. 105–108.

[Zav2015] ZAVALNIJ, B. “Speeding up Parallel Combinatorial Optimization Algorithms with Las Vegas Method.” *10th International Conference on Large-Scale Scientific Computations.* June 8–12, 2015, Sozopol, Bulgaria. (accepted for publication in the Lecture Notes in Computer Science (LNCS) – Springer)