Jagiellonian University
Department of Theoretical Computer Science

Adrian Siwiec

Perfect Graph Recognition and Coloring

Master Thesis

Supervisor: dr inż. Krzysztof Turowski

September 2020

Abstract

Perfect graphs are a subject of intense study since their introduction in 1961. Only in 2001 the famous Perfect graph conjecture was proven to be true and in 2005 a polynomial method of determining if a graph is perfect was found. Since 1988 an algorithm is known for coloring perfect graphs, but it uses an ellipsoid method which is said to be complicated and impractical. As recently as in 2018 a polynomial algorithm that uses a combinatorial approach for coloring perfect graphs without squares was published.

We begin with basic definitions in Chapter 1. In Chapter 2 we introduce perfect graphs and Berge graphs, give an overview of the strong perfect graph theorem and talk about an algorithm for polynomial perfect graph recognition. Chapter 3 is a study of the ellipsoid method of coloring perfect graphs. In Chapter 4 we present our implementation of algorithms from Chapters 2 and 3, along with notes on optimization and parallelisation. Appendix A is a overview of the recent algorithm for coloring square-free perfect graphs.

List of definitions

List of algorithms

Contents

Chapter 1

Basic Graph Definitions

We begin by recalling the basic notions of graph theory. We use standard definitions, sourced from the book by Bollobás *Modern graph theory*, modified and extended as needed.

Definition 1.1 (graph)**.** *A* graph G *is an ordered pair of disjoint sets* (V, E) *such that* E *is the subset of the set* $\binom{V}{2}$*, that is, of unordered pairs of* V*.*

We will only consider finite graphs, that is, V and E are always finite. If G is a graph, then $V = V(G)$ is the *vertex set* of G, and $E = E(G)$ is the *edge set*. When the context of G is clear we will use V and E to denote its vertex and edge set.

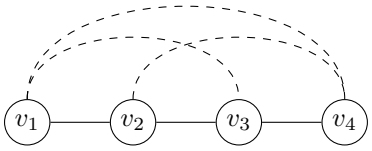


Figure 1.1: An example graph G_0

An edge $\{x, y\}$ is said to *join*, or *be between* vertices x and y and is denoted by xy. Thus xy and yx denote the same edge (all our graphs are *undirected*). If $xy \in E(G)$ then x and y are *adjacent*, *connected* or *neighboring*. By $N(x)$ we will denote the *neighborhood* of x, that is, all vertices y such that xy is an edge. Similarly, for $X \subseteq V(G)$, by $N(X)$ we will denote the *neighborhood* of X, meaning all vertices of $v \in V(G) \setminus X$, so that there is a $x \in X$, that xv is an edge in G. If $xy \notin E(G)$ then xy is a *nonedge* and x and y are *nonneighbors* or *anticonnected*.

Figure 1.1 shows an example graph $G_0 = (V, E)$ with $V = \{v_1, v_2, v_3, v_4\}$ and $E = \{v_1v_2, v_2v_3, v_3v_4\}$. We will mark edges as solid lines on figures. Nonedges significant to the ongoing reasoning will be marked as dashed lines.

Definition 1.2 (subgraph)**.** *A graph* $G' = (V', E')$ *is a* subgraph *of* $G = (V, E)$ *if and only if* $V' \subseteq V$ *and* $E' \subseteq E$*.*

Definition 1.3 (induced subgraph)**.** *If* $G' = (V', E')$ *is a subgraph of* G *and it contains* all edges *of* G *that join two vertices in* V'*, then* G' *is said to be an* induced subgraph *of* G *and is denoted* $G[V']$*.*

Given a graph $G = (V, E)$ and a set $X \subseteq V$ by $G \setminus X$ we will denote a induced subgraph $G[V \setminus X]$.

For example $(\{v_1, v_2, v_3\}, \{v_1v_2\})$ is *not* an induced subgraph of the example graph G_0, while $(\{v_1, v_2, v_3\}, \{v_1v_2, v_2v_3\}) = G_0[\{v_1, v_2, v_3\}] = G_0 \setminus \{v_0\}$ is.

Definition 1.4 (X-completeness)**.** *Given a graph $G = (V, E)$ and a set $X \subseteq V$, vertex $v \in V(G) \setminus X$ is* X-complete *if and only if it is adjacent to every vertex $x \in X$. A set $Y \subseteq V$ is X-complete if and only if $X \cap Y = \emptyset$ and every vertex $y \in Y$ is X-complete.*

For example, for $X = \{v_2\}$, the set $\{v_1, v_3\}$ is X-complete in G, while the set $\{v_3, v_4\}$ is not.

Definition 1.5 (path)**.** *A* path *is a graph P of the form*

$$V(P) = \{x_1, x_2, \ldots, x_l\}, \quad E(P) = \{x_1x_2, x_2x_3, \ldots, x_{l-1}x_l\}$$

This path P is usually denoted by $x_1x_2 \ldots x_l$ or x_1-x_2-…-x_l. The vertices x_1 and x_l are the *endvertices* and $l - 1 = |E(P)|$ is the *length* of the path P. $\{x_2, \ldots x_{l-1}\}$ is the *inside* of the path P, denoted as P^*. Notice that we don't allow any edges other than the ones between consecutive vertices for a graph to be called a path.

Our graph G_0 is a path of length 3, with the inside $G_0^* = \{v_2, v_3\}$. If we added any edge to G_0 it would stop being a path.

Definition 1.6 (connected graph, subset)**.** *A graph G is* connected *if and only if for every pair $\{x, y\} \subseteq V(G)$ of distinct vertices, there is a path from x to y. A subset $X \subseteq V(G)$ is connected if and only if the graph $G[X]$ is connected.*

Definition 1.7 (component)**.** *A* component *of a graph G is its maximal connected induced subgraph.*

Definition 1.8 (cycle)**.** *A* cycle *is a graph C of the form*

$$V(C) = \{x_1, x_2, \ldots, x_l\}, \quad E(C) = \{x_1x_2, x_2x_3, \ldots, x_{l-1}x_l, x_lx_1\}$$

This cycle C is usually denoted by $x_1x_2 \ldots x_lx_1$ or x_1-x_2-…-x_l-x_1. $l = |E(C)|$ is the *length* of the cycle C. Sometimes we will denote the cycle of length l as C_l.

Definition 1.9 (hole)**.** *A* hole *is a cycle of length at least four.*

If a path, a cycle or a hole has an odd length, it will be called *odd*. Otherwise, it will be called *even*. Notice that if we add an edge v_1v_4 to the path G_0 it becomes an even cycle C_4.

Definition 1.10 (complement)**.** *A* complement *of a graph $G = (V, E)$ is a graph $\overline{G} = (V, \binom{V}{2} \setminus E)$, that is, two vertices x, y are adjacent in $\overline{G}$ if and only if they are not adjacent in G.*

Definition 1.11 (anticonnected graph, subset)**.** *A graph G is* anticonnected *if and only if $\overline{G}$ is connected. A subset X is* anticonnected *if and only if $\overline{G}[X]$ is connected.*

Definition 1.12 (anticomponent)**.** *An* anticomponent *of a graph G is an induced subgraph whose complement is a component in $\overline{G}$.*

Definition 1.13 (antipath)**.** *An* antipath *is a graph G such that $\overline{G}$ is a path.*

Definition 1.14 (antihole)**.** *An* antihole *is a graph G such that $\overline{G}$ is a hole.*

Definition 1.15 (clique)**.** *A* complete graph *or a* clique *is a graph of the form $G = (V, \binom{V}{2})$, that is, every two vertices are connected. We will denote a clique on n vertices as K_n.*

Definition 1.16 (clique number)**.** *A* clique number *of a graph G, denoted as $\omega(G)$, is a cardinality of its largest induced clique.*

Definition 1.17 (anticlique)**.** *An* independent set *or an* anticlique *is a graph of the form $G = (V, \emptyset)$, that is no vertices are connected.*

In a similar fashion, given a graph $G = (V, E)$, a subset of its vertices $V' \subseteq V$ will be called *independent* (in the context of G) if and only if $G[V']$ is an anticlique.

Definition 1.18 (stability number)**.** *A* stability number *of a graph G, denoted as $\alpha(G)$, is a cardinality of its largest induced stable set.*

Definition 1.19 (coloring)**.** *Given a graph G, its* coloring *is a function $c : V(G) \to \mathbb{N}^+$, such that for every edge $xy \in E(G)$, $c(x) \neq c(y)$. A k-coloring of G (if exists) is a coloring, such that $c(x) \leq k$ for all vertices $x \in V(G)$.*

Definition 1.20 (chromatic number)**.** *A* chromatic number *of a graph G, denoted as $\chi(G)$, is a smallest natural number k, for which there exists a k-coloring of G.*

Definition 1.21 (line graph)**.** *The* line graph *of a graph $G = (V, E)$ is the graph $L(G)$ with $V(L(G)) = E$ and $E(L(G)) = \{e_1e_2 : e_1, e_2 \in E, e_1 \cap e_2 \neq \emptyset\}$, that is, $e_1, e_2 \in E$ are adjacent if and only if they share an endpoint in G.*

Chapter 2

Perfect Graphs

Given a graph G, let us consider a problem of coloring it using as few colors as possible. If G contains a clique K as a subgraph, we must use at least $|V(K)|$ colors to color it. This gives us a lower bound for a chromatic number $\chi(G)$ – it is always greater or equal to the cardinality of the largest clique $\omega(G)$. The reverse is not always true, in fact we can construct a graph with no triangles and requiring arbitrarily large numbers of colors (e.g. construction by Mycielski [41]).

Do graphs that admit coloring using only $\omega(G)$ color are "simpler" to further analyze? Not necessarily so. Given a graph $G = (V, E)$, let us construct a graph G' as the union of G and a clique $K_{|V|}$. We can see that indeed $\chi(G') = \omega(G') = |V|$, but it gives us no indication of the structure of G or G'. This leads us to the hereditary definition of *perfect graphs*.

Definition 2.1 (perfect graph)**.** *A graph G is* perfect *if and only if for its every induced subgraph H we have $\chi(H) = \omega(H)$.*

The notion of perfect graphs was first introduced by Berge in 1961 [5] and it indeed captures some of the idea of graph being "simple" – in all perfect graphs the coloring problem, maximum (weighted) clique problem, and maximum (weighted) independent set problem can be solved in polynomial time [24]. Other classical NP-complete problems remain hard in perfect graphs e.g. Hamiltonian path [40], maximum cut problem [6] or dominating set problem [21]. There are many well-known subclasses of perfect graphs, we take a look and benchmark our implementations on some of them in Section 4.4

The most fundamental problem – the problem of recognizing perfect graphs – was open since its posing in 1961 until recently. Its solution, a polynomial algorithm recognizing perfect graphs, is a union of the strong perfect graph theorem (Section 2.1) stating that a graph is perfect if and only if it is Berge and an algorithm for recognizing Berge graphs in polynomial time (Section 2.2).

Definition 2.2 (Berge graph)**.** *A graph G is* Berge *if and only if both G and $\overline{G}$ have no odd hole as an induced subgraph.*

Perfect graphs are interesting not only because of their theoretical properties, but they are also used in other areas of study e.g. integrality of polyhedra [16, 14], radio channel assignment problem [37, 38] and appear in the theory of Shannon capacity of a graph [30]. Also, as pointed out in [43, 14], algorithms to solve semidefinite programs grew out of the theory of perfect graphs. We will take a look at semidefinite programs and at perfect graph's relation to the Shannon capacity in Section 3.1.1.

2.1 Strong perfect graph theorem

The first step to solve the problem of recognizing perfect graphs was the *(weak) perfect graph theorem* first conjured by Berge in 1961 [5] and then proven by Lovász in 1972 [31].

Theorem 2.1.1 (Perfect graph theorem [31])**.** *A graph is perfect if and only if its complement graph is also perfect.*

This theorem is a consequence of a stronger result proven by Lovász:

Theorem 2.1.2 ([31], Claim 1 of [19])**.** *A graph G is perfect if and only if for every induced subgraph H, the number of vertices of H is at most $\alpha(H)\omega(H)$.*

Then, since $\alpha(H) = \omega(\overline{H})$ and $\omega(H) = \alpha(\overline{H})$ Theorem 2.1.2 implies Theorem 2.1.1. We skip the proof of Theorem 2.1.2, as it is quite technical.

Odd holes are not perfect, since their chromatic number is 3 and their largest cliques are of size 2. It is also easy to see, that an odd antihole of size n has a chromatic number of $\frac{n+1}{2}$ and largest cliques of size $\frac{n-1}{2}$. A graph with no odd hole and no odd antihole is called *Berge* (Definition 2.2) after Claude Berge who studied perfect graphs.

In 1961 Berge conjured that a graph is perfect if and only if it contains no odd hole and no odd antihole in what has become known as a strong perfect graph conjecture. In 2001 Chudnovsky et al. have proven it and published the proof in an over 150 pages long paper "The strong perfect graph theorem" [15].

Theorem 2.1.3 (Strong perfect graph theorem, Theorem 1.1 of [15])**.** *A graph is perfect if and only if it is Berge.*

The proof is long and complicated. Moreover, it has little noticeable connection to the algorithm of recognizing Berge graphs we discuss later. Therefore we will discuss it very briefly following the overview by Cornuéjols [19].

Basic classes of perfect graphs

Bipartite graphs are perfect, since we can color them with two colors. From the theorem of König [26] we get that line graphs of bipartite graphs are also perfect [19]. From the perfect graph theorem (Theorem 2.1.1) it follows that complements of bipartite graphs and complement of line graphs of bipartite graphs are also perfect. We will call these four classes *basic*.

2-join, Homogeneous Pair and Skew Partition

A graph G has a *2-join* if and only if its vertices can be partitioned into sets V_1, V_2, each of size at least three, and there are nonempty disjoint subsets $A_1, B_1 \subseteq V_1$ and $A_2, B_2 \subseteq V_2$, such that all vertices of A_1 are adjacent to all vertices of A_2, all vertices of B_1 are adjacent to all vertices of B_2, and these are the only edges between V_1 and V_2. When a graph G has a 2-join, it can be decomposed onto two smaller graphs G_1, G_2, so that G is perfect if and only if G_1 and G_2 are perfect [20].

A graph G has a *homogeneous pair* if $V(G)$ can be partitioned into subsets A_1, A_2, B, such that $|A_1| + |A_2| \geq 3$, $|B| \geq 2$ and if a vertex $v \in B$ is adjacent to a vertex from A_i, then it is adjacent to all vertices from A_i. Chvátal and Sbihi proved that no minimally imperfect graph has a homogeneous pair [17].

A graph G has a *skew partition* if $V(G)$ can be partitioned into nonempty subsets A, B, C, D such that there are all possible edges between A and B and no edges between C and D. Chudnovsky et at. proved that no minimally imperfect graph has a skew partition.

The proof of Theorem 2.1.3 is a consequence of the *Decomposition theorem*:

Theorem 2.1.4 (Decomposition theorem, stated as Theorem 4.2 of [19])**.** *Every Berge graph G is basic or has a skew partition or a homogeneous pair, or either G or $\overline{G}$ has a 2-join.*

See [15] for the full proof of Theorem 2.1.3 and [19] for its shorter overview.

2.2 Recognizing berge graphs

The following section is based on the paper by Chudnovsky et al. "Recognizing Berge Graphs" [11]. We will not provide full proof of its correctness, but we will aim to show the intuition behind the algorithm.

Berge graph recognition algorithm (later called CCLSV from the names of its authors) could be divided into two parts: first we check if either G or $\overline{G}$ contain any of a number of simple forbidden structures (Section 2.2.1). If they do, we output that graph is not Berge and stop. Else, we check if there is a near-cleaner for a shortest odd hole or for shortest odd antihole (Section 2.2.2). If we find no such near-cleaners, then, as we will see, the graph is Berge.

2.2.1 Simple forbidden structures

Pyramids

A *pyramid* in G is an induced subgraph formed by the union of a triangle [1] $\{b_1, b_2, b_3\}$, three paths $\{P_1, P_2, P_3\}$ and another vertex a, so that:

[1] A triangle is a clique K_3.

- $\forall_{1 \le i \le 3}$ P_i is a path between a and b_i,
- $\forall_{1 \le i < j \le 3}$ a is the only vertex in both P_i and P_j and $b_i b_j$ is the only edge between $V(P_i) \setminus \{a\}$ and $V(P_j) \setminus \{a\}$,
- a is adjacent to at most one of $\{b_1, b_2, b_3\}$.

We will say that a is *linked onto* the triangle $\{b_1, b_2, b_3\}$ *via* the paths P_1, P_2, P_3. Let us notice, that a pyramid is uniquely determined by its paths P_1, P_2, P_3.

It is easy to see that every graph containing a pyramid contains an odd hole – at least two of the paths P_1, P_2, P_3 will have the same parity.

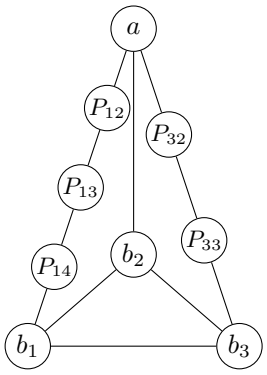


Figure 2.1: An example of a pyramid.

Finding Pyramids.

Algorithm 2.2.1 (Test if G contains a pyramid)
Input: A graph G.
Output: Returns whether G contains a pyramid as an induced subgraph.

```
1: procedure CONTAINS-PYRAMID(G)
2:   for each triangle b1, b2, b3 do
3:     for each s1, s2, s3, such that for 1 ≤ i < j ≤ 3, {bi, si} is disjoint
         from {bj, sj} and bi bj is the only edge between them do
4:       if there is a vertex a, adjacent to all of s1, s2, s3, and to at most
           one of b1, b2, b3, such that if a is adjacent to bi, then bi = si
           then
5:         M ← V(G) \ {b1, b2, b3, s1, s2, s3}
6:         for each m ∈ M do
7:           S1(m) ← the shortest path between s1 and m such that
               s2, s3, b2, b3 have no neighbors in its interior, if such a
               path exists.
8:           calculate S2(m), S3(m) similarly
9:           T1(m) ← the shortest path between m and b1, such that
               s2, s3, b2, b3 have no neighbors in its interior, if such a
               path exists
10:          calculate T2(m), T3(m) similarly
11:        end for
12:        if s1 = b1 then                    \\ calculate all possible P1 paths
13:          P1(b1) ← the one-vertex path b1
14:          for each m ∈ M do P1(m) ← UNDEFINED
15:        else
16:          P1(b1) ← UNDEFINED
17:          for each m ∈ M do
18:            if m is nonadjacent to all of b2, b3, s2, s3 and
                 S1(m) and T1(m) both exist and
```

```
                                    V(S_1(m) ∩ T_1(m)) = {m}  and
                                    there are no edges between V(S_1(m) \ m)
                                    and V(T_1(m) \ m)
                                     then
19:                                 P_1(m) ← s_1-S_1(m)-m-T_1(m)-b_1
20:                               else
21:                                 P_1(m) ← UNDEFINED
22:                               end if
23:                           end for
24:                       end if
25:                       assign P_2 and P_3 in a similar manner
26:                       good_pairs_{1,2} ← ∅                    \\ see below for definition
27:                       for each m_1 ∈ M ∪ {b_1} do     \\ calculate good (1,2)-pairs
28:                           if P_1(m_1) ≠ UNDEFINED then
29:                               color black the vertices of M that either belong to
                                    P_1(m_1) or have a neighbor in P_1(m_1)
30:                               color all other vertices white.
31:                               for each m_2 ∈ M ∪ {b_2} do
32:                                   if P_2(m_2) exists and contains no black vertices
                                          then
33:                                       add (m_1, m_2) to good_pairs_{1,2}
34:                                   end if
35:                               end for
36:                           end if
37:                       end for
38:                       calculate good_pairs_{1,3} and good_pairs_{2,3} in similar way
39:                       for each triple m_1, m_2, m_3 such that m_i ∈ M ∪ {b_i} do
40:                           if ∀_{1≤i<j≤3}: (m_i, m_j) ∈ good_pairs_{i,2} then
41:                               return TRUE
42:                           end if
43:                       end for
44:                   end if
45:               end for
46:           end for
47:           return FALSE
48: end procedure
```

With definitions as above, for $1 \leq i < j \leq 3$, we say that (m_i, m_j) is a *good* (i, j)*-pair*, if and only if $m_i \in M \cup \{b_i\}$, $m_j \in M \cup \{b_j\}$, $P_i(m_i)$ and $P_j(m_j)$ both exist, and the sets $V(P_i(m_i))$,$V(P_j(m_j))$ are both disjoint and $b_i b_j$ is the only edge between them. In line 29 we color vertices of $P_1(m_1)$ black, so that for each m_2 we can check if paths $P_1(m_1)$ and $P_2(m_2)$ are disjoint in $O(|V|)$ time.

It is easy to see, that if CONTAINS-PYRAMID(G) outputs that G contains a pyramid, it indeed does – when we return in line 41 the vertex a found in line 4 can be linked into a triangle b_1, b_2, b_3 via paths $P_1(m_1), P_2(m_2), P_3(m_3)$ for m_1,

m_2, m_3 from line 39. The proof of the converse is rather technical and we refer to Theorem 2.2 of [11] for it.

Now we will prove the time complexity.

Theorem 2.2.1 (part of Theorem 2.2 of [11])**.** *Procedure* CONTAINS-PYRAMID(G) *works in* $O(|V|^9)$ *time.*

Proof. There are $O(|V|^3)$ triangles (line 2) and $O(|V|^3)$ triples s_1, s_2, s_3 (line 3), so lines 4-44 are executed at most $O(|V|^6)$ times.

Checking if there exists an appropriate a takes linear time (line 4). Calculating paths S_i and T_i (lines 6-11) takes $O(|V|^2)$ time for each $m \in M$ and $O(|V|^3)$ in total. Similarly, it takes $O(|V|^3)$ time to calculate all P_i paths (lines 12-25).

Then, for each m_1 we do at most $O(|V|)$ work in line 29 and for each (m_1, m_2) we do at most $O(|V|)$ work in line 32.

Finally, there are $O(|V|^3)$ pairs m_1, m_2, m_3 and checking each takes $O(1)$ time which gives us the overall running time of $O(|V|^9)$. □

Jewels

Five vertices $v_1, \ldots, v_5$ and a path P form a *jewel* if and only if:

- $v_1, \ldots, v_5$ are distinct vertices,
- $v_1v_2, v_2v_3, v_3v_4, v_4v_5, v_5v_1$ are edges,
- v_1v_3, v_2v_4, v_1, v_4 are nonedges,
- P is a path between v_1 and v_4, such that v_2, v_3, v_5 have no neighbors in its inside.

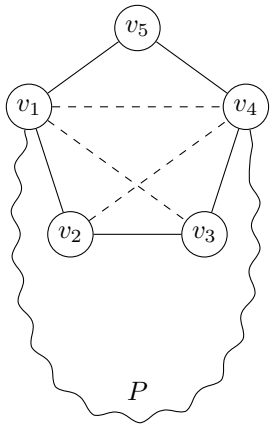


Figure 2.2: An example of a jewel.

Notice that given a jewel $v_1, \ldots, v_5, P$, if the path P is odd, then v_1-P-v_4-v_5-v_1 is an odd hole. If the path P is even, then v_1-P-v_4-v_3-v_2-v_1 is an odd hole.

Most obvious way to find a jewel is to enumerate all (possibly chordal) cycles of length 5 as $v_1, \ldots v_5$, check if it has all required nonedges and if it does, try to find a path P as required. This gives us a time of $O(|V|^7)$. We could speed it up to $O(|V|^6)$ with more careful algorithm (see Algorithm 3.1 of [11]), but since whole Berge recognition algorithm takes time $O(|V|^9)$ and our testing showed that time it takes to test for jewels is negligible we decided against it.

Configurations of type $\mathcal{T}_1$

A configuration of type $\mathcal{T}_1$ is a hole of length 5. To find it, we simply iterate over all paths of length 4 and check if there exists a fifth vertex to complete the hole. See Section 4.1 for more implementation details.

Configurations of type $\mathcal{T}_2$

A configuration of type $\mathcal{T}_2$ is a tuple $(v_1, v_2, v_3, v_4, P, X)$, such that:

- $v_1v_2v_3v_4$ is a path in G,
- X is an anticomponent of the set of all $\{v_1, v_2, v_4\}$-complete vertices,
- P is a path in $G \setminus (X \cup \{v_2, v_3\})$ between v_1 and v_4 and no vertex in P^* is X-complete or adjacent to v_2 or adjacent to v_3.

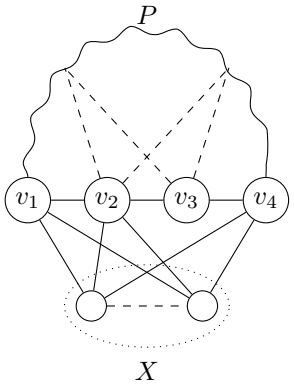


Figure 2.3: An example of a $\mathcal{T}_2$.

Checking if configuration of type $\mathcal{T}_2$ exists in our graph is straightforward: we enumerate all paths $v_1 \dots v_4$, calculate set of all $\{v_1, v_2, v_4\}$-complete vertices and its anticomponents. Then, for each anticomponent X we check if required path P exists.

To prove that the existence of a configuration of type $\mathcal{T}_2$ implies that the graph is not Berge, we will need the following lemma:

Lemma 2.2.2 (Roussel-Rubio Lemma [44, 11])**.** *Let G be Berge, X be an anticonnected subset of $V(G)$, P be an odd path $p_1 \dots p_n$ in $G \setminus X$ with length at least 3, such that p_1 and p_n are X-complete and $p_2, \dots, p_{n-1}$ are not. Then:*

- *P is of length at least 5 and there exist nonadjacent $x, y \in X$, such that there are exactly two edges between x, y and P^*, namely xp_2 and yp_{n-1},*
- *or P is of length 3 and there is an odd antipath joining internal vertices of P with interior in X.*

Now, we shall prove the following:

Theorem 2.2.3 (Theorem 6.3 of [11])**.** *If G contains configuration of type $\mathcal{T}_2$ then G is not Berge.*

Proof. Let $(v_1, v_2, v_3, v_4, P, X)$ be a configuration of type $\mathcal{T}_2$. Let us assume that G is not Berge and consider the following:

- if P is even, then $v_1, v_2, v_3, v_4, P, v_1$ is an odd hole,
- if P is of length 3, let us name its vertices v_1, p_2, p_3, v_4. It follows from Lemma 2.2.2, that there exists an odd antipath between p_2 and p_3 with interior in X. We can complete it with v_2p_2 and v_2p_3 into an odd antihole,
- if P is odd with the length of at least 5, it follows from Lemma 2.2.2 that we have $x, y \in X$ with only two edges to P being xp_2 and yp_{n-1}. This gives us an odd hole: v_2-x-p_2-$\dots$-p_{n-1}-y-v_2.

□

Configurations of type $\mathcal{T}_3$

A configuration of type $\mathcal{T}_3$ is a sequence $v_1, \ldots, v_6$, P, X, such that:

- $v_1, \ldots v_6$ are distinct vertices,
- v_1v_2, v_3v_4, v_1v_4, v_2v_3, v_3v_5, v_4v_6 are edges, and v_1v_3, v_2v_4, v_1v_5, v_2v_5, v_1v_6, v_2v_6, v_4v_5 are nonedges,
- X is an anticomponent of the set of all $\{v_1, v_2, v_5\}$-complete vertices, and v_3, v_4 are not X-complete,
- P is a path of $G \setminus (X \cup \{v_1, v_2, v_3, v_4\})$ between v_5 and v_6 and no vertex in $P*$ is X-complete or adjacent to v_1 or adjacent to v_2,
- If v_5v_6 is an edge, then v_6 is not X-complete.

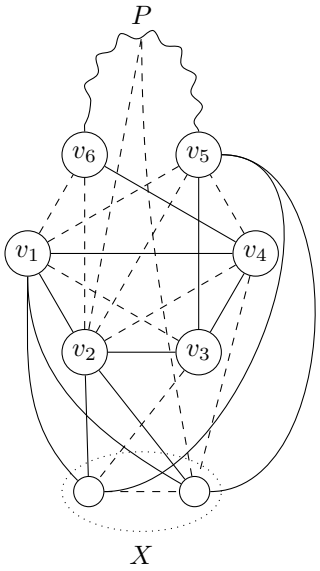


Figure 2.4: An example of a $\mathcal{T}_3$.

Algorithm 2.2.2 (Test if G contains a $\mathcal{T}_3$)
Input: A graph G.
Output: Returns whether G contains a configuration of type $\mathcal{T}_3$ as an induced subgraph.

```
1:  procedure Contains-T3(G)
2:      for each v1, v2, v5 ∈ V(G), so that v1v2 is an edge and
            v1v5, v2v5 are nonedges do
3:          Y ← the set of all {v1, v2, v5}-complete vertices.
4:          for each X – an anticomponent of Y do
5:              F' ← maximal connected subset containing v5, such that v1, v2
                 have no neighbors in F' and no vertex of F' \ {v5} is X-complete.
6:              F'' ← the set of all X-complete vertices that have a neighbor in
                 F' and are nonadjacent to all of v1, v2 and v5
7:              F ← F' ∪ F''
8:              for each v4 ∈ V(G) \ {v1, v2, v5}, such that v4 is adjacent to v1
                    and not to v2 and v5 do
9:                  if v4 has a neighbor in F and a nonneighbor in X then
10:                     v6 ← a neighbor of v4 in F
11:                     for each v3 ∈ V(G) \ {v1, v2, v4, v5, v6} do
12:                         if v3 is adjacent to v2, v4, v5 and not adjacent to v1
                                then
13:                             P ← a path from v6 to v5 with interior in F'
14:                             return TRUE          \\ v1, ... v6, P, X is a T3
15:                         end if
16:                     end for
17:                 end if
```

```
18:  |  |  |  end for
19:  |  |  end for
20:  |  end for
21:  |  return FALSE
22: end procedure
```

It is easy to see that when Algorithm 2.2.2 reports a configuration of type $\mathcal{T}_3$ in line 14, there indeed is one. We will skip the proof that each graph containing a $\mathcal{T}_3$ is not Berge, as it is quite technical. See section 6.7 of [11] for the proof.

Theorem 2.2.4 (part of Theorem 6.4 of [11])**.** *Algorithm 2.2.2 runs in $O(|V|^6)$ time.*

Proof. There are $O(|V|^3)$ triples (v_1, v_2, v_5) we enumerate in line 2. For each, there are $O(|V|)$ Xs, so lines 5-18 are executed $O(|V|^4)$ times.

Each execution of lines 5-7 takes $O(|V|^2)$ time, and there are $O(|V|)$ v_4s to enumerate. Line 9 takes $O(|V|)$ time and then there are $O(|V|)$ v_3 to enumerate. Each is checked in $O(1)$ time. If we find the correct one, we calculate a path P, but return right away, so we calculate P at most once in whole algorithm. □

When graphs G and $\overline{G}$ contain no pyramids, no jewels and no configurations of type $\mathcal{T}_1, \mathcal{T}_2$ or $\mathcal{T}_3$, we will say that graph G contains no *simple forbidden structures*.

2.2.2 Amenable holes.

As we will see (Theorem 2.2.5), if graph G contains no simple forbidden structures, then every shortest odd hole in G or $\overline{G}$ will have a special structure and we will call it amenable. This fact will help us determine if G or $\overline{G}$ contain an odd hole. But first, let us introduce a few new definitions.

Definition 2.3 (relevant triple)**.** *Given a graph G, a triple (a, b, c) of vertices is called* relevant *if and only if $a \neq b$ (but possibly $c \in \{a, b\}$) and $G[\{a, b, c\}]$ is an independent set.*

Definition 2.4 (C-major vertices)**.** *Given a shortest odd hole C in G, a vertex $v \in V(G) \setminus V(C)$ is C-major if and only if the set of its neighbors in C is not contained in any 3-vertex path of C.*

Definition 2.5 (clean odd hole)**.** *An odd hole C of G is* clean *if and only if no vertex in G is C-major.*

Definition 2.6 (cleaner)**.** *Given a shortest odd hole C in G, a subset $X \subseteq V(G)$ is a* cleaner for C *if and only if $X \cap V(C) = \emptyset$ and every C-major vertex belongs to X.*

Let us notice, that if X is a cleaner for C, then C is a clean hole in $G \setminus X$.

Definition 2.7 (near-cleaner)**.** *Given a shortest odd hole C in G, a subset $X \subseteq V(G)$ is a* near-cleaner for C *if and only if X contains all C-major vertices and $X \cap V(C)$ is a subset of vertex set of some 3-vertex path of C.*

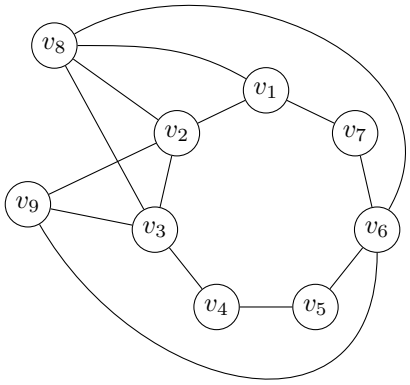


Figure 2.5: Amenable odd hole

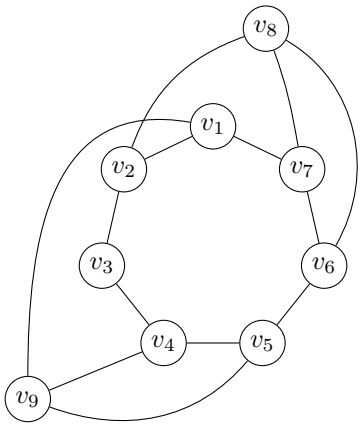


Figure 2.6: Not amenable odd hole

Definition 2.8 (amenable odd hole)**.** *An odd hole C of G is* amenable *if and only if it is a shortest odd hole in G, $|E(C)| \geq 7$, and for every anticonnected set X of C-major vertices there is a X-complete edge in C.*

Some pictures might be helpful. Figure 2.5 shows an example of an amenable odd hole $C = v_1$-...-v_7-v_1. It is not a clean hole – vertices v_8 and v_9 are C-major, and $\{v_8, v_9, v_1, v_2, v_3\}$ is an example of a near-cleaner for C. Figure 2.6 shows an odd hole $C = v_1$-...-v_7-v_1 that is not amenable, because there is no $\{v_8, v_9\}$-complete edge in C.

Theorem 2.2.5 (Theorem 8.1 of [11])**.** *Let G be a graph, such that G and $\overline{G}$ contain no pyramids, no jewels and no configurations of types $\mathcal{T}_1, \mathcal{T}_2$ or $\mathcal{T}_3$. Then every shortest odd hole in G is amenable.*

The proof of this theorem is quite technical and we will not discuss it here. See section 8 of [11] for the proof. We can see an example of the contraposition of the Theorem 2.2.5 working in Figure 2.6 – vertices v_1, v_2, v_8, v_6, v_7 and a path v_1-v_9-v_5-v_6 form a jewel.

With Theorem 2.2.5 we can describe the rest of the algorithm.

Algorithm 2.2.3 (List possible near cleaners)
Input: A graph G containing no simple forbidden structures.
Output: $O(|V|^5)$ subsets of $V(G)$, such that if C is an amenable hole in G, then one of the subsets is a near-cleaner for C.

1: **procedure** List-Possible-Near-Cleaners(G)
2: | $\mathcal{X} \leftarrow \emptyset$
3: | **for each** relevant triple (a, b, c) **do**

```
4:      A ← anticomponents of N({a, b})
5:      if c is N({a, b})-complete then
6:          r ← 0
7:      else
8:          r ← the cardinality of the largest set of A that contains
              a nonneibhbor of c
9:      end if
10:     Y ← ∅
11:     for each A_i ∈ A do
12:         if |V(A_i)| > r then
13:             Y ← Y ∪ A_i
14:         end if
15:         if c ∈ A_i then
16:             W ← A_i
17:         end if
18:     end for
19:     Z ← the set of all (Y ∪ W)-complete vertices
20:     X(a, b, c) ← Y ∪ Z    \\ we use X(a, b, c) in the proof of correctness
21:     X ← X ∪ X(a, b, c)
22: end for
23: N ← ∅
24: for each edge uv ∈ E(G) do
25:     N ← N ∪ N({u, v})
26: end for
27: R ← ∅
28: for each X_i ∈ X do
29:     for each N_j ∈ N do
30:         R ← R ∪ (X_i ∪ N_j)
31:     end for
32: end for
33: return R
34: end procedure
```

To prove the correctness of Algorithm 2.2.3 we will need the following theorem.

Theorem 2.2.6 (Theorem 9.1 of [11]). *Let C be a shortest odd hole in G, with length at least seven. Then there is a relevant triple (a, b, c) of vertices such that*

- *the set of all C-major vertices not in $X(a, b, c)$ is anticonnected,*
- *$X(a, b, c) \cap V(C)$ is a subset of the vertex set of some 3-vertex path of C.*

Theorem 2.2.7 (Theorem 9.2 of [11]). *Let G be a graph, such that G and $\overline{G}$ contain no pyramids, no jewels and no configurations of types $\mathcal{T}_1, \mathcal{T}_2$ or $\mathcal{T}_3$. Then the result of the Algorithm 2.2.3 on the graph G is a set $\mathcal{R}$, such that for every shortest amenable hole C, there is a near-cleaner for C in $\mathcal{R}$.*

Proof. Let us suppose that C is an amenable hole in G. By Theorem 2.2.6, there is a relevant triple (a, b, c) satisfying that theorem. Let T be the set of all C-major vertices not in $X(a, b, c)$. From Theorem 2.2.6 we get that T is anticonnected. Since C is amenable, there is an edge uv of C that is T-complete, and therefore $T \subseteq N(u, v)$. But then $N(u, v) \cup X(a, b, c) \in \mathcal{R}$ is a near-cleaner for C. Therefore the output of the Algorithm 2.2.3 is correct. □

Algorithm 2.2.4 (Test possible near cleaner)
Input: A graph G containing no simple forbidden structures, and a subset $X \subseteq V(G)$.
Output: Determines one of the following:

- *G has an odd hole,*
- *There is no shortest odd hole C such that X is a near-cleaner for C.*

```
1:  procedure TEST-NEAR-CLEANER(G, X)
2:      for each pair x, y ∈ V(G), such that x ≠ y do
3:          if there is a path between x and y with no internal vertices in X
                then
4:              R(x, y) ← the shortest path R(x, y) between x, y with no
                  internal vertex in X.
5:              r(x, y) ← the length of R(x, y)
6:          else
7:              R(x, y) ← UNDEFINED
8:              r(x, y) ← ∞
9:          end if
10:         for each y1 ∈ V(G) \ X do
11:             for each 3-vertex path x1-x2-x3 of G \ y1 do
12:                 if R(x1, y1), R(x2, y2) both exist then
13:                     y2 ← the neighbor of y1 in R(x2, y1)
14:                     if r(x2, y1) = r(x1, y1) + 1 = r(x1, y2)  and
                            r(x2, y1) ≤ min(r(x3, y1), r(x3, y2))  then
15:                         return an odd hole found
16:                     end if
17:                 end if
18:             end for
19:         end for
20:     end for
21:     return no odd hole found
22: end procedure
```

Theorem 2.2.8 (Theorem 5.1 of [11])**.** *Let G be a graph, such that G and $\overline{G}$ contain no pyramids, no jewels and no configurations of types $\mathcal{T}_1, \mathcal{T}_2$ or $\mathcal{T}_3$. Let X be a subset of $V(G)$. If Algorithm 2.2.5 reports an odd hole, then G contains an odd hole, else there is no shortest odd hole C in G, such that X is a near-cleaner for C.*

Proof. Below we will prove that if the Algorithm 2.2.5 reports an odd hole in G, there indeed is one. The rest of the proof is more complicated, see section 4 and Theorem 5.1 of [11].

Let us suppose that there is a choice of x_1, x_2, x_3, y_1 satisfying conditions in lines 12 and 14, let y_2 be defined as in line 13 and let $k = r(x_2, y_1)$. We claim that G contains an odd hole.

Let $R(x_1, y_1) = p_1$-...-p_k, and let $R(x_2, y_1) = q_1$-...-q_{k+1}, where $p_1 = x_1$, $p_k = q_{k+1} = y_1$, $q_1 = x_2$ and $q_k = y_2$. From the definition of $R(x_1, y_1)$ and $R(x_2, y_1)$, none of $p_2, \ldots, p_{k-1}, q_2, \ldots, q_k$ belong to X. Also, from the definition of y_1, $y_1 \notin X$.

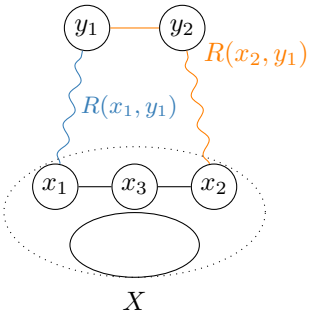


Figure 2.7: An odd hole is found

Since $r(x_1, y_1) = r(x_2, y_1) - 1$ and since x_1, x_2 are nonadjacent it follows that x_2 does not belong to $R(x_1, y_1)$ and x_1 does not belong to $R(x_2, y_1)$. Since $r(x_3, y_1), r(x_3, y_2) \geq k = r(x_1, y_2)$ it follows that x_3 does not belong to $R(x_1, y_1)$ or to $R(x_2, y_1)$, and has no neighbors in $R(x_1, y_1) \cup R(x_2, y_1)$ other than x_1, x_2. Since $r(x_1, y_2) = k$ we get that y_2 does not belong to $R(x_1, y_1)$.

We claim first that the insides of paths $R(x_1, y_1)$ and $R(x_2, y_1)$ have no common vertices. For suppose that there are $2 \leq i \leq k-1$ and $2 \leq j \leq k$ that $p_i = q_j$. Then the subpaths of these two paths between p_i, y_1 are both subpaths of the shortest paths, and therefore have the same length equal to $j = i + 1$. So p_1-...-p_i-q_{j+1}-...-q_k contains a path between x_1, y_2 of length $\leq k-2$, contradicting that $r(x_1, y_2) = k$. So $R(x_1, y_1)$ and $R(x_2, y_1)$ have no common vertices except y_1.

If there are no edges between $R(x_1, y_1) \setminus \{y_1\}$ and $R(x_2, y_1) \setminus \{y_1\}$ then the union of these two paths and a path x_1-x_3-x_2 forms an odd hole, so the answer is correct.

Suppose that $p_i q_j$ is an edge for some $1 \leq i \leq k-1$ and $1 \leq j \leq k$. We claim $i \geq j$. If $j = 1$ this is clear so let us assume $j > 1$. Then there is a path between x_1, y_2 within $\{p_1, \ldots, p_i, q_j, \ldots q_k\}$ which has length $\leq k - j + 1$ and has no internal vertex in X (since $j > 1$); and since $r(x_1, y_2) = k$, it follows that $k - j + i \geq k$, that is, $i \geq j$ as claimed.

Now, since x_1, x_2 are nonadjacent, we have $i \geq 2$. But also $r(x_2, y_1) \geq k$ and so $j + k - i \geq k$, which gives us $j \geq i$. So we get $i = j$. Let us choose i minimum, then x_3-x_1-...-p_i-q_i-...-x_2-x_3 is an odd hole, which was what we wanted. □

2.2.3 Algorithm for perfect graph recognition

To summarize, we present a complete algorithm for recognizing perfect graphs.

Algorithm 2.2.5 (Determine if a graph is perfect)
Input: A graph G.

Output: Determines if G is perfect.

```
1: procedure CHECK-IS-PERFECT(G)
2:     if Contains-Pyramid(G) or Contains-Jewel(G) or
           Contains-T1(G) or Contains-T2(G) or Contains-T3(G) then
3:         return G is not perfect
4:     end if
5:     if Contains-Pyramid(Ḡ) or Contains-Jewel(Ḡ) or
           Contains-T1(Ḡ) or Contains-T2(Ḡ) or Contains-T3(Ḡ) then
6:         return G is not perfect
7:     end if
8:     R ←List-Possible-Near-Cleaners(G)
9:     for each X ∈ R do
10:        if Test-Near-Cleaner(G, X) reports an odd hole then
11:            return G is not perfect
12:        end if
13:    end for
14:    R' ←List-Possible-Near-Cleaners(Ḡ)
15:    for each X ∈ R' do
16:        if Test-Near-Cleaner(Ḡ, X) reports an odd hole then
17:            return G is not perfect
18:        end if
19:    end for
20:    return G is perfect
21: end procedure
```

Theorem 2.2.9 (Theorem 10.1 of [11])**.** *Given a graph $G = (V, E)$, Algorithm 2.2.5 works in $O(|V|^9)$ time and determines if G is perfect.*

Proof. First, in lines 2-7 we test whether G or $\overline{G}$ contain a pyramid, a jewel or a configuration of type $\mathcal{T}_1, \mathcal{T}_2$ or $\mathcal{T}_3$. If it is not the case, then by Theorem 2.2.5 every shortest odd hole in G and in $\overline{G}$ is amenable.

By Theorem 2.2.7 in line 8 we get a set $\mathcal{R}$, such that if G contains an amenable hole C, one of the sets $X \in \mathcal{R}$ is a near-cleaner for C. In lines 9-13 we test each $X \in \mathcal{R}$ and if an odd hole is reported, then by Theorem 2.2.8 G is not Berge. If for all $X \in \mathcal{R}$ no odd holes are reported, then by Theorem 2.2.5, Theorem 2.2.7 and Theorem 2.2.8 there are no odd holes in G. We run the same checks on $\overline{G}$ and if again we do not detect an odd hole, then G is Berge and by Theorem 2.1.3 G is perfect.

Finally, let us add up the total running time. By Theorem 2.2.1 checking if there is a pyramid in G or $\overline{G}$ takes $O(|V|^9)$ time. Testing for jewels takes $O(|V|^7)$ time, testing for a configuration of type $\mathcal{T}_1$ takes $O(|V|^5)$ time and testing for configuration of type $\mathcal{T}_2$ takes $O(|V|^6)$ time. By Theorem 2.2.4 testing for a configuration of type $\mathcal{T}_3$ takes $O(|V|^6)$ time. Running Algorithm 2.2.3 takes $O(|V|^5)$ time and returns $O(|V|^5)$ subsets. For each subset running Algorithm 2.2.5 takes $O(|V|^4)$ time, so testing all possible near-cleaners takes $O(|V|^9)$ time.

The overall running time is $O(|V|^9)$ with testing for pyramids and testing all possible near-cleaners being parts with biggest time complexity. $\square$

Chapter 3

Coloring Perfect Graphs

A natural problem for perfect graphs is a problem of coloring them. In 1988 Grötschel et al. published an ellipsoid-method-based polynomial algorithm for coloring perfect graphs [23]. We consider it in Section 3.3. However due to its use of the ellipsoid method, this algorithm has been usually considered unpractical [13, 14, 28].

There has been much progress on the quest of finding an algorithm coloring perfect graphs, without the use of ellipsoid method (see Section 3.4), however there is still no known polynomial combinatorial algorithm to do this.

3.1 Information theory background

Section 3.1 and Section 3.2 are based on lecture notes by Lovász [32].

The only polynomial technique of coloring perfect graphs known so far arose in the field of semidefinite programming. Semidefinite programs are linear programs over the cones of semidefinite matrices. The connection of coloring graphs and the cones of semidefinite matrices might be surprising, so let us take a brief digression into the field of information theory, where we will see the connection more clearly. Also, this was exactly the background which motivated Berge to introduce perfect graphs [14].

3.1.1 Shannon capacity of a graph

Suppose we have a noisy communication channel in which certain signal values can be confused with others. For instance, suppose our channel has five discrete signal values, represented as 0, 1, 2, 3, 4. However, each value of i when sent across the channel can be confused with value $(i \pm 1)$ mod 5. This situation can be modeled by a graph C_5 (fig. 3.1) in which vertices correspond to signal values and two vertices are connected if and only if the values they represent can be confused.

We are interested in transmission without possibility of confusion. For this example it is possible for two values to be transmitted without ambiguity e.g. values 1 and 4, which allows us to send 2^n non-confoundable messages in n steps. But we could do better, for example we could communicate five two-step codewords e.g. "00", "12", "24", "43", "31". Each pair of these codewords includes at least one position where its values differ by two or more modulo 5, which allows the recipient to distinguish them without confusion. This allows us to send $5^{n/2}$ non-confoundable messages in n steps.

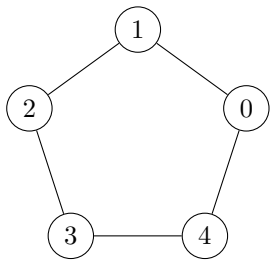


Figure 3.1: An example of a noisy channel

Let us be more precise. Given a graph G modeling a communication channel and a number $k \geq 1$ we say that two messages $v_1v_2 \dots v_k$, $w_1w_2 \dots w_k \in V(G)^k$ of length k are non-confoundable if and only if there is $1 \leq i \leq k$ such that v_i, w_i are non-confoundable. We are interested in the maximum rate at which we can reliably transmit information, that is, the *Shannon capacity* of the channel defined by G.

For $k = 1$, maximum number of messages we can send without confusion in a single step is equal to $\alpha(G)$. To describe longer messages we use *strong product* $G \cdot H$ of two graphs $G = (V, E)$, $H = (W, F)$ as the graph with $V(G \cdot H) = V \times W$, with $(i, u)(j, v) \in E(G \cdot H)$ if and only if $ij \in E$ and $uv \in F$, or $ij \in E$ and $u = v$, or $i = j$ and $uv \in F$. Given channel modeled by G it is easy to see that the maximum number of distinguishable words of length 2 is equal to $\alpha(G \cdot G)$, and in general the number of distinguishable words of length k is equal to $\alpha(G^k)$ which gives us $\sqrt[k]{\alpha(G^k)}$ as the number of distinguishable signals per single transmission. So, we can define the Shannon capacity of the channel defined by G as $\Theta(G) = \sup_k \sqrt[k]{\alpha(G^k)}$.

Unfortunately, it is not known whether $\Theta(G)$ can be computed for all graphs in finite time. If we could calculate $\alpha(G^k)$ for a first few values of k (we will show how to do it in Algorithm 3.3.1) we could have a lower bound on $\Theta(G)$. Let us now turn into search for some usable upper bound.

3.1.2 Lovász number

For a channel defined by a graph C_5, using five messages of length 2 to communicate gives us a lower bound on $\Theta(G)$ equal $\sqrt{5}$ (as does calculating $\sqrt{\alpha(C_5^2)}$).

Consider an "umbrella" in $\mathbb{R}^3$ with the unit vector $e_1 = (1, 0, 0)$ as its "handle" and 5 "ribs" of unit length. Open it up to the point where non-consecutive ribs are orthogonal, i.e., form an angle of 90°. This way we get a representation of C_5 by 5 unit vectors $u_1, \dots, u_5$ so that each u_i forms the same angle with e_1 and any two non-adjacent vertices are represented with orthogonal vectors. See Figure 3.2. We can calculate $e_1^\intercal u_i = 5^{-1/4}$.

It turns out, that we can obtain a similar representation of the vertices of C_5^k by unit vectors $v_i \in \mathbb{R}^{3k}$, so that any two non-adjacent vertices are labeled with orthogonal vectors (this representation is sometimes called the *orthogonal representation* [33]). Moreover, we still get $e_1^\intercal v_i = 5^{-k/4}$ for every $i \in V(C_5^k)$ (the proof is quite technical and we omit it here).

If S is any stable set in C_5^k, then $\{v_i : i \in S\}$ is a set of mutually orthogonal unit vectors, so we could extend S to S', so that $\{v_i : i \in S'\}$ is a basis of $\mathbb{R}^{3k}$. This gives us

$$\sum_{i \in S} (e_1^\intercal v_i)^2 \leq \sum_{i \in S'} (e_1^\intercal v_i)^2 \leq |e_1|^2 = 1$$

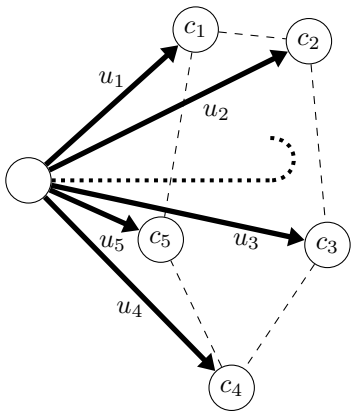


Figure 3.2: Lovász's umbrella

Each term on the left hand side is equal to $5^{-k/4}$, so the left hand side is equal to $|S|5^{-k/2}$, and so $|S| \leq 5^{k/2}$. Since $|S|$ was an arbitrary stable set, we get $\alpha(C_5^k) \leq 5^{k/2}$ and $\Theta(C_5) = \sqrt{5}$.

It turns out that this method extends to any graph G in place of C_5. All we have to do is to find a orthogonal representation that will give us the best bound. So, we can define the *Lovász number* of a graph G as:

$$\vartheta(G) = \min_{c,U} \max_{i \in V} \frac{1}{(c^\intercal u_i)^2},$$

where c is a unit vector in $\mathbb{R}^{|V(G)|}$ and U is an orthogonal representation of G.

Contrary to Lovász's first hope [30] $\vartheta(G)$ does not always equal $\Theta(G)$, but it is only an upper bound on it. However, these two are equal for some graphs, including all perfect graphs, as is demonstrated in the Lovász "Sandwich theorem".

Theorem 3.1.1 (Lovász "Sandwich theorem" [25])**.** *For any graph G:*

$$\omega(G) \leq \vartheta(\overline{G}) \leq \chi(G).$$

Because in perfect graphs $\omega(G) = \chi(G)$, we get $\omega(G) = \vartheta(\overline{G}) = \chi(G)$. Therefore, if for any perfect graph G, we could calculate $\vartheta(G)$ and $\vartheta(\overline{G})$, we would get $\omega(G)$, $\chi(G)$ and $\alpha(G)$.

But how can we construct an optimum (or even sufficiently good) orthogonal representation? It turns out that it can be computed in polynomial time using semidefinite optimization.

3.2 Computing ϑ

First, let us recall some definitions, with [45] as a reference for the linear algebra.

Definition 3.1 (eigenvector, eigenvalue)**.** *Let A be an $n \times n$ real matrix. An* eigenvector *of A is a vector x such that Ax is parallel to x. In other words, there is a real or a complex number λ, such that $Ax = \lambda x$. This λ is called the* eigenvalue *of A belonging to eigenvector x.*

If a matrix A is symmetric[1], all the eigenvalues are real.

Definition 3.2 (positive semidefinite matrix)**.** *Let A be an $n \times n$ symmetric matrix. A is* positive semidefinite *if and only if all of its eigenvalues are nonnegative. We denote it by $A \succeq 0$.*

We have equivalent definitions of semidefinite matrices.

Theorem 3.2.1 (Proposition 2.2 of [32])**.** *For a real symmetric $n \times n$ matrix A, the following are equivalent:*

(i) A is positive semidefinite,

(ii) for every $x \in \mathbb{R}^n$, $x^\intercal Ax$ is nonnegative,

(iii) for some matrix U, $A = U^\intercal U$,

(iv) A is a nonnegative linear combination of matrices of the type $xx^\intercal$.

From (ii) it follows that diagonal entries of any positive semidefinite matrix are nonnegative and the sum of two positive semidefinite matrices is positive semidefinite.

We may think equivalently of $n \times n$ matrices as vectors with n^2 coordinates.

Definition 3.3 (convex cone)**.** *A subset C of $\mathbb{R}^n$ is a* convex cone*, if and only if for any positive scalars α, β and for any $x, y \in C$, $\alpha x + \beta y \in C$.*

The fact that the sum of two positive semidefinite matrices is again positive semidefinite, with the fact that every positive scalar multiple of a positive semidefinite matrix is positive semidefinite, translates into the geometric statement that the set of all positive semidefinite matrices forms a convex closed cone $\mathcal{P}_n$ in $\mathbb{R}^{n \times n}$ with its apex at 0. This cone $\mathcal{P}_n$ is important but its structure is not trivial.

Semidefinite programs. Now, we can define a *semidefinite program* to be an optimization problem of the following form:

$$\begin{array}{ll} \text{minimize} & c^\intercal x \\ \text{subject to} & x_1 A_1 + \ldots + x_n A_n - B \succeq 0 \end{array}$$

Here $A_1, \ldots, A_n, B$ are given symmetric $m \times m$ matrices and $c \in \mathbb{R}^n$ is a given vector. Any choice of the values x_i that satisfies the given constraint is called a *feasible solution.*

[1] Matrix A is symmetric if and only if $A = A^\intercal$

The special case when $A_1, \dots A_n, B$ are diagonal matrices is a "generic" linear program, in fact we can think of semidefinite programs as generalizations of linear programs. Not all properties of linear programs are carried over to semidefinite programs, but the intuition is helpful.

Solving semidefinite programs is a complex topic, we refer to [24] for details. All we need to know is that we can solve semidefinite programs up to an arbitrarily small error in polynomial time. One of the methods to do this is called the *ellipsoid method*, hence the name for the coloring algorithm.

Calculating ϑ. Let us recall, that an orthogonal representation of a graph $G = (V, E)$ is a labeling $u : V \to \mathbb{R}^d$ for some d, such that $u_i^\intercal u_j = 0$ for all anticonnected vertices i, j. An *orthonormal* representation is an orthogonal representation with $|u_i| = 1$ for all i. The *angle* of an orthogonal representation is the smallest half-angle of a rotational cone containing the representing vectors.

Theorem 3.2.2 (Proposition 5.1 of [32])**.** *The minimum angle ϕ of any orthogonal representation of G is given by $\cos^2 \phi = 1/\vartheta(G)$.*

This leads us to definition of $\vartheta(G)$ in terms of semidefinite programming.

Theorem 3.2.3 (Proposition 5.3 of [32])**.** *$\vartheta(G)$ is the optimum of the following semidefinite program:*

$$\begin{array}{lll} \textit{minimize} & t & \\ \textit{subject to} & Y \succeq 0 & \\ & Y_{ij} = -1 & (\forall\ ij \in E(\overline{G})) \\ & Y_{ii} = t - 1 & \end{array}$$

It is also the optimum of the dual program

$$\begin{array}{lrcll} \textit{maximize} & \sum_{i \in V} \sum_{j \in V} Z_{ij} & & & \\ \textit{subject to} & Z & \succeq & 0 & \\ & Z_{ij} & = & 0 & (\forall\ ij \in E(G)) \\ & tr(Z) & = & 1 & \end{array}$$

Any stable set S of G provides a feasible solution of the dual program, by choosing $Z_{ij} = \frac{1}{S}$ if $i, j \in S$ and 0 otherwise. Similarly, any k-coloring of $\overline{G}$ provides a feasible solution of the former semidefinite program, by choosing $Y_{ij} = -1$ if i and j have different colors, $Y_{ii} = k - 1$, and $Y_{ij} = 0$ otherwise.

Now, that we know how to calculate $\vartheta(G)$, let us describe the algorithm to calculate the coloring of G.

3.3 Coloring perfect graphs using ellipsoid method

The following part is based on "Semidefinite Programming and Integer Programming" by Laurent and Rendl [27].

3.3.1 Maximum cardinality stable set

Given graph G, recall that the stability number of G is equal the clique number of the complement of G. This gives us a way to compute $\alpha(G)$ for any perfect graph G.

In fact, to calculate $\chi(\overline{G})$ and $\alpha(G)$ we only need an approximated value of $\vartheta(G)$ with precision smaller than $1/2$, as the former values are always integral.

We will now show how to find a stable set in G of size $\alpha(G)$. We will construct a sequence of induced subgraphs $G = G_0 \supseteq G_1 \supseteq \ldots \supseteq G_{|V|}$, so that $G_{|V|}$ is a required stable set.

Algorithm 3.3.1 (Maximum cardinality stable set in a perfect graph)
Input: A perfect graph $G = (V, E)$.
Output: A maximum cardinality stable set in G.

```
1: procedure GET-MAX-STABLE-SET(G)
2:     v_1, ..., v_n ← vertices of G
3:     G_0 ← G
4:     α_G = α(G)
5:     for each 1 ≤ i ≤ |V| do
6:         if α(G_{i-1} \ v_i) = α_G then
7:             G_i ← G_{i-1} \ {v_i}
8:         else
9:             G_i ← G_{i-1}
10:        end if
11:    end for
12:    return G_{|V|}
13: end procedure
```

Theorem 3.3.1. *Algorithm 3.3.1 works in polynomial time and for an input of G returns a maximal cardinality stable set of G.*

Proof. Let us prove that $G_{|V|}$ is indeed a stable set. Suppose otherwise and let v_iv_j be an edge in $G_{|V|}$ with $i < j$ and i minimal. But then $\alpha(G_{i-1} \setminus v_i) = \alpha(G_{i-1}) = \alpha(G)$ so by our construction v_i is not in G_i and v_iv_j is not an edge of G_n. Therefore there are no edges in $G_{|V|}$.

Because at every step we have $\alpha(G_i) = \alpha(G_{i-1})$, therefore $\alpha(G_{|V|}) = \alpha(G)$, so $G_{|V|}$ is required maximum cardinality stable set.

The running time of Algorithm 3.3.1 is polynomial, because we construct $|V|$ auxiliary graphs, each requiring calculating α once with additional $O(|V|^2)$ time for constructing the graph. □

Given a weight function $w : V \to \mathbb{N}$ we could calculate the maximum weighted stable set in G using the following modification:

Algorithm 3.3.2 (Maximum weighted stable set in a perfect graph)
Input: A perfect graph $G = (V, E)$ and a weight function $w : V \to \mathbb{N}$.
Output: A maximum weighted stable set in G.

```
1: procedure GET-MAX-WEIGHTED-STABLE-SET(G, w)
2:     for each v ∈ V(G) do
3:         W_v ← a set of nonadjacent vertices of size w(v)
4:     end for
5:     V' ← ⋃ W_v, for v ∈ V(G)                    \\ |V'| = Σ_{v∈V(G)} w(v)
6:     E' ← {xy for x, y ∈ V', such that for some u, v ∈ V, u ≠ v:
               x ∈ W_u, y ∈ W_v, and uv ∈ E(G)}
7:     G' ← (V', E')
8:     S' ← GET-MAX-STABLE-SET(G')
9:     S ← {v ∈ V(G), such that for some x ∈ W_v, x ∈ S'}
10:    return S
11: end procedure
```

We create graph G' by replacing every vertex v by a set W_v of $w(v)$ nonadjacent vertices, making two vertices $x \in W_v$, $y \in W_u$ adjacent in G' if and only if the vertices v, u are adjacent in G. Then we can calculate a maximum cardinality stable set in G' (we remark that G' is still perfect because every new introduced hole is even) and return a result of those vertices in G whose any (and therefore all) copies were chosen.

3.3.2 Stable set intersecting all maximum cardinality cliques

Next, let us show how to find a stable set intersecting all the maximum cardinality cliques of G. We will create a list $Q_1, \dots Q_t$ of all maximum cardinality cliques of G and return a stable set intersecting them all.

Recall that to calculate a maximum cardinality clique of G, we can calculate a maximum cardinality stable set of $\overline{G}$.

Algorithm 3.3.3 (Stable set intersecting all maximum cardinality cliques)
Input: A perfect graph $G = (V, E)$.
Output: A stable set which intersects all the maximum cardinality cliques of G.

```
1: procedure GET-STABLE-SET-INTERSECTING-MAX-CLIQUES(G)
2:     ω_G ← ω(G)
3:     Q_1 ← MAX-STABLE-SET(Ḡ)
4:     t ← 1
5:     while true do
6:         w ← a weight function V → ℕ, so that for v ∈ V, w(v) is equal to
             the number of cliques Q_1, ... Q_t that contain v.
7:         S ← MAX-WEIGHTED-STABLE-SET(G, w)
8:         if ω(G \ S) < ω_G then
9:             return S
10:        else
11:            t ← t + 1
12:            Q_t ← MAX-STABLE-SET(complement of (G \ S))
13:        end if
14:    end while
15: end procedure
```

To understand Algorithm 3.3.3, let us note that S calculated in line 7 has a weight equal t, which means that S meets each of $Q_1, \dots Q_t$. Then, if $\omega(G \setminus S) < \omega(G)$, then S meets all the maximum cardinality cliques in G so we return S.

There are at most $|V|$ maximum cardinality cliques in G. Adding a single clique to the list of maximum cardinality cliques requires constructing auxiliary graph for weighted maximum stable set, which is of size $O(|V|^2)$ and running Algorithm 3.3.2 on it. Therefore total running time is polynomial.

3.3.3 Minimum coloring

Algorithm 3.3.4 (Color perfect graph)
Input: A perfect graph $G = (V, E)$.
Output: C – a coloring of G using $\chi(G)$ colors.

```
1: procedure COLOR-PERFECT-GRAPH(G)
2:     if G = GET-MAX-STABLE-SET(G) then
3:         C ← coloring all vertices of G with one color
4:         return C
5:     else
6:         S ← GET-STABLE-SET-INTERSECTING-MAX-CLIQUES(G)
7:         C' ← COLOR-PERFECT-GRAPH(G \ S)
8:         C ← C' with vertices of S colored with an additional color
9:         return C
10:    end if
11: end procedure
```

Note that coloring C' uses $\chi(G \setminus S) = \omega(G \setminus S) = \omega(G) - 1$ colors, so we will call the recursion $O(|V|)$ times, each step of recursion requires polynomial time. Therefore the total running time of Algorithm 3.3.4 is polynomial.

3.4 Classical algorithms

Ever since Grötschel et al. published an ellipsoid-method-based polynomial algorithm for coloring perfect graphs, a combinatorial algorithm for doing the same has been sought. As of yet, it is not known, although there is much progress in the field.

Definition 3.4 (prism)**.** *A* prism *is a graph consisting of two disjoint triangles and two disjoint paths between them. Notice that for a graph to contain no odd holes, all three paths in a prism must have the same parity.*

A prism with all three paths odd is called an *odd prism* and a prism with all three paths even is called an *even prism*.

In 2005 Maffray and Trotignon showed a coloring algorithm that colors graphs containing no odd holes, no antiholes and no prisms (sometimes called Artemis graphs) in $O(|V|^4|E|)$ time [35]. They later improved the time com-

plexity to $O(|V|^2|E|)$ [28]. In 2015 Maffray showed an algorithm for coloring Berge graphs with no squares[2] (a square is a C_4) and no odd prisms [34].

In 2016 Chudnovsky et al. published an algorithm that given a perfect graph G with $\omega(G) = k$ colors it optimally in a time polynomial for a fixed k [12]. In general case this algorithm has a time complexity of $O(|V|^{(k+1)^2})$.

A most recent advancement (2018) is an algorithm by Chudnovsky et al. that colors any square-free Berge graphs in time of $O(|V|^9)$ [13]. Before proving strong perfect graph conjecture a similar conjecture for square-free Berge graphs has been proven by Conforti et al. [18]. During one of her lectures, Maria Chudnovsky expressed hope that discovery of full algorithm for coloring Berge graphs might follow a similar pattern.

The algorithm for coloring square-free Berge graphs is much more complex than CCLSV algorithm for Berge graphs recognition and would be much harder to implement. The main idea is that, if square-free Berge graph G is not a member of class that is simple to recognize and color (Artemis graphs), then it must have an induced prism. We find it and grow it until it is maximal and then we can create a partition of G, that we color recursively. The paper in which the algorithm is presented [13] is of much more mathematical "flavor" than [11] – there is a proof that a $O(|V|^9)$ algorithm exists, but actually deducing it from the paper is far from straightforward. We provide its pseudocode in the Appendix A.

[2]sometimes called square-free Berge graphs

Chapter 4

Implementation

A repository containing the discussed source code and tests is available under the link: `https://github.com/AdrianSiwiec/Perfect-Graphs`

4.1 Berge graphs recognition: naïve approach

When implementing an algorithm with the running time of $O(|V|^9)$ (especially one as complex as CCLSV), one of the first questions that pop up is whether this algorithm is at all usable. We couldn't find any existing programs that test whether the graph is perfect, be it an implementation of CCLSV, or a naïve approach. So, to test the usability of CCLSV implementation, we measured it against our own naïve perfect recognition algorithm. Let us describe it briefly.

We try find an odd hole directly, then, we run the same algorithm on $\overline{G}$, and if no odd hole is found we report that the graph is perfect, else that it is not.

To find an odd hole, we use a backtracking technique. We will enumerate all paths and holes, and for each test if it is an odd hole. A partial solution to our backtracking problem is a path in G. To move forward from a partial solution, we append consecutively to its end all neighbors of current path's last vertex and continue recursively after each. When a current solution is an odd hole we return it. When we exhaust all possible paths from the current partial solution, we remove its last vertex and continue.

This backtracking algorithm is very simple, but has a great potential for optimizations in its implementation. We also use path enumeration algorithm in the CCLSV algorithm, so we give much attention to its optimization.

Path enumerating optimizations

Now let's turn our attention to enumerating all paths. This is a generalization of the problem of finding an odd hole, if we allow a path to have a single additional edge between the first and the last of its vertices if and only if the hole formed would be odd.

First of all, there are many methods for generating all possible paths. We could simply enumerate all sequences of vertices without repetition and for each one of them check if it is a path, i.e., if all pairs of vertices next to each other are connected and that there are no other edges, but this would be too slow and render our algorithm unusable.

We will create a method that enumerates all paths in some order. We notice that returning a new path for each call of the enumeration method is wasteful. Therefore, we need a method that receives a reference to a path on the input (or a special flag indicating it should generate first path) and returns next path in some order (or a first path, or a code signaling all paths have been generated) using memory from the input. As this method will be used many, many times we require of it to work in place with constant additional space.

With those requirements defined a simplest algorithm would be to implement a sort of a "counter" with base of $|V(G)|$. In short: we take a path on a input, increment its last vertex until it is a neighbor of the vertex one before last. Then we check if generated sequence is a path – all of its vertices are unique and there are edges only consecutive vertices share an edge. If it is we return it and if we run out of vertices before a path is found, we increment one before last vertex until it is a neighbor of one vertex before it, set last vertex to first and continue the process. If still no path is found, we increment vertices closer to beginning of the path, until all a path is found or we check all candidates.

But we can do much better with some care and a suitable data structure. In addition to having for each vertex a list of its neighbors we create data structure that will allow us to generate a candidate for next path in amortized $O(1)$ time.

We have an array $first$ in which for each vertex is written its first neighbor on its neighbors list, and an array $next$ of size $|V(G)|^2$ where for each pair of connected vertices a, b, there is written a neighbor of a that is next after b in a neighbors list of a (or a flag indicating b is a's last neighbor). Then, say our input is a path $v = v_1, \dots v_k$. If $next[v_{k-1}][v_k]$ exists we change v_k to $next[v_{k-1}][v_k]$ and return v. If it indicated that v_k is v_{k-1}'s last neighbor, we set v_{k-1} to $next[v_{k-2}][v_{k-1}]$ and then v_k to $first[v_{k-1}]$ (or we go further back, if all neighbors of v_{k-2} are done).

This simple change in the data structure design gave us a speedup of overall running time of our naïve algorithm in the range of about 5x.

Another crucial optimization is to eliminate partial solutions that have edges between not consecutive vertices right away. Also, when we look for an odd hole specifically, we can assume that its first vertex we enumerate is the lowest. Those two optimizations make a huge difference. With them, our naïve algorithm implementation went from being able to check graphs with $|V|$ up to 12-14, within a reasonable time, to being faster than CCLSV on most of our tests.

Array unique. For each path candidate v we call a subroutine `areUnique`(v) to determine whether all vertices in it are unique. As this subroutine could be called many times for generating a single path, its optimization is very important.

A theoretically optimal solution in general would be to have a hashing set of vertices. For each vertex in a path candidate, we check if it is already in the set. If it is, return that not all vertices are unique, else add it to the set.

In our use case, a few optimizations can be made. First, we don't need a hashing set. We can have an array of bools, of size $|V(G)|$ and mark vertices of the path there. Paths are usually much shorter than $|V(G)|$, but we operate on small graphs, so we can afford this. Second, we notice that we don't need to create this array for each call of `areUnique` procedure. Let's instead have a static array *stamp* of ints and a static *counter* of how many times we called `areUnique` method. For each element of a path v_i, if the value of *stamp* array is equal to *counter* we report that vertices are not unique. Else we set $stamp[v_i] = counter$. When returning from `areUnique` method we increment *counter*.

This optimization alone almost cuts down naïve algorithm's running time in half.

Other uses for path generation. Because of a good performance of our optimized algorithm for generating paths, we use it whenever possible. It can be easily modified to enumerate all "paths" that can have additional edges or to enumerate holes and therefore has much use in the CCLSV algorithm. For example, when searching for jewels we generate with it all "paths" with possible additional edges, of length 5 and for each check if it has required properties of a jewel. This proved to be much faster than generating vertices and checking their properties one by one. We use the same algorithm for generating starting vertices for a possible $\mathcal{T}_2$ and $\mathcal{T}_3$. Before optimizing path enumeration algorithm, all calls of *areUnique* alone took about 70% of total running time of the CCLSV algorithm. After optimizing *areUnique*, path enumeration still took more than 50% of total running time. After all optimizations it is almost unnoticeable.

With all those optimizations made, our naïve algorithm proved to be quite fast in some cases, although its running time is very dependent on the properties of the input, as we should expect from an optimized non-polynomial algorithm. See Section 4.4 for the running times.

4.2 Berge graphs recognition: polynomial algorithm

The Berge recognition algorithm's running time is $O(|V|^9)$, which brings into question its applicability to any real use case. Although time complexity is indeed a limiting factor, a number of lower level optimizations done on implementation's level make a very big difference and make it more viable than naïve algorithm, at least on some test cases. We also explore a new frontier of implementing its most time consuming part on massively parallel GPU architecture, with some good results (Section 4.3).

4.2.1 Optimizations

When implementing a complicated algorithm that has a time complexity of $O(|V|^9)$ optimizations can be both crucial and difficult to implement. There is not a single code path that takes up all the running time – or at least there isn't one from a theoretical point of view. Therefore a tool for inspecting running time bottlenecks is needed. We used Valgrind's tool called *callgrind* [2].

Callgrind is a profiling tool that records the call history and event counts (data reads, cache misses etc.) of a program and presents it as a call-graph. It is then possible to visualize this, we used a tool *gprof2dot* [22] to generate visual call graphs from callgrind's output.

Before any optimizations, a major part of Berge recognition algorithm was being spent on enumerating all possible paths of given length – this is done either to find a simple forbidden structure or to check a possible near cleaner for an amenable odd hole. Therefore optimizations of the path enumerating algorithm were also helpful in speeding up the CCLSV implementation.

Another algorithm for which we found major speedups is the algorithm to generate all possible near-cleaners (Algorithm 2.2.3). We optimized it by using a a `dynamic_bitset` from the boost library [8]. It is a data structure to represent a set of bits, that also allows for fast bitwise operators that one can apply to builtin integers. At the very end of Algorithm 2.2.3 a set $\mathcal{R}$ is constructed, which is a set of all possible unions of pairs X_i and N_j. We used `dynamic_bitset` to represent each of the sets X_i and N_j, and the elements of the set $\mathcal{R}$. This significantly speeds up the calculation of $\mathcal{R}$ (line 30). With this modification, the speedup of Algorithm 2.2.3 was about 20%.

After these optimizations, checking each potential near-cleaner by Algorithm 2.2.5 is by far the biggest bottleneck of the CCLSV algorithm. We didn't find any major optimizations there, but the Algorithm 2.2.5 has a good potential for parallelization, which we explore in Section 4.3.

4.2.2 Correctness testing

Unit tests

In an algorithm this complex, debugging can be difficult and time consuming, so we used extensive unit testing and principles of test driven development to make sure that the program results are correct.

Whole algorithm was divided into subroutines used in many places. One of such subroutines is a path generation algorithm described above (Section 4.1). There are many other generalized methods we implemented: checking if a set of vertices is X-complete, getting a set of components of a graph, creating a induced graph or finding a shortest path, such that each internal vertex satisfies a predicate. Each of those and many more methods have unit tests that check their general use and edge cases. This allowed us to debug very effectively and have complex algorithms be simple to analyze.

In addition to correctness checking, extensive unit test suite allowed us to optimize our algorithm and test different subroutines with ease, without the

fear of introducing bugs.

End to end tests

In addition to unit tests, we employ a range of end to end tests, that check the final answer of the algorithm. We compare answer to the naïve algorithm's answer and also to the result of the algorithm on CUDA (Section 4.3).

Also, we test our implementations on graphs that we know are perfect – such as bipartite graphs, line graphs of bipartite graphs and complements thereof. In addition to that, a test on all perfect graphs up to a size of 10 was performed – we used [39] as a source of perfect graphs.

We also test on graphs that we know are not perfect: we generate them by generating an odd hole and joining it with some edges to a random graph.

In total over 200 CPU-hours of end to end tests were performed without any errors.

4.3 Parallelism with CUDA

CUDA is a parallel computing platform model created by Nvidia and executed on Nvidia GPUs. It allows to run a program by many CUDA threads at once. Although usually described by the the PRAM models, implementing PRAM algorithms in CUDA architecture is not as straightforward as implementing RAM algorithms on a CPU. Some of the causes are: synchronization between threads is very costly, there are multiple memory models to choose from, threads run divided in blocks and it is possible to efficiently share memory within blocks and copying memory from and back to CPU suffers from a large latency. What is more, there are no standard dynamic structures in the PRAM model, let alone CUDA, that are taken for granted in the RAM model, such as hashing maps or sets.

We posed a question if it would be profitable to use GPU for the problem of recognizing Berge graphs. On one hand the graphs we are working on are small and our time complexity so large that a speedup from massively parallel architecture could be profitable. On the other hand, the algorithm is complex and not easy to parallelize. Therefore we decided to analyze the CCLSV algorithm and parallelize parts that would benefit from it the most.

We identified testing all possible near-cleaners (Algorithm 2.2.5) as the biggest bottleneck for larger graphs. We considered two approaches: run whole Algorithm 2.2.5 on a single CUDA core, testing multiple Xs in parallel, or parallelize Algorithm 2.2.5 itself and run it on one X at a time. It turned out that second approach is better and much simpler.

Let us recall the Algorithm 2.2.5. It calculates array R of shortest paths, then for each 3-vertex path and an additional vertex it does $O(1)$ checks to see if they along with two paths from R give us an odd hole. We will parallelize the work after calculating R, that is, lines 10-19. Let us notice, that for all Xs, all 3-vertex paths enumerated in line 11 are the same. We calculate them

beforehand, then each CUDA thread receives one 3-vertex path x_1-x_2-x_3 and an additional vertex y_1 and performs the required checks in lines 12-17. This is almost perfect scenario for the GPU – we do a simple SIMD[1] work in parallel, without a lot scattered memory access.

The part of the CCLSV implemented on the GPU achieved speedup of up to 15x. This gave us a speedup of up to 9x for overall CCLSV algorithm, depending on a class of graph we benchmark on.

4.4 Experiments

Data sets

Let us describe experiments and their results. All of our algorithms search for an odd hole or an odd antihole and stop when find one, or a evidence of one. Therefore, their running times are greatest with perfect graphs on the input and we decided to use perfect graphs for our performance benchmarks. For every test, the vertex numbers were shuffled. For every randomly generated graph class and for each size, we ran the tests on 10 graphs generated with different seeds and took their average running time.

Below, we first present our data sets then overall time results on them and lastly a plot breaking down what parts of CCLSV and GPU CCLSV implementations constitute their overall running times.

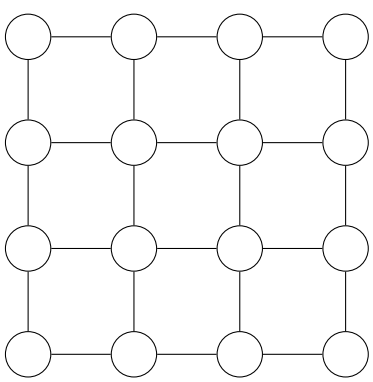
We ran tests on eight classes of perfect graphs:

- random bipartite graphs,
- line graphs of random bipartite graphs,
- lattice graphs (fig. 4.1a),
- rook graphs (fig. 4.1b),
- knight graphs (fig. 4.1c),
- hypercube graphs (fig. 4.1d),
- split graphs (fig. 4.1e),
- full binary trees.

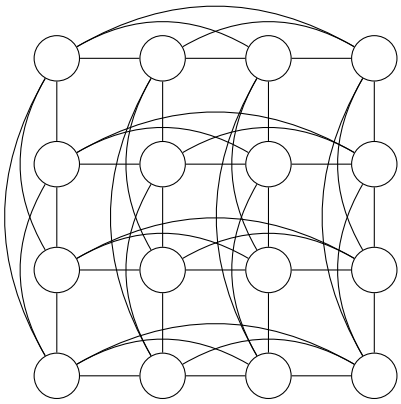
In Figure 4.1 we present small examples of some of the above classes. Let us describe them briefly.

We generate random bipartite graphs by creating two equal-sized sets of vertices, and then adding every possible edge between them with a probability of $p = 0.5$. In this way we can also generate line graphs of random bipartite graphs. This gives us graphs of varying $|V|$, so we repeated the process until we had sufficient number of graphs for each $|V|$.

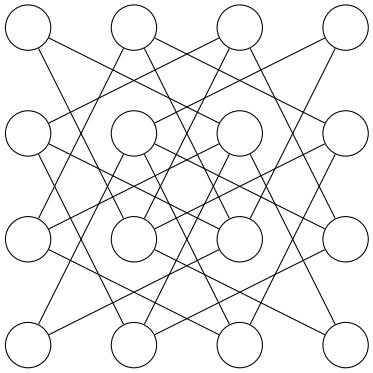
[1]Single Instruction Multiple Data, meaning performing same operations on multiple data points in parallel



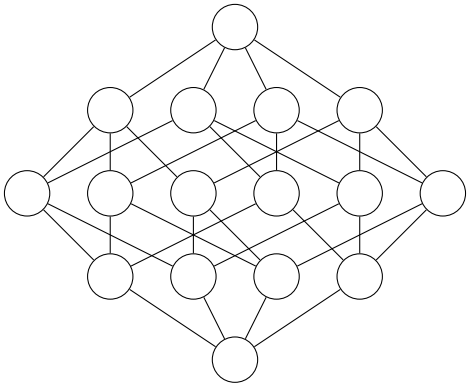
(a) Lattice graph



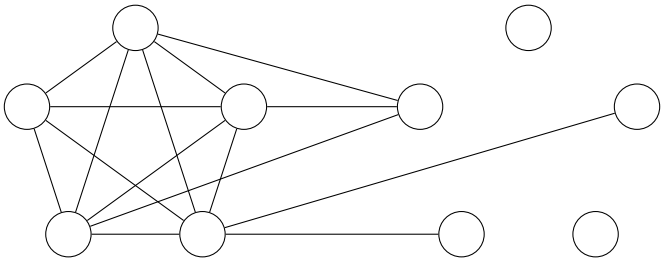
(b) Rook graph



(c) Knight's graph



(d) Hypercube, $n = 16$



(e) Example split graph, $n = 10$

Figure 4.1: Graph classes

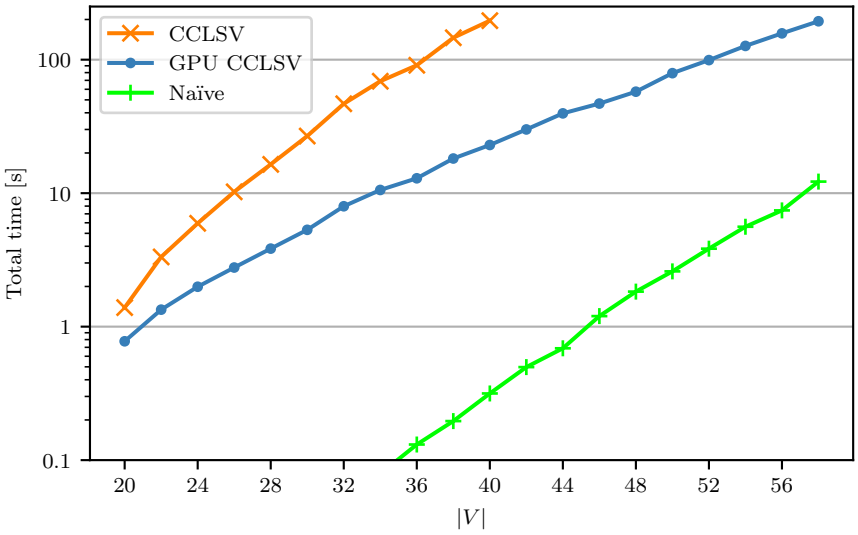


Figure 4.2: Random bipartite graphs

There are also a few interesting classes of perfect graphs that appear on a grid, or a chessboard. In lattice graph, each vertex is connected to vertices that are above, below, to the left and to the right of it, if such vertices exist. See Figure 4.1a for an example of a 4×4 lattice graph. If we imagine our grid to be a chessboard we can define two vertices a rook graph to share an edge, when the rook from chess can move from one vertex to the other in one move. See Figure 4.1b for an example of 4×4 rook graph. In similar manner we can define a knight graph (Figure 4.1c).

A hypercube is a generalization of a cube to a higher dimensional space. We use hypercubes that are not complete, to achieve higher granularity of data. In a hypercube of n vertices there is an edge between vertices u and v if and only if binary representations of u and v differ by exactly one bit.

Split graphs we use are unions of cliques and independent sets of same size, with some edges between them. In generating them, each edge between a vertex from a clique and a vertex from an independent set has had a chance $p = 0.5$ to appear.

Finally, we benchmark on full binary trees to see how CCLSV works with a tree as an input.

Time results

We ran our tests on Intel Xeon E5-2690 CPU and Nvidia Tesla K80 GPU with 4992 CUDA cores.

First, let us take a look at the performance of the CCLSV algorithm. If

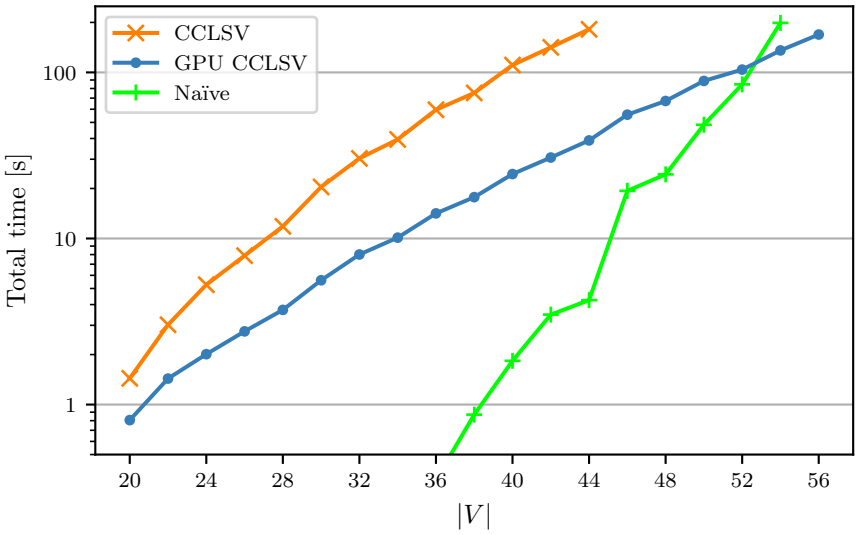


Figure 4.3: Line graphs of random bipartite graphs

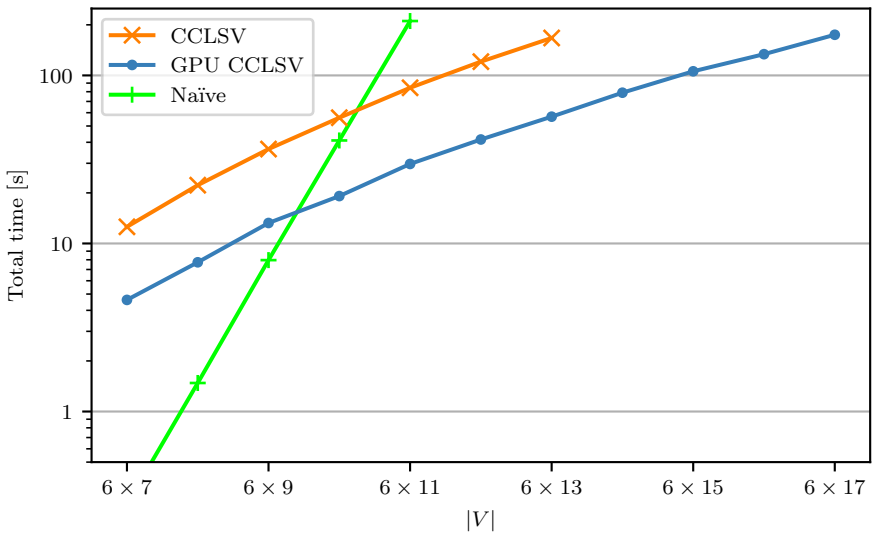


Figure 4.4: Lattice graphs

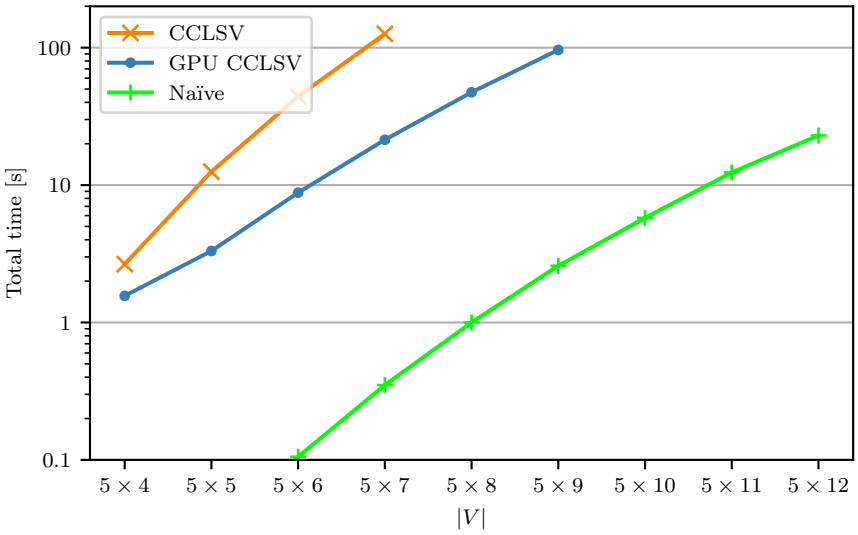


Figure 4.5: Rook graphs

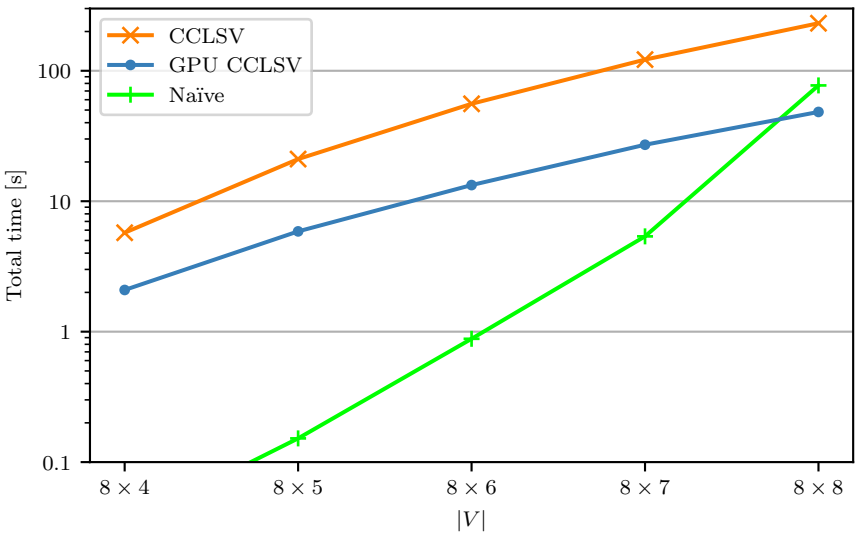


Figure 4.6: Knight graphs

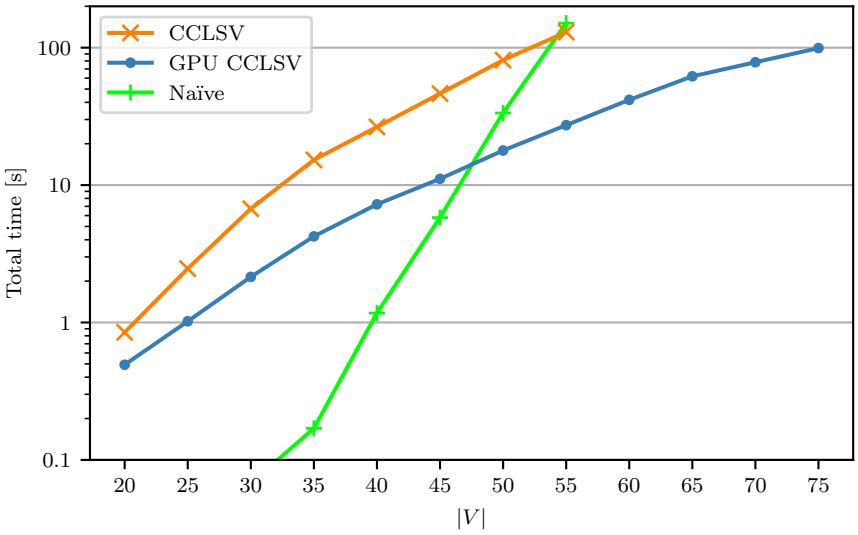


Figure 4.7: Hypercube graphs

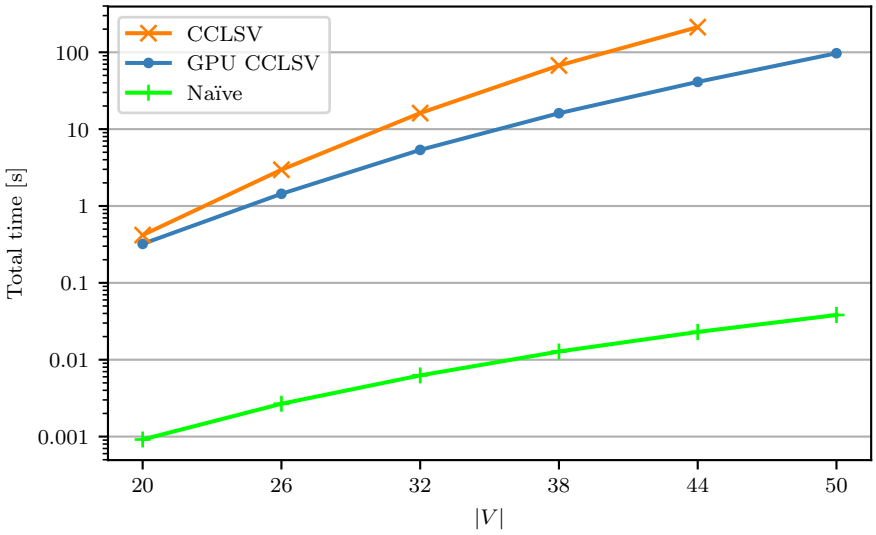


Figure 4.8: Split graphs

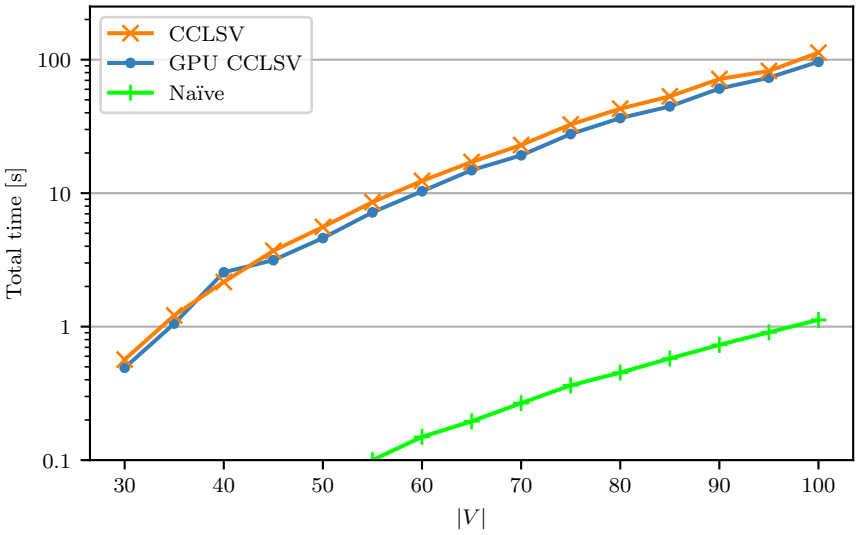


Figure 4.9: Full binary trees

we limit the runtime to about two minutes, we can run it on graphs of sizes up to 40-70, depending on the type of the graph we test. Worth noting is the fact that the CCLSV algorithm usually grows slower than the naïve algorithm. Oftentimes this is mitigated by the fact that the naïve algorithm works very well on smaller graphs, but it is not always the case. For example, for larger instances of lattice graphs and hypercube graphs the CCLSV works better than naïve algorithm. If we would want to test graphs larger than presented here, the GPU CCLSV seems to be the best solution. For every type of graph we tested, it finished within two minutes for graphs of $|V|$ up to about 50. The naïve algorithm is very fast for some graphs, like split graphs or random bipartite graphs, but GPU CCLSV still seems to be the best solution for larger graphs of unknown structure.

Let us try to understand where the speedup of GPU CCLSV comes from. Figure 4.10 shows breakdown of each implementation running times on a single graph of each class. We can clearly see that in most cases testing possible near-cleaners takes most of the running time, so the speedup achieved by utilizing the GPU is significant.

Examples of full binary graphs, and lattice graphs show that getting possible near cleaners still can be a big bottleneck. Parallelization of this method could be considered in the future. A possible idea would be to use recently developed dynamic dictionary for the GPU [4]. When running GPU CCLSV on split graphs, quite a lot of time is spent checking simple structures. The biggest challenge we can see is in an attempt to parallelize this is spreading work evenly across CUDA threads.

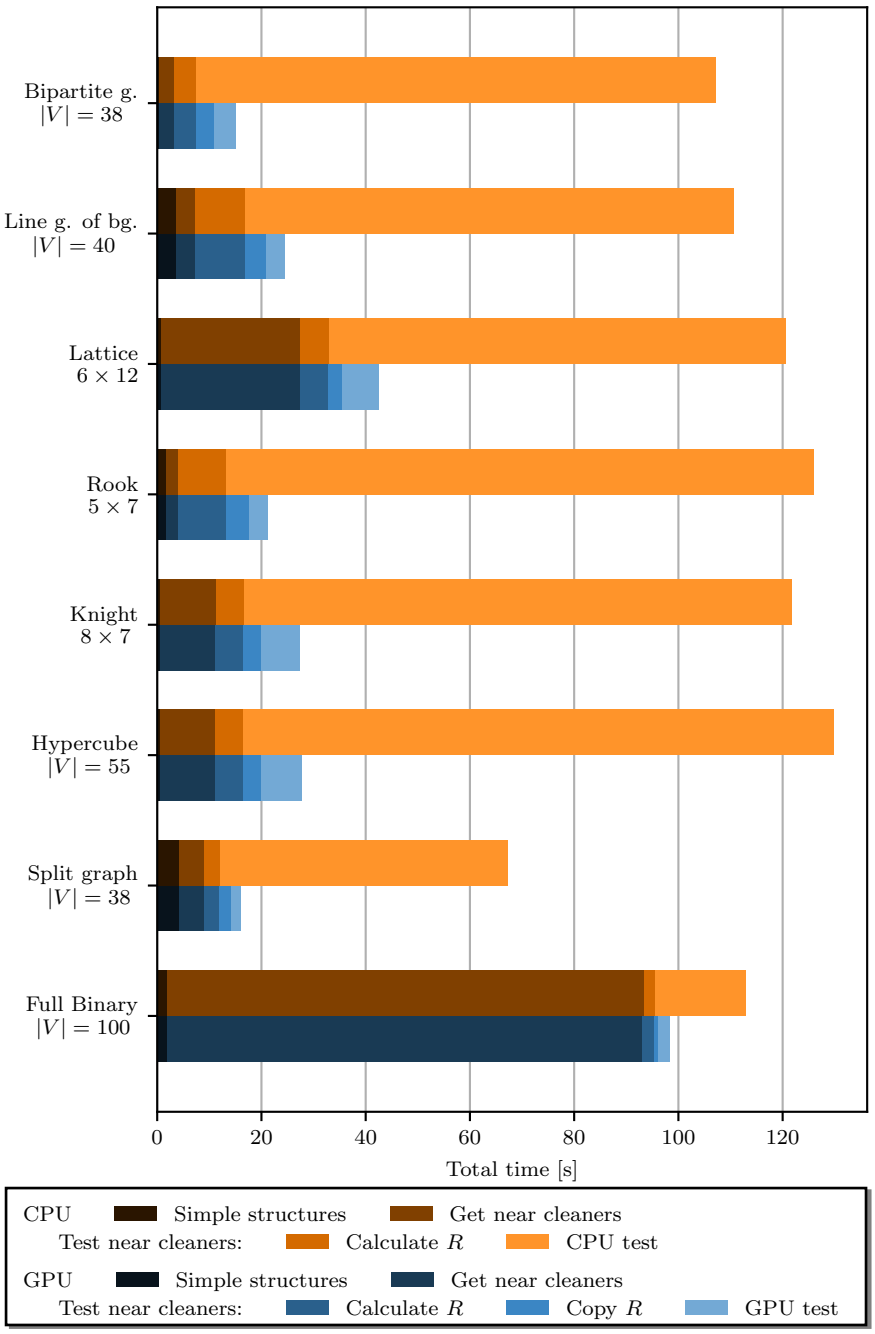


Figure 4.10: Detailed running times

4.5 Coloring Berge graphs

Ellipsoid method

We used an open source CSDP [9, 10] library, that implements predictor corrector variant of the semidefinite programming algorithm to calculate $\vartheta(G)$, given G on the input. The CSDP library has been used in many recent publications across different fields, such as [3, 1].

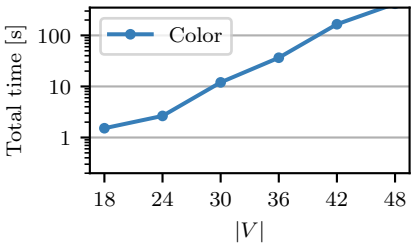
Calculating $\vartheta(G)$ is the most complicated part of the coloring algorithm. With that done by an external library, the rest of the program is a straightforward implementation of the algorithms in Section 3.3. In most of our tests, the majority of running time was consumed on calculating ϑ of various graphs. As it was done by an external library, there isn't much optimization potential for us. One thing of note is, that by specifying the precision of ϑ we want to calculate to be 1/3 the algorithm sped up by 20-40%. Also, we cache results of calculating ϑ, meaning that through whole execution of our coloring algorithm, we only calculate ϑ for each graph once. Our main goal of the implementation was to check if this method is still impractical, even on modern equipment. Let us proceed straight to experiments and results.

Experiments and results

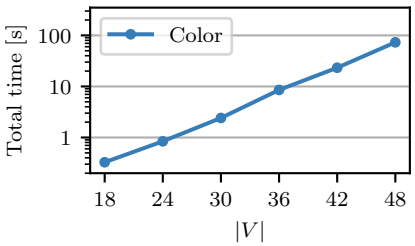
We ran our tests on the data sets generated in the same way as for perfect graph recognition. We only adjusted graph sizes to get the running times of up to a couple of minutes.

First, we present overall running times of the coloring algorithm (Figure 4.11) and then breakdown the running time in two parts: calculating ϑ by the CSDP library and graphs manipulations which constitute the rest of the coloring algorithm (Figure 4.12). We note that ϑ calculation is very dependent on the type of the graph. For our biggest tested cases of $|V| = 48$, all required calculations of ϑ take almost 7 minutes in bipartite graphs, but only 12 seconds in line graphs of bipartite graphs. Because CSDP library is highly optimized, it is expected that it would run much faster on some graphs. However, it is not clear what we could do to speedup the running times on tests that CSDP is slow on, such as bipartite graphs or lattice graphs. A potential improvement would be to again use CUDA, this time for ϑ calculations, possibly utilizing the cuSOLVER library [42].

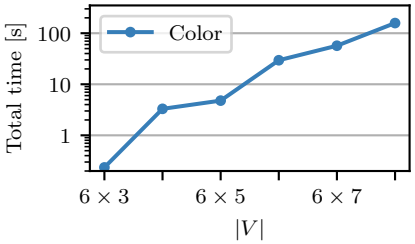
We notice that on some graphs, namely line graphs of bipartite graphs, rook graphs and split graphs the time taken by operations other than calculating ϑ is quite significant. However, the overall running times on these graphs are low, so we didn't particularly optimize this, so probably some improvements could be made here.



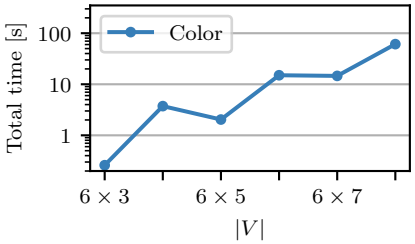
(a) Random bipartite graphs



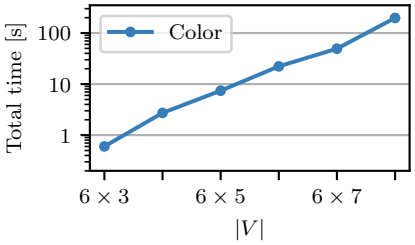
(b) Line graphs of random bipartite graphs



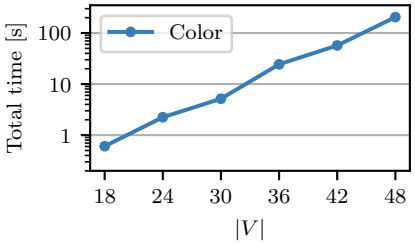
(c) Lattice graphs



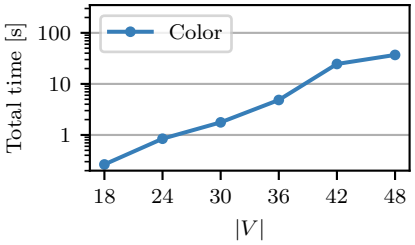
(d) Rook graphs



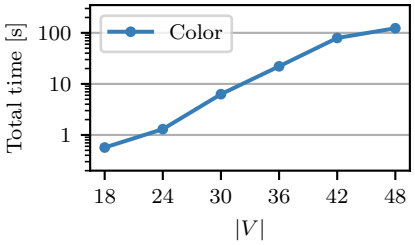
(e) Knight graphs



(f) Hypercube graphs



(g) Split graphs



(h) Full binary trees

Figure 4.11: Overall times of CSDP Color

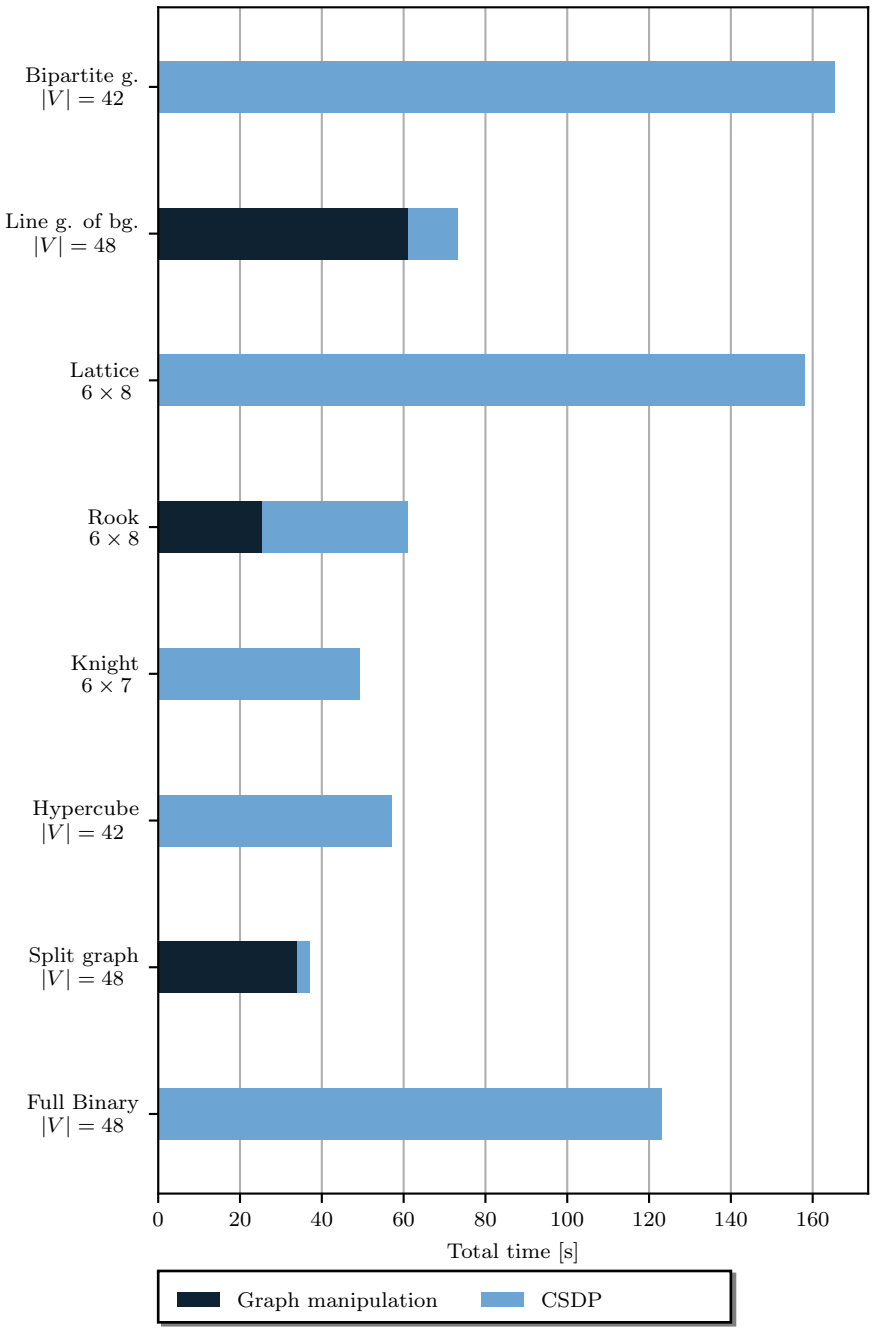


Figure 4.12: Detailed times of coloring

4.6 Conclusions

We implemented and benchmarked two different approaches for recognizing perfect graphs. Both have their positives and drawbacks – testing big graph of unknown structure is probably better done with the CCLSV algorithm, while the naïve algorithm is much faster on some cases. The GPU CCLSV implementation shows that using parallel computation can be very profitable for this task. An additional research could be done to see if further progress can be made, however there is no obvious improvement we can see. Generating all possible near-cleaners makes a heavy use of sets (our tests indicated that if we simply list all combinations of sets X_i and N_j, the resulting list $\mathcal{R}$ would be much bigger than necessary). Possibly a newly developed dynamic dictionary for the GPU [4] would be of use here. Testing for simple forbidden structures should also be parallelizable, although our preliminary tests indicated that spreading all the work evenly between CUDA threads is not trivial.

A GPU implementation of the naïve algorithm could also be considered for future work. We have created one, but after all path-enumeration optimizations of the CPU implementation we didn't manage to achieve any noticeable speedup. The reason again was, that we couldn't spread the work evenly between CUDA threads. Because some prefixes of paths require much more processing than others, this is not a trivial task and a careful study would be needed.

We also implemented algorithm for coloring perfect graphs. It is not entirely unusable, although far from fast. Maybe a better way of calculating ϑ could be found, or the GPU could again be usable for this task. We point to Nvidia's cuSOLVER [42] as a possible starting point.

Appendices

Appendix A

Coloring Square-free Berge graphs

The following appendix is based on the paper "Coloring square-free Berge graphs" by Chudnovsky et al. [13].

A.1 Introduction

We give a brief overview of the square-free perfect graph coloring algorithm and a pseudocode thereof. The work is not intended to be self-contained and will be impossible to fully understand without [13] at hand. But when reading [13], one may pose a question of what sort of an algorithm emerges from it, and we believe the pseudocode below gives a first overview and is a major step required before an attempt to implement it would be made. For some, looking at the pseudocode could provide a better way to analyze and possibly simplify the algorithm.

What we present would not be trivial to implement. For example in many places we say "choose x, so that ...", without specifying how to do it. A most glaring example is calculating an inverse of a line graph (in Algorithm A.3.9) which is a complicated algorithm in and of itself, see [29] for reference.

Recall, that a prism (definition 3.4) is a graph consisting of two disjoint triangles and two disjoint paths between them.

Definition A.1 (Artemis graph)**.** *A graph G is an* Artemis graph *if and only if it contains no odd hole, no antihole of length at least five, and no prism.*

In 2003 Maffray et al. published a paper on Artemis graphs where they show polynomial algorithms to recognize and color them [35]. In 2005 Maffray et al. published a paper on Artemis graphs [36] they showed an polynomial algorithm that checks if a perfect graph contains a prism, and returns it if so.

Theorem A.1.1. *[36] There is a polynomial algorithm, that given a perfect graph G, returns an induced prism K of G, or an answer that G contains no prisms.*

In 2009 Maffray et al. published a paper on coloring Artemis graphs in time of $O(|V|^2|E|)$ [28]. The paper contains simple pseudocode of the algorithm.

Theorem A.1.2. *[28] There is a polynomial algorithm, that given an Artemis graph G, returns a $\omega(G)$-coloring of G.*

So, we will focus on square-free perfect graphs that are not Artemis graphs. Let us notice, that every antihole of length at least 6 contains a square, therefore a square-free perfect graph that is not an Artemis graph must contain a prism. We will use this fact extensively.

Let us state a few definitions.

Definition A.2 (subdivision)**.** *In a graph G, subdividing an edge $uv \in E(G)$ means removing the edge uv and adding a new vertex w and two new edges uw, wv. Starting with a graph G, the effect of repeatedly subdividing edges prioduces a graph H called a* subdivision of G. *Note that $V(G) \subseteq V(H)$.*

Definition A.3 (triad)**.** *In a graph G, a* triad *is a set of three pairwise non-adjacent vertices.*

Definition A.4 (good partition)**.** *A* good partition *of a graph G is a partition (K_1, K_2, K_3, L, R) of $V(G)$ such that:*

- *L and R are not empty, and L is anticomplete to R,*
- *$K_1 \cup K_2$ and $K_2 \cup K_3$ are cliques,*
- *in the graph obtained from G by removing all edges between K_1 and K_3, every path with one end in K_1, the other in K_3, and interior in L contains a vertex from L that is complete to K_1,*
- *either K_1 is anticomplete to K_3, or no vertex in L has neighbors in both K_1 and K_3,*
- *for some $x \in L$ and $y \in R$, there is a triad of G that contains $\{x, y\}$.*

The algorithm we present is derived from contructive proof of the following two theorems.

Theorem A.1.3. *[Theorem 2.1 of [13]] Let G be a square-free Berge graph. If G contains a prism, then G has a good partition.*

Theorem A.1.4. *[Lemma 2.2 of [13]] Let G be a square-free Berge graph. Suppose that $V(G)$ has a good partition (K_1, K_2, K_3, L, R). Let $G_1 = G \setminus R$, $G_2 = G \setminus L$ and for $i = 1, 2$ let c_i be an $\omega(G_i)$-coloring of G_i. Then an $\omega(G)$-coloring of G can be obtained in polynomial time.*

A few more structures are used throughout the algorithm. Let us define them.

For definitions A.5 and A.6, let K be a prism with triangles $\{a_1, a_2, a_3\}$ and $\{b_1, b_2, b_3\}$ and with paths R_1, R_2, R_3, where each R_i has ends a_i and b_1.

Definition A.5 (major neighbor of a prism)**.** *A vertex $v \in V(G) \setminus K$ is a* major neighbor *of pritm K is and only if it has at least two neighbors in $\{a_1, a_2, a_3\}$ and at least two neighbors in $\{b_1, b_2, b_3\}$.*

Definition A.6 (local subset of a prism)**.** *A subset $X \subseteq V(K)$ is* local *if and only if either $X \subseteq \{a_1, a_2, a_3\}$, or $X \subseteq \{b_1, b_2, b_3\}$, or $X \subseteq R_i$ for some $i \in \{1, 2, 3\}$.*

Definition A.7 (attachment)**.** *Let F, K be an induced subgraphs of a graph G, with $V(F) \cap V(K) = \emptyset$. Any vertex $k \in V(K)$ that has a neighbor in $V(F)$ in a graph G is called an* attachment *of F in K. Whenever any vertex $k \in V(K)$ has an attachment of F we say that F* attaches *to K.*

Definition A.8 (hyperprism)**.** *A* hyperprism *is a graph H, whose vertices can be partitioned into nine sets:*

$$\begin{array}{ccc} A_1 & C_1 & B_1 \\ A_2 & C_2 & B_2 \\ A_3 & C_3 & B_3 \end{array}$$

with the following properties:

- *each of $A_1, A_2, A_3, B_1, B_2, B_3$ is nonempty,*
- *for distinct $i, j \in \{1, 2, 3\}$, A_i is complete to A_j, and B_i is complete to B_j, and there are no other edges between $A_i \cup B_i \cup C_i$ and $A_j \cup B_j \cup C_j$,*
- *for each $i \in \{1, 2, 3\}$, every vertex of $A_i \cup B_i \cup C_i$ belongs to a path between A_i and B_i with interior in C_i.*

Whenever we will speak about hyperprisms, we will denote its subsets as in the above definition, unless stated otherwise.

For a hyperprism H we have a few more definitions:

Definition A.9 (i-rung of a hyperprism)**.** *For each $i \in \{1, 2, 3\}$, any path from A_i to B_i with interior in C_i is called an i-*rung*.*

Definition A.10 (strip of a hyperprism)**.** *For each $i \in \{1, 2, 3\}$, the triple (A_i, C_i, B_i) is called a* strip *of the hyperprism.*

Definition A.11 (instance of a hyperprism)**.** *For each $i \in \{1, 2, 3\}$, let us pick any i-rung R_i. Then R_1, R_2, R_3 form a prism. Any such prism is called an* instance *of a hyperprism.*

From the definition of an instance of a hyperprism, we can see that any prism is also a hyperprism.

Let us note, that if a hyperprism H contains no odd hole, all rungs have the same parity. We then call the hyperprism odd or even accordingly.

Given a graph G that contains a hyperprism H, we can define a few more structures.

Definition A.12 (major neighbor of a hyperprism)**.** *A vertex $v \in V(G) \setminus V(H)$ is a* major neighbor *of H, if and only if it is a major neighbor of some instance of H.*

Definition A.13 (local subset of a hyperprism)**.** *A subset $X \subseteq V(H)$ is a local subset of a hyperprism H if and only if either $X \subseteq A_i \cup A_2 \cup A_3$ or $X \subseteq B_1 \cup B_2 \cup B_3$ or $X \subseteq A_i \cup B_i \cup C_i$ for some $i \in \{1, 2, 3\}$.*

Definition A.14 (maximal hyperprism)**.** *A hyperprism H is* maximal *if and only if there is no hyperprism H' in G, such that $V(H) \subsetneq V(H')$.*

Algorithm A.3.3 for growing hyperprism is based upon the following theorem. Recall that by K_4 we denote a clique on 4 vertices.

Theorem A.1.5 (Lemma 3.3 of [13])**.** *Let G be a berge graph, let H be a hyperprism in G and let M be the set of major neighbors of H in G. Let F be a component of $V(G) \setminus (V(H) \cup M)$, such that the set of attachments of F in H is not local. Then one can find in polynomial time one of the following*

- *a path P, with $\emptyset \neq V(P) \subseteq V(F)$, such that $V(H) \cup V(P)$ induces a hyperprism,*
- *a path P, with $\emptyset \neq V(P) \subseteq V(F)$, and for each $i \in \{1, 2, 3\}$ an i-rung R_i of H, such that $V(P) \cup V(R_1) \cup V(R_2) \cup V(R_3)$ induces the line graph of a bipartite subdivision of K_4.*

If at any time during growing a hyperprism we encounter the latter outcome of the Theorem A.1.5 we stop it and instead focus on the newly found line graph of a bipartite subdivision of K_4. Algorithms A.3.6, A.3.7, A.3.8 and A.3.9 take their roots in the following theorem.

Theorem A.1.6 (Theorem 6.1 of [13])**.** *Let G be a square-free Berge graph, and assume that G contains the line graph of a bipartite subdivision of K_4. Then G admits a good partition.*

Before providing the pseudocode, we need a few more definitions.

Definition A.15 (branch)**.** *Given a graph G, a* branch *is a path whose interior vertices have degree 2 and whose ends have degree at leas 3. A* branch-vertex *is any vertex of degree at least 3.*

Definition A.16 (appearance of a graph)**.** *Given a graph G, and* appearance *of a graph J is any induced subgraph of G that is isomorphic to the line graph $L(H)$ of a bipartite subdivision H of J. An appearance of J is* degenerate *if and only if either $J = H = K_{3,3}$[1], or $J = K_4$ and the four vertices of J form a square in H.*

Note that a degenerate appearance of a graph contains a square.

Definition A.17 (overshadowed appearance)**.** *An appearance of $L(H)$ of J in G is* overshadowed *if and only if there is a branch B of H, of length at least 3, with ends b_1, b_2, such that some vertex of G is non-adjacent in G to at most one vertex in $\{b_1x \in V(L(H))$, for $x \in V(H), b_1x \in E(H)\}$ and at most one vertex in $\{b_2x \in V(L(H))$, for $x \in V(H), b_2x \in E(H)\}$.*

Definition A.18 (J-enlargement)**.** *An* enlargement *of a 3-connected graph J, or a J-enlargement is any 3-connected graph J' such that there is a proper induced subgraph of J' that is isomorphic to a subdivision of J.*

Definition A.19 (J-strip system, uv-rung)**.** *Let J be a 3-connected graph and let G be a perfect graph. A* J-strip system *(S, N) in G means*

- *for each edge uv of J, a subset $S_{uv} = S_{vu}$ of $V(G)$,*
- *for each vertex v of J, s subset N_v of $V(G)$,*
- *$N_{uv} = S_{uv} \cap N_u$,*

such that if we define a uv-rung to be a path R of G with ends s, t, where $V(R) \subseteq S_{uv}$, and s is the unique vertex of R in N_u, and t is the unique vertex of R in N_v, the following conditions are met:

- *the sets S_{uv}, for $uv \in E(J)$ are pairwise disjoint,*
- *for each $u \in V(J)$, $N_u \subseteq \bigcup_{uv \in E(J)} S_{uv}$,*
- *for each $uv \in E(J)$, every vertex of S_{uv} is in a uv-rung,*
- *for any two edges uv, wx of J, with u, v, w, x all distinct, there are no edges between S_{uv} and S_{wx},*
- *if $uv, uw \in E(J)$ with $v \neq w$, then N_{uv} is complete to N_{uw} and there are no other edges between S_{uv} and S_{uw},*
- *for each $uv \in E(J)$, there is a* special *uv-rung, such that for every cycle C of J, the sum of the lengths of the special uv-rungs for $uv \in E(C)$ has the same parity as $|V(C)|$.*

The vertex set of (S, N), denoted by $V(S, N)$ is the set $\bigcup_{uv \in E(J)} S_{uv}$. Note that in general N_{uv} is different from N_{vu}. On the other hand $S_{uv} = S_{vu}$.

A J-strip system has the following properties:

[1] $K_{n,m}$ is a complete bipartite graph with n vertices on the one side and m vertices on the other

- for distinct $u, v \in V(J)$, if $uv \in E(J)$, then $N_u \cap N_v \subseteq S_{uv}$ and if $uv \notin E(J)$, then $N_u \cap N_v = \emptyset$,
- for $uv \in E(J)$ and $w \in V(J)$, if $w \neq u$, then $S_{uv} \cap N_w = \emptyset$,
- for every $uv \in E(J)$, all uv-rungs have lengths of the same parity,
- for every cycle C of J and every choice of uv-rung for every $uv \in E(C)$, the sums of the lengths of the uv-rungs have the same parity as $|V(C)|$. In particular, for each edge $uv \in E(J)$, all uv-rungs have the same parity.

Definition A.20 (choice of rungs)**.** *Given a J-strip system, by a* choice of rungs *we mean the choice of one uv-rung for each edge uv of J.*

Given a square-free perfect graph G and a J-strip system, for every choice of rungs, the subgraph of G induced by their union is the line-graph of a bipartite subdivision of J.

Definition A.21 (saturating J-strip system)**.** $X \subseteq V(G)$ saturates *the strip system if and only if for every $u \in V(J)$ there is at most one neighbor v of u such that $N_{uv} \not\subseteq X$.*

Definition A.22 (major vertex w.r.t. a strip system)**.** *A vertex $v \in V(G) \setminus V(S, N)$ is* major *with respect to (w.r.t.) the strip system if and only if the set of its neighbors saturates the strip system.*

Definition A.23 (major vertex w.r.t. some choice of rungs)**.** *A vertex $v \in V(G) \setminus V(S, N)$ is* major *with respect to (w.r.t.) some choice of rungs if and only if the J-strip system defined by this choice of rungs is saturated by the set of neighbors of v.*

Definition A.24 (subset local w.r.t. a strip system)**.** *A subset $X \subseteq V(S, N)$ is* local *with respect to (w.r.t.) the strip system, if and only if either $X \subseteq N_v$ for some $v \in V(J)$ or $X \subseteq S_{uv}$ for some $uv \in E(J)$.*

The outline of the algorithm is as follows. Given a square-free perfect graph, first we test if it is an Artemis graph. If so, we color it according to Theorem A.1.2. If not, Theorem A.1.1 gives us a prism K. We want to extend K either to a maximal hyperprism or to the line graph of a bipartite subdivision of K_4. For each of them, we can construct a good partition, which we color recursively.

A.2 Notation

In the following algorithms we use a slightly different notation than before, with many concepts represented by inline symbols. This is intended to reduce the length of algorithm's text and simplify its analysis.

- $a :\in X$, when X is a set – let a be equal to any element of X,

- $a \veebar b$, when a and b are logical values – a xor b,
- $V(X)$ – vertices of structure X. Will be written as X when obvious,
- $a - b$, when a and b are vertices – a and b are neighbors,
- $a \cdots b$, when a and b are vertices – a and b are not neighbors,
- $a - X$, when a is a vertex and X is a set of vertices – a has a neighbor in X,
- $a \cdots X$, when a is a vertex and X is a set of vertices – a has a nonneighbor in X,
- $a \blacktriangleleft X$, when a is a vertex and X is a set of vertices – a is complete to X,
- $a \not\triangleleft X$, when a is a vertex and X is a set of vertices – a is anticomplete to X,
- $X \blacksquare Y$, when X and Y are set of vertices – X is complete to Y,
- $X \not\square Y$, when X and Y are set of vertices – X is anticomplete to Y,
- $[n]$ – $\{1, \ldots, n\}$,
- $L(BS(K_4))$– the line-graph of a bipartite subdivision of K_4.

Also, throughout the algorithms, we have many lines with asserts. These check some of the properties required before proceeding, and should all be true.

A.3 Algorithms

Algorithm A.3.1 (Color square-free perfect graph)

```
COLOR-GRAPH (G)
Input: G – square-free Berge graph
Output: A ω(G)-coloring of G
if G is an Artemis graph then                                  // Theorem A.1.1
    return coloring of an Artemis graph G                      // Theorem A.1.2
H ← an induced prism of G                                      // Theorem A.1.1
while P = UNDEFINED do
    if ∃ a component of G \ (H ∪ M) with a set of attachments in H not
     local then
        F ← a minimal component of G \ (H ∪ M) with a set of
         attachments in H not local
        M ← {v : v is a major neighbor of H}
        H' ← GROW-HYPERPRISM(G, H, M, F)
        if H' is a L(BS(K4)) then
            J ← H'
            (S, N) ← a J-strip system
        else
            H ← H'
    else                                          // H is a maximal hyperprism
        M ← {v : v is a major neighbor of H}
        if H is an even hyperprism then
            P ← GOOD-PARTITION-FROM-EVEN-HYPERPRISM(G, H, M)
             break
        else
            P ← GOOG-PARTITION-FROM-ODD-HYPERPRISM(G, H, M)
            break
    if J ≠ UNDEFINED then // a J-strip system was encountered
        J', (S', N') ← GROWING-J-STRIP(G, J, (S, N))
        M ← a set of major vertives w.r.t. (S, N)
        if J', (S', N') is a special K4 system then
            P ← GOOD-PARTITION-FROM-SPECIAL-STRIP-
              SYSTEM(G, (S, N), M)
        else
            P ←
              GOOD-PARTITION-FROM-J-STRIP-SYSTEM(G, J, (S, N), M)
return COLOR-GOOD-PARTITION(G, P)
```

Algorithm A.3.2 (Color good partition)

COLOR-GOOD-PARTITION$(G, (K_1, K_2, K_3, L, R), c_1, c_2)$
Input: G – square-free, Berge graph
(K_1, K_2, K_3, L, R) – good partition
c_1, c_2 – colorings of $G \setminus R$ and $G \setminus L$ (possibly NULL)
Output: $\omega(G)$-coloring of G

$G_1 \leftarrow G \setminus R$
$G_2 \leftarrow G \setminus L$
if $c_1, c_2 =$ NULL **then**
  $c_1 \leftarrow$ COLOR-GRAPH(G_1)
  $c_2 \leftarrow$ COLOR-GRAPH(G_2)
foreach $u \in K_1 \cup K_2$ **do**
  relabel c_2, so that $c_1(u) = c_2(u)$
$B \leftarrow \{u \in K_3 : c_1(u) \neq c_2(u)\}$
if $B = \emptyset$ **then return** $c_1 \cup c_2$
foreach $h \in [2]$, distinct colors i, j **do**
  $G_h^{i,j} \leftarrow$ subgraph induced on G_h by $\{v \in G_h : c_h(v) \in \{i, j\}\}$
foreach $u \in K_3$ **do**
  $C_h^{i,j}(u) \leftarrow$ component of $G_h^{i,j}$ containing u
ASSERT: $C_h^{c_1(u),c_2(u)}(u) \cap K_2 = \emptyset$
if $\exists u \in B, h \in [2] : C_h^{c_1(u),c_2(u)}(u) \cap K_1 = \emptyset$ **then**
  $c_1' \leftarrow c_1$ with colors i and j swapped in $C_1^{i,j}(u)$
  ASSERT: c_1' and c_2 agree on $K_1 \cup K_2$
  ASSERT: $\forall u \in K_3 \setminus B : c_1'(u) = c_1(u)$
  ASSERT: $c_1'(u) = j = c_2(u)$
  return COLOR-GOOD-PARTITION$(G, K_1, K_2, K_3, L, R, c_1', c_2)$
else
  $w \leftarrow$ vertex of B with nost neighbors in K_1
  ASSERT: $\forall u \in B : N(u) \cap K_1 \subset N(w) \cap K_1$
  relabel c_1, c_2, so that $c_1(w) = 1, c_2(w) = 2$
  $P \leftarrow$ a path $w - p_1 - \ldots - p_k - a$ in $C_1^{1,2}(w)$ so that
    $k \geq 1$, $p_1 \in K_3 \cup L$, $p_2 \ldots p_k \in L$, $a \in K$, $c_1(a) \in [2]$
  $Q \leftarrow$ a path $w - q_1 - \ldots - q_l - a$ in $C_2^{1,2}(w)$ so that
    $l \geq 1$, $q_1 \in K_3 \cup R$, $q_2 \ldots q_l \in R$, $a \in K$, $c_2(a) \in [2]$
  $i \leftarrow c_1(a)$
  $j \leftarrow 3 - i$
  ASSERT: exactly one of the colors 1 and 2 appears in K_1 (as in Lemma 2.2.(3))
  ASSERT: $|P|$ and $|Q|$ have different parities
  ASSERT: $p_1 \in K_3 \vee p_2 \in K_3$ (as in Lemma 2.2.(4))
  ASSERT: $\nexists y \in K_3 : c_1(y) = 2 \wedge c_2(y) = 1$ (as in Lemma 2.2.(5))

// **else** // $\nexists u \in B, h \in [2] : C_h^{c_1(u),c_2(u)}(u) \cap K_1 = \emptyset$

if $p_1 \in K_3$ **then**

ASSERT: $c_2(p_1) \notin [2]$

relabel c_2, so that $c_2(p_1) = 3$

ASSERT: color 3 does not appear in K_2

ASSERT: color 3 does not appear in K_1

ASSERT: $C_2^{j,3}(p_1) \cap K_1 = \emptyset$

$c_2' \leftarrow c_2$ with colors j and 3 swapped in $C_2^{j,3}(p_1)$

ASSERT: $j = 2$

return Color-Good-Partition(G, K_1, K_2, K_3, L, R, c_1, c_2')

else

relabel c_1, so that $c_1(q_1) = 3$

if 3 *does not appear in* K_1 **then**

ASSERT: $C_1^{j,3}(q_1) \cap K_1 = \emptyset$

ASSERT: $j = 1$

$c_1' \leftarrow c_1$ with colors j and 3 swapped in $C_1^{j,3}(q_1)$

return Color-Good-Partition(G, K_1, K_2, K_3, L, R, c_1', c_2)

else

ASSERT: $q_1 \not\triangleleft \{a, a_3\}$

ASSERT: $C_1^{i,3}(q_1) \cap K_1 = \emptyset$

ASSERT: $i = 1$

$c_1' \leftarrow c_1$ with colors i and 3 swapped in $C_1^{i,3}(q_1)$

return Color-Good-Partition(G, K_1, K_2, K_3, L, R, c_1', c_2)

Algorithm A.3.3 (Grow hyperprism)

GROW-HYPERPRISM(G, H, M, F) // Lemma 3.3

Input: G – square-free, Berge graph
$H = (A_1, \dots B_3)$ – a hyperprism in G
M – the set of major neighbors of H in G
F – a minimal component of $G \setminus (H \cup M)$ with a set of attachments in H not local.

Output: H' – a larger hyperprism, or
L – a $L(BS(K_4))$

$X \leftarrow$ set of attachments of F in H

if $\exists i : X \cap C_i \neq \emptyset$ **then**
  relabel strips of H, so that $X \cap C_1 \neq \emptyset$
  $x_1 :\in X \cap C_1$
  ASSERT: $X \cap S_2 \neq \emptyset$
  $x_2 :\in X \cap S_2$
  $R_1 \leftarrow$ 1-rung of H, so that $x_1 \in V(R_1)$
  $R_2 \leftarrow$ 2-rung of H, so that $x_2 \in V(R_2)$
  $R_3 \leftarrow$ a 3-rung of H
  $\forall i \in [3] : a_i, b_i \leftarrow$ends of R_i, so that $a_i \in A_i, b_i \in B_i$
  $K \leftarrow$ a prism (R_1, R_2, R_3)
  ASSERT: no vertex in F is major w.r.t. K (as in SPGT 10.5)
  $f_1 - \ldots - f_n \leftarrow$ a minimal path in F, so that
    $f_1 \blacktriangleleft \{a_2, a_3\}$,
    $f_n - R \setminus \{a_1\}$
    there are no other edges between $\{f_1, \dots f_n\}$ and $V(K) \setminus \{a_1\}$
  ASSERT: $F = \{f_1, \dots, f_n\}$
  ASSERT: $f_1 \blacktriangleleft A_3$
  $A'_1 \leftarrow A_1 \cup \{f_1\}$
  $C'_1 \leftarrow C_1 \cup \{f_2, \dots, f_n\}$
  return $H' \leftarrow (A'_1, A_2, \dots, B_3, C'_1, C_2, C_3)$
else
  relabel strips of H, so that there is $\{x_1 :\in A_1, x_2 :\in A_2\} \subset X$ that is not local
  find a path $x - f_1 - \ldots - f_n - x_2$
  ASSERT: $F = \{f_1, \dots f_n\}$
  if n is even and H is even, or n is odd and H is odd **then**
    ASSERT: $f_1 - a_3 \trianglelefteq f_n - b_3$
    if $f_1 - a_3$ **then**
      $H' \leftarrow$ mirrored H – every A_i and B_i are swapped
      return GROW-HYPERPRISM(G, H', M, F)
    else
      if $f_n \blacktriangleleft B_2 \cup B_3$ **then**
        $B'_1 \leftarrow B_1 \cup \{f_n\}$

// **else** // $\forall_{i\in[3]} X \cap C_i = \emptyset$

// **if** n is even and H is even, or n is odd and H is odd **then**

// **else** // $f_n - b_3$

// **if** $f_n \blacktriangleleft B_2 \cup B_3$ **then**

$C_1' \leftarrow C_1 \cup \{f_1, \ldots, f_{n-1}\}$

return $H' \leftarrow \begin{pmatrix} A_1 & C_1' & B_1' \\ A_2 & C_2 & B_2 \\ A_3 & C_3 & B_3 \end{pmatrix}$

else

$\forall_{i\in[3]} : A_i' \leftarrow$ neighbors of f_1 in A_i

$\forall_{i\in[3]} : A_i'' \leftarrow A_i \setminus A_i'$

$\forall_{i\in[3]} : B_i'' \leftarrow$ neighbors of f_n in B_i

$\forall_{i\in[3]} : B_i' \leftarrow B_i \setminus B_i''$

ASSERT: Every i-rung is between A_i' and B_i' or A_i'' and B_i''

$\forall_{i\in[3]} : C_i' \leftarrow$ union of interiors of i-rings between A_i' and B_i'

$\forall_{i\in[3]} : C_i'' \leftarrow$ union of interiors of i-rings between A_i'' and B_i''

ASSERT: $C_i = C_i' \cup C_i''$, $C_i' \cap C_i'' = \emptyset$

ASSERT: $A_i' \cup C_i' \not\boxplus C_i'' \cup B_i''$, $A_i'' \cup C_i'' \not\boxplus C_i \cup B_i$

ASSERT: $A_i' \blacksquare A_i''$, $B_i' \blacksquare B_i''$

ASSERT: $A_1', A_2'', A_3', A_3'' \neq \emptyset$

$H' \leftarrow \begin{pmatrix} A_1' & C_1' & B_1' \\ A_2' \cup A_3' & C_2' \cup C_3' & B_2' \cup B_3' \\ \bigcup_i A_i'' \cup \{f_1\} & \bigcup_i C_i'' \cup \{f_2, \ldots, f_n\} & \bigcup_i B_i'' \end{pmatrix}$

return H'

else

$a_1 \leftarrow$ neighbor of f_1 in A_1

$R_1 \leftarrow$ 1-rung with end a_1

$b_1 \leftarrow$ the other end of R_1

$b_2 \leftarrow$ neighbor of f_2 in B_2

$R_2 \leftarrow$ 2-rung with end b_2

$a_2 \leftarrow$ the other end of R_2

ASSERT: $b_1 \in X$, $a_2 \in X$

ASSERT: $(b_1 - f_1 \wedge a_2 - f_n) \veebar (b_1 - f_n \wedge a_2 - f_1)$

if $f_1 - b_1$ **then**

ASSERT: H is odd

$R_3 \leftarrow$ any 3-rung with ends a_3, b_3, such that $\{a_3, b_3\} \not\boxplus \{f_1, f_n\}$

return $V(R_1) \cup V(R_2) \cup V(R_3) \cup \{f_1, \ldots, f_n\}$ - a $L(BS(K_4))$

// **else** // $\forall_{i\in[3]} X \cap C_i = \emptyset$

// **else** // n is odd and H is even, or n is even and H is odd

45 **else //** $f_1 - a_2$

46 $\forall_{i\in[3]} : A'_i \leftarrow A_i \cap X$, $A''_i \leftarrow A_i \setminus X$

47 $\forall_{i\in[3]} : B'_i \leftarrow B_i \cap X$, $B''_i \leftarrow B_i \setminus X$

48 $\forall_{i\in[3]} : C'_i \leftarrow$ union of i-rungs between A'_i and B'_i

49 $\forall_{i\in[3]} : C''_i \leftarrow$ union of i-rungs between A''_i and B''_i

ASSERT: $C_i = C'_i \cup C''_i$, $C'_i \cap C''_i = \emptyset$

50 **if** f_1 *is complete to at least two of* A_i **then**

51 relabel strips of H, so that f_1 is complete to A_1 and A_2

ASSERT: f_n is complete to B_1 and B_2

ASSERT: $n > 1$ (as in SPGT 10.5)

52 **return** $\begin{pmatrix} A_1 & C_1 & B_1 \\ A_2 & C_2 & B_2 \\ A_3 \cup \{f_1\} & C_3 \cup \{f_2, \ldots, f_{n-1}\} & B_3 \cup \{f_n\} \end{pmatrix}$

53 **else**

ASSERT: $A'_i \blacksquare A''_i$

ASSERT: $B'_i \blacksquare B''_i$

54 **return**

$$\begin{pmatrix} A'_1 & C'_1 & B'_1 \\ A'_2 \cup A'_3 & C_2 \cup C'_3 & B'_2 \cup C'_3 \\ \bigcup_i A''_i \cup \{f_1\} & \bigcup_i C''_i \cup \{f_2, \ldots, f_{n-1}\} & \bigcup_i B''_i \cup \{f_n\} \end{pmatrix}$$

Algorithm A.3.4 (Good partition from an even hyperprism)

GOOD-PARTITION-FROM-EVEN-HYPERPRISM(G, H, M)
Input: G – square-free, Berge graph containing no $L(BS(K_4))$
$H = (A_1, \ldots, B_3)$ – maximal even hyperprism in G
M – set of major neighbors of H
Output: A good partition of G

1 $Z \leftarrow \bigcup\{V(C) : C$ is a component of $G \setminus \{V(H) \cup M\}$ with no attachments in $H\}$
2 relabel strips of H, so that $M \cup A_1$ and $M \cup B_1$ are cliques
3 $F_1 \leftarrow \bigcup\{V(C) : C$ is a component of $G \setminus \{H \cup M \cup Z\}$ that attaches to $A_1 \cup B_1 \cup C_1$ $\}$
ASSERT: M is a clique
ASSERT: $M \cup A_i$ is a clique for at least two values of i
ASSERT: $M \cup B_j$ is a clique for at least two values of j
4 $K_1 \leftarrow A_1$, $K_2 \leftarrow M$, $K_3 \leftarrow B_1$
5 $R \leftarrow C_1 \cup F_1 \cup Z$
6 $L \leftarrow G \setminus \{K_1 \cup K_2 \cup K_3 \cup R\}$
7 **return** (K_1, K_2, K_3, L, R)

Algorithm A.3.5 (Good partition from an odd hyperprism)

GOOD-PARTITION-FROM-ODD-HYPERPRISM(G, H, M)
Input: G – square-free, Berge graph containing no $L(BS(K_4))$
$H = (A_1, \ldots, B_3)$ – maximal odd hyperprism in G
M – set of major neighbors of H
Output: A good partition of G

$Z \leftarrow \bigcup\{V(C) : C$ is a component of $G \setminus \{V(H) \cup M\}$ with no attachments in $H\}$
relabel strips of H, so that $A_1 \not\subseteq B_1$ and $A_2 \not\subseteq B_2$
ASSERT: $C_1 \neq \emptyset, C_2 \neq \emptyset$
$\forall_{i \in [3]} F_i \leftarrow \bigcup\{V(C) : C$ is a component of $G \setminus \{H \cup M \cup Z\}$ that attaches to $A_i \cup B_i \cup C_i$ $\}$
$F_B \leftarrow \bigcup\{V(C) : C$ is a component of $G \setminus \{H \cup M \cup Z \cup F_1 \cup F_2 \cup F_3\}$ that attaches to $B_1 \cup B_2 \cup B_3$ $\}$
ASSERT: At least two of A_i and at least two of B_i are cliques
ASSERT: M is complete to at least two of A_i and at least two of B_i
ASSERT: M is a clique
ASSERT: For at least two i : $A_i \cup M$ is a clique
ASSERT: For at least two j : $A_j \cup M$ is a clique
choose h, so that $M \cup A_h$ and $M \cup B_h$ are cliques
if $h = 1 \vee h = 2$ **then**
    $K_1 \leftarrow A_1$, $K_2 \leftarrow M$, $K_3 \leftarrow B_1$
    $R \leftarrow C_1 \cup F_1 \cup Z$
    $L \leftarrow G \setminus \{K_1 \cup K_2 \cup K_3 \cup R\}$
    return (K_1, K_2, K_3, L, R)
else
    relabel H so that $M \cup A_1$ and $M \cup B_2$ are cliques
    $K_1 \leftarrow B_2 \cup B_3$, $K_2 \leftarrow M$, $K_3 \leftarrow A_1 \cup A_3$
    $L \leftarrow B_1 \cup C_1 \cup F_1 \cup F_B$
    $R \leftarrow G \setminus \{K_1 \cup K_2 \cup K_3 \cup L\}$
    return (K_1, K_2, K_3, L, R)

Algorithm A.3.6 (Good partition from a J-strip system)

GOOD-PARTITION-FROM-J-STRIP-SYSTEM$(G, J, (S, N), M)$
Input: G – square-free, Berge graph
J – a maximal 3-connected graph with appearance in G
(S, N) – a maximal J-strip system
M – a set of major vertices w.r.t. (S, N)
Output: A good partition of G

$S^*_{uv} \leftarrow S_{uv} \cup$ (components of $G \setminus V(S, N)$ that attach in S_{uv} only)
$T_{uv} \leftarrow N_u \cap N_v$
ASSERT: $T_{uv} \blacksquare N_u \setminus N_{uv}$, $T_{uv} \blacksquare N_v \setminus N_{vu}$
ASSERT: $M \cup T_{uv}$ is a clique
Strip S_{uv} is *rich* if and only if $S_{uv} \setminus T_{uv} \neq \emptyset$
if $\exists S_{uv}$ – *a rich strip in* (S, N) **then**
  if $\exists S_{uv}$ – *a rich strip in* (S, N)*, such that* $M \cup (N_u \setminus N_{uv})$ *and* $M \cup (N_v \setminus N_{vu})$ *are cliques* **then**
    $K_1 \leftarrow N_u \setminus N_{uv}$, $K_2 \leftarrow M \cup T_{uv}$, $K_3 \leftarrow N_v \setminus N_{vu}$
    $L \leftarrow (S^*_{uv} \setminus T_{uv}) \cup$ (components of $G \setminus V(S, N)$ that attach only to N_u and those that attach only to N_v)
    $R \leftarrow G \setminus (K_1 \cup K_2 \cup K_3 \cup L)$
    return (K_1, K_2, K_3, L, R)
  else
    $S_{uv} \leftarrow$ a rich strip in (S, N), such that $M \cup (N_u \setminus N_{uv})$ is not a clique and $M \cup (N_v \setminus N_{vu})$ is a clique
    $K_1 \leftarrow N_{uv} \setminus T_{uv}$, $K_2 \leftarrow M \cup T_{uv}$, $K_3 \leftarrow N_v \setminus N_{vu}$
    $R \leftarrow (S^*_{uv} \setminus N_u) \cup$ (components of $G \setminus V(S, U)$ that attach only to N_v)
    $L \leftarrow G \setminus (K_1 \cup K_2 \cup K_3 \cup R)$
    return (K_1, K_2, K_3, L, R)
else
  ASSERT: $\forall uv \in E(J) : S_{uv} = T_{uv}, S_{uv}$ is a clique
  $S_{uv} \leftarrow$ any strip
  $K_1 \leftarrow N_u \setminus S_{uv}$, $K_2 \leftarrow M$, $K_3 \leftarrow N_v \setminus S_{uv}$
  $L \leftarrow S^*_{uv} \cup$ (components of $G \setminus V(S, N)$ that attach only to N_u and only to N_v)
  $R \leftarrow G \setminus (K_1 \cup K_2 \cup K_3 \cup L)$
  return (K_1, K_2, K_3, L, R)

Algorithm A.3.7 (Good partition from a special K_4 strip system)

GOOD-PARTITION-FROM-SPECIAL-STRIP-SYSTEM$(G, J, (S, N), M)$
Input: G – square-free, Berge graph
(S, N) – a special K_4 strip system
M – a set of major vertices w.r.t. (S, N)
Output: A good partition of G

$\forall_{i,j\in[4]} O_{ij} \leftarrow$ set of vertices in $V(G) \setminus V(S, N)$ that are complete to $(N_i \cup N_j) \setminus S_{ij}$ and anticomplete to $V(S, N) \setminus (N_i \cup N_j \cup S_{ij})$
if $(N_1 \setminus N_{12}) \cup M \cup O_{12}$ *and* $(N_2 \setminus N_{12}) \cup M \cup O_{12}$ *are cliques* **then**
  $K_1 \leftarrow N_1 \setminus N_{12}$, $K_2 \leftarrow O_{12} \cup M$, $K_3 \leftarrow N_2 \setminus N_{12}$
  $L \leftarrow$ union of those components of $G \setminus (K_1 \cup K_2 \cup K_3)$ that contain vertices of S_{12}
  $R \leftarrow G \setminus (K_1 \cup K_2 \cup K_3 \cup L)$
  return (K_1, K_2, K_3, L, R)
else
  if $(N_1 \setminus N_{12}) \cup M \cup O_{12}$ *is a clique* **then**
    relabel 1 and 2 in J so that $(N_1 \setminus N_{12}) \cup M \cup O_{12}$ is not a clique
  ASSERT: $N_{12} \cup M \cup O_{12}$ is a clique
  if $N_{21} \cup M \cup O_{12}$ *is a clique* **then**
    $X \leftarrow N_{21}$
  else
    $X \leftarrow N_2 \setminus N_{21}$
  $K_1 \leftarrow N_{12}$, $K_2 \leftarrow M \cup O_{12}$, $K_3 \leftarrow X$
  $L \leftarrow$ component of $G \setminus (K_1 \cup K_2 \cup K_3)$ that contains $N_1 \setminus N_{12}$
  $R \leftarrow G \setminus (K_1 \cup K_2 \cup K_3 \cup L)$
  return (K_1, K_2, K_3, L, R)

Algorithm A.3.8 (Find a special K_4 system)

FIND-SPECIAL-K4-STRIP-SYSTEM$(G, J, (S, N), m)$
Input: G – square-free, Berge graph
J – a 3-connected graph with appearance in G
(S, N) – a J-strip system
$m \in G \setminus (S, N)$ that is major w.r.t. some choice of rungs of (S, N) but not w.r.t. (S, N)
Output: (S', N') – A special K_4-strip system, or
(S'', N'') – a bigger J-strip system

1 $X \leftarrow N(m)$
2 $M \leftarrow$ vertices of $G \setminus V(S, N)$ major w.r.t. (S, N)
3 $M^* \leftarrow$ vertices of $G \setminus V(S, N)$ major w.r.t. some choice of rungs
ASSERT: $J = K_4$ (as in SPGT 8.4)
4 $V(J) \leftarrow [4]$
5 $\forall_{i \neq j \in [4]}$: choose rungs R_{ij}, R_{ij}, forming line graphs $L(H)$ and $L(H')$ so that X saturates $L(H)$ but does not saturate $L(H')$
ASSERT: $R_{ij} \neq R'_{ij}$ if and only if $\{i, j\} = [2]$
6 $r_{ij}, r_{ji} \leftarrow$ ends of each R_{ij}
7 $r'_{ij}, r'_{ji} \leftarrow$ ends of each R'_{ij}
8 $\forall_{i \in [4]} T_i \leftarrow \{r_{ij}, j \in [4] \setminus \{i\}\}$
9 $\forall_{i \in [4]} T'_i \leftarrow \{r'_{ij}, j \in [4] \setminus \{i\}\}$
ASSERT: X has at least two members in each $T_1, \dots T_4$
ASSERT: There is T'_i that contains at most one member of X
ASSERT: $T_3 = T'_3, T_4 = T'_4$
10 relabel 1 and 2 in J, so that $|X \cap T_1| = 2$ and $|X \cap T'_1| = 1$
ASSERT: $r_{12} \in X, r'_{12} \notin X$
ASSERT: $r_{13} \in X \trianglelefteq r_{14} \in X$
11 relabel 3 and 4 in J, so that $r_{13} \in X, r_{14} \notin X$
ASSERT: R_{34} is even and $[X \cap V(L(H'))] \setminus V(R_{34}) = \{r_{31}, r_{32}, r_{41}, r_{42}\}$ (as in 6.3.(3))
ASSERT: R_{14} has odd length, $r_{21} \in X$ (as in 6.3.(4))
ASSERT: R_{12} has length 0, every 12-rung has even length (as in 6.3.(5))
ASSERT: R_{24} has length 0 and R_{23} has odd length (as in 6.3.(6))
ASSERT: Every 34-rung has non-zero even length (as in 6.3.(7))
12 $\forall_{i \neq j \in [4]} O_{ij} \leftarrow$ set of vertices that are not major w.r.t $L(H')$ and are complete to $(T'_i \cup T'_j) \setminus R'_{ij}$
ASSERT: $r_{12} = r_{21} \in O_{12}$, $m \in O_{34}$
ASSERT: $O_{34} = M^* \setminus M$
13 $(S', N') \leftarrow$ strip system obtained from (S, N) by replacing S_{12} with $S_{12} \setminus O_{12}$
14 **if** $\exists_{rung R}$: *adding* R *to* S'_{12} *produces enlargement of* (S, N) **then**
15 **return** (S'', N'') – an enlargement of (S, N)
16 **else**
17 **return** (S', N') – a special K_4 strip system

Algorithm A.3.9 (Growing a J-strip)

GROWING-J-STRIP$(G, J, (S, N))$
Input: G – square-free, Berge graph
J – a 3-connected graph with appearance in G
(S, N) – a J-strip system

Output: J' and a maximal J'-strip system, or a special strip system

$M \leftarrow$ vertices of $G \setminus V(S, N)$ that are major on some choice of Rungs of (S, N)
if $\exists m : m$ *is not major on some choice of rungs of* (S, N) **then**
  $OUT \leftarrow$ FIND-SPECIAL-K4-STRIP-SYSTEM$(G, J, (S, N), m)$
  if OUT *is a special strip system* **then**
    return OUT
  else
    return GROWING-J-STRIP(G, OUT)
else if $\exists F$ *: F is a component of* $G \setminus (V(S, N) \cup M)$*, such that no member of F is major w.r.t.* (S, N) *and set of attachments of F on H is not local* **then**
  ASSERT: (as in 6.2, or actually SPGT 8.5)
  $F \leftarrow$ minimal component with this property
  if $\exists v \in V(J) : X \subset \bigcup(S_{uv} : uv \in E(J))$ **then**
    $x :\in X \cap S_{uv} \setminus N_v$, for some $uv \in E(J)$
    $x' :\in X \cap S_{u'v}$, for some $u'v \in E(J), u' \neq u$
    ASSERT: $\{x, x'\}$ is not local w.r.t. (S, N)
    $L(H) \leftarrow \forall_{i,j \in E(J)}$ choose ij-rung R_{ij}, so that $x \in V(R_{uv}), x' \in V(R_{u'v})$
    ASSERT: $\{x, x'\}$ is not local w.r.t. $L(H)$
    $H \leftarrow$ inverse line graph of $L(H)$
    $D \leftarrow$ a branch of H with ends d, u: $\delta_H(d) \setminus E(D) = (X \cap E(H)) \setminus E(D)$
    $P \leftarrow$ a path with ends p_1, p_2, so that:
      $p_1 \blacktriangleleft N_v \setminus N_{vu}$ and no other vertex of P has neighbors in $N_v \setminus N_{uv}$
      $p_2 - x$ and no other vertex of P has neighbors in $S_{uv} \setminus N_v$
    $(S', N') \leftarrow$ add p_1 to N_v and F to S_{uv}
    return GROWING-J-STRIP $(G, J, (S', N'))$
  else
    $K \leftarrow \{uv \in E(J) : X \cap S_{uv} \neq \emptyset\}$
    ASSERT: There are two disjoint edges in K (as in SPGT 8.5.(3))
    F is a vertex set of a path $\leftarrow f_1 - \ldots - f_n$
    ASSERT: Every choice of rungs is broad

```
    // if ∃F ... then
        // else  // ∄v ∈ V(J) : X ⊂ ⋃(S_uv : uv ∈ E(J))
            ASSERT: every choice of rungs has the same traversal. (Hard to
             assert)
22          ij ← the traversal edge
23          A_1 ← N_i \ S_ij, A_2 ← N_j \ S_ij
            ASSERT: X ∩ (V(S, N) \ S_ij) = A_1 ∪ A_2
24          if n = 1 then
25              (S', N') ← add f_1 to N_i, N_j, S_ij
26              return GROWING-J-STRIP (G, J, (S', N'))
27          else
28              x_1 :∈ A_1, x_2 :∈ A_2, so that x_1 and x_2 are in disjoint strips
                ASSERT: x_1 − f_1 ⊻ x_1 − f_n
29              if x_1 − f_n then
30                  relabel f_1 − ... − f_n front to back
31              (S', N') ← add f_1 to N_i, f_n to N_j and F to S_ij
32              return GROWING-J-STRIP (G, J, (S', N'))
33  else
34      return J, (S, N) – a maximal J-strip
```

Bibliography

[1] Pablo Adasme, Abdel Lisser, and Ismael Soto. "Robust semidefinite relaxations for a quadratic OFDMA resource allocation scheme". In: *Computers & Operations Research* 38.10 (Oct. 2011), pp. 1377–1399. DOI: 10.1016/j.cor.2011.01.002. URL: https://doi.org/10.1016%2Fj.cor.2011.01.002.

[2] Josef Weidendorfer et al. *Callgrind*. URL: https://valgrind.org/docs/manual/cl-manual.html.

[3] Konstantinos Ampountolas, Nan Zheng, and Nikolas Geroliminis. "Macroscopic modelling and robust control of bi-modal multi-region urban road networks". In: *Transportation Research Part B: Methodological* 104 (Oct. 2017), pp. 616–637. DOI: 10.1016/j.trb.2017.05.007. URL: https://doi.org/10.1016%2Fj.trb.2017.05.007.

[4] Saman Ashkiani, Shengren Li, Martin Farach-Colton, Nina Amenta, and John D. Owens. "GPU LSM: A Dynamic Dictionary Data Structure for the GPU". In: *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, May 2018. DOI: 10.1109/ipdps.2018.00053. URL: https://doi.org/10.1109/ipdps.2018.00053.

[5] Claude Berge. "Färbung von Graphen, deren sämtliche beziehungsweise deren ungerade Kreise starr sind". In: Wissenschaftliche Zeitschrift der Martin-Luther-Universität Halle-Wittenberg, Mathematisch-naturwissenschaftliche Reihe, 1961, p. 114.

[6] Hans L. Bodlaender and Klaus Jansen. "On the complexity of the maximum cut problem". In: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 1994, pp. 769–780. DOI: 10.1007/3-540-57785-8_189. URL: https://doi.org/10.1007/3-540-57785-8_189.

[7] Béla Bollobás. *Modern graph theory*. New York: Springer, 1998. ISBN: 978-0-387-98488-9.

[8] *Boost c++ libraries*. URL: https://www.boost.org/.

[9] Brian Borchers. *CSDP*. URL: https://github.com/coin-or/Csdp.

[10] Brian Borchers. "CSDP, A C library for semidefinite programming". In: *Optimization Methods and Software* 11.1-4 (1999), pp. 613–623. DOI: 10.1080/10556789908805765. URL: https://doi.org/10.1080/10556789908805765.

[11] Maria Chudnovsky, Gérard Cornuéjols, Xinming Liu, Paul Seymour, and Kristina Vušković. "Recognizing Berge Graphs". In: *Combinatorica* 25.2 (Mar. 2005), pp. 143–186. DOI: 10.1007/s00493-005-0012-8. URL: https://doi.org/10.1007/s00493-005-0012-8.

[12] Maria Chudnovsky, Aurélie Lagoutte, Paul Seymour, and Sophie Spirkl. "Colouring perfect graphs with bounded clique number". In: *Journal of Combinatorial Theory, Series B* 122 (Jan. 2017), pp. 757–775. DOI: 10.1016/j.jctb.2016.09.006. URL: https://doi.org/10.1016/j.jctb.2016.09.006.

[13] Maria Chudnovsky, Irene Lo, Frédéric Maffray, Nicolas Trotignon, and Kristina Vušković. "Coloring square-free Berge graphs". In: *Journal of Combinatorial Theory, Series B* 135 (Mar. 2019), pp. 96–128. DOI: 10.1016/j.jctb.2018.07.010. URL: https://doi.org/10.1016/j.jctb.2018.07.010.

[14] Maria Chudnovsky, Neil Robertson, P. D. Seymour, and Robin Thomas. "Progress on perfect graphs". In: *Mathematical Programming* 97.1 (July 2003), pp. 405–422. DOI: 10.1007/s10107-003-0449-8. URL: https://doi.org/10.1007/s10107-003-0449-8.

[15] Maria Chudnovsky, Neil Robertson, Paul Seymour, and Robin Thomas. "The strong perfect graph theorem". In: *Annals of Mathematics* 164.1 (July 2006), pp. 51–229. DOI: 10.4007/annals.2006.164.51. URL: https://doi.org/10.4007/annals.2006.164.51.

[16] Vašek Chvátal. "On certain polytopes associated with graphs". In: *Journal of Combinatorial Theory, Series B* 18.2 (Apr. 1975), pp. 138–154. DOI: 10.1016/0095-8956(75)90041-6. URL: https://doi.org/10.1016/0095-8956(75)90041-6.

[17] Vašek Chvátal and Najiba Sbihi. "Bull-free Berge graphs are perfect". In: *Graphs and Combinatorics* 3.1 (Dec. 1987), pp. 127–139. DOI: 10.1007/bf01788536. URL: https://doi.org/10.1007/bf01788536.

[18] Michele Conforti, Gérard Cornuéjols, and Kristina Vušković. "Square-free perfect graphs". In: *Journal of Combinatorial Theory, Series B* 90.2 (Mar. 2004), pp. 257–307. DOI: 10.1016/j.jctb.2003.08.003. URL: https://doi.org/10.1016/j.jctb.2003.08.003.

[19] Gérard Cornuéjols. "The strong perfect graph theorem". In: *Optima* 70 (June 2003), pp. 2–6. URL: https://www.andrew.cmu.edu/user/gc0v/webpub/optima.pdf.

[20] Gérard Cornuéjols and William H. Cunningham. "Compositions for perfect graphs". In: *Discrete Mathematics* 55.3 (1985), pp. 245–254. DOI: 10.1016/s0012-365x(85)80001-7. URL: https://doi.org/10.1016/s0012-365x(85)80001-7.

[21] Alexander K. Dewdney. “Reductions Between NP-complete Problems”. In: *Fast Turing Reductions Between Problems in NP*. University of Western Ontario: Department of Computer Science, University of Western Ontario, 1981. Chap. 4. ISBN: 9780771403057. URL: https://doi.org/10.1007/978-3-642-78240-4.

[22] José Fonseca. *gprof2dot*. URL: https://github.com/jrfonseca/gprof2dot.

[23] Martin Grötschel, László Lovász, and Alexander Schrijver. “Coloring Perfect Graphs”. In: *Geometric Algorithms and Combinatorial Optimization*. Springer Berlin Heidelberg, 1993. Chap. 9.1, pp. 296–299. DOI: 10.1007/978-3-642-78240-4. URL: https://doi.org/10.1007/978-3-642-78240-4.

[24] Martin Grötschel, László Lovász, and Alexander Schrijver. “Stable Sets in Graphs”. In: *Geometric Algorithms and Combinatorial Optimization*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993. Chap. 9, pp. 273–303. ISBN: 9783642782404. URL: https://doi.org/10.1007/978-3-642-78240-4.

[25] Donald E. Knuth. “The Sandwich Theorem”. In: *The Electronic Journal of Combinatorics* 1.1 (Apr. 1994). DOI: 10.37236/1193. URL: https://doi.org/10.37236/1193.

[26] Dénes König. “Über Graphen und ihre Anwendung auf Determinantentheorie und Mengenlehre”. In: *Mathematische Annalen* 77.4 (Dec. 1916), pp. 453–465. DOI: 10.1007/bf01456961. URL: https://doi.org/10.1007/bf01456961.

[27] Monique Laurent and Franz Rendl. “Semidefinite Programming and Integer Programming”. In: *Discrete Optimization*. Elsevier, 2005, pp. 393–514. DOI: 10.1016/s0927-0507(05)12008-8. URL: https://doi.org/10.1016/s0927-0507(05)12008-8.

[28] Benjamin Lévêque, Frédéric Maffray, Bruce Reed, and Nicolas Trotignon. “Coloring Artemis graphs”. In: *Theoretical Computer Science* 410.21-23 (May 2009), pp. 2234–2240. DOI: 10.1016/j.tcs.2009.02.012. URL: https://doi.org/10.1016/j.tcs.2009.02.012.

[29] Dajie Liu, Stojan Trajanovski, and Piet Van Mieghem. “ILIGRA: An Efficient Inverse Line Graph Algorithm”. In: *Journal of Mathematical Modelling and Algorithms in Operations Research* 14.1 (Apr. 2014), pp. 13–33. DOI: 10.1007/s10852-014-9251-2. URL: https://doi.org/10.1007/s10852-014-9251-2.

[30] László Lovasz. “On the Shannon capacity of a graph”. In: *IEEE Transactions on Information Theory* 25.1 (Jan. 1979), pp. 1–7. DOI: 10.1109/tit.1979.1055985. URL: https://doi.org/10.1109/tit.1979.1055985.

[31] László Lovász. “Normal hypergraphs and the perfect graph conjecture”. In: *Discrete Mathematics* 2.3 (June 1972), pp. 253–267. DOI: 10.1016/0012-365x(72)90006-4. URL: https://doi.org/10.1016/0012-365x(72)90006-4.

[32] László Lovász. *Semidefinite programs and combinatorial optimization (Lecture notes)*. 1995. DOI: https://doi.org/10.1007/0-387-22444-0_6. URL: http://www.cs.elte.hu/~lovasz/semidef.ps.

[33] László Lovász, Michael Saks, and Alexander Schrijver. "Orthogonal representations and connectivity of graphs". In: *Linear Algebra and its Applications* 114-115 (Mar. 1989), pp. 439–454. DOI: 10.1016/0024-3795(89)90475-8. URL: https://doi.org/10.1016/0024-3795(89)90475-8.

[34] Frédéric Maffray. "Even pairs in square-free Berge graphs with no odd prism". In: (Feb. 2015). URL: https://arxiv.org/abs/1502.03695.

[35] Frédéric Maffray and Nicolas Trotignon. "A class of perfectly contractile graphs". In: *Journal of Combinatorial Theory, Series B* 96.1 (Jan. 2006), pp. 1–19. DOI: 10.1016/j.jctb.2005.06.011. URL: https://doi.org/10.1016/j.jctb.2005.06.011.

[36] Frédéric Maffray and Nicolas Trotignon. "Algorithms for Perfectly Contractile Graphs". In: *SIAM Journal on Discrete Mathematics* 19.3 (Jan. 2005), pp. 553–574. DOI: 10.1137/s0895480104442522. URL: https://doi.org/10.1137/s0895480104442522.

[37] Colin McDiarmid. *Graph imperfection and channel assignment*. 1999. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.98.9490.

[38] Colin McDiarmid and Bruce Reed. "Channel assignment and weighted coloring". In: *Networks* 36.2 (2000), pp. 114–117. DOI: 10.1002/1097-0037(200009)36:2<114::aid-net6>3.0.co;2-g. URL: https://doi.org/10.1002/1097-0037(200009)36:2%3C114::aid-net6%3E3.0.co;2-g.

[39] Brendan McKay. *Graph repository*. URL: https://users.cecs.anu.edu.au/~bdm/data/graphs.html.

[40] Haiko Müller. "Hamiltonian circuits in chordal bipartite graphs". In: *Discrete Mathematics* 156.1-3 (Sept. 1996), pp. 291–298. DOI: 10.1016/0012-365x(95)00057-4. URL: https://doi.org/10.1016/0012-365x(95)00057-4.

[41] Jan Mycielski. "Sur le coloriage des graphs". In: *Colloquium Mathematicum* 3.2 (1955), pp. 161–162. DOI: 10.4064/cm-3-2-161-162. URL: https://doi.org/10.4064/cm-3-2-161-162.

[42] *Nvidia cuSOLVER*. URL: https://developer.nvidia.com/cusolver.

[43] Jorge L. Ramírez Alfonsin and Bruce A. Reed, eds. *Perfect graphs*. Wiley-Interscience series in discrete mathematics and optimization. Chichester ; New York: Wiley, 2001. ISBN: 9780471489702.

[44] Florian Roussel and Philippe Rubio. "About Skew Partitions in Minimal Imperfect Graphs". In: *Journal of Combinatorial Theory, Series B* 83.2 (Nov. 2001), pp. 171–190. DOI: 10.1006/jctb.2001.2044. URL: https://doi.org/10.1006/jctb.2001.2044.

[45] Gilbert Strang. *Linear Algebra and Its Applications, 4th Edition*. Cengage Learning, Feb. 2020. ISBN: 0030105676. URL: https://www.xarg.org/ref/a/0030105676/.