Jagiellonian University
Department of Theoretical Computer Science

Adrian Siwiec

Perfect Graph Recognition and Coloring

Master Thesis

Supervisor: dr in. Krzysztof Turowski

September 2020

Abstract

TODOa

List of definitions

List of Algorithms

Contents

Definitions

Run proofreader on all text (temporarily disabled because it slowed down IDE)

We use standard definitions, sourced from the book by **BB98 BB98**, modified and extended as needed.

Definition 0.1 (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.

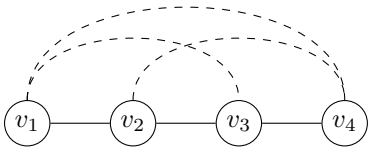


Figure 1: An example graph G_0

An edge $\{x, y\}$ is said to *join*, or be between vertices x and y and is denoted by xy. Thus xy and yx mean 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. If $xy \notin E(G)$ then xy is a *nonedge* and x and y are *anticonnected*.

Figure 1 shows an example of a 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 0.2 (subgraph)**.** $G' = (V', E')$ *is a* subgraph *of* $G = (V, E)$ *if* $V' \subseteq V$ *and* $E' \subseteq E$*.*

Definition 0.3 (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* 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 0.4 (X-completeness)**.** *Given set $X \subseteq V$, vertex $v \notin X$ is* X-complete *if it is adjacent to every node $x \in X$. A set $Y \subseteq V$ is X-complete if $X \cap Y = \emptyset$ and every node $y \in Y$ is X-complete.*

Definition 0.5 (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 $x_1 - x_2 - \ldots - x_l$. 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^*.

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

Definition 0.6 (connected graph, subset)**.** *A graph G is* connected *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 the graph $G[X]$ is connected.*

Definition 0.7 (component)**.** *A* component *of a graph G is its maximal connected induced subgraph.*

Definition 0.8 (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 $x_1 - x_2 - \ldots - x_l - x_1$. $l = |E(C)|$ is the *length* of the cycle C. Sometimes we will denote the cycle of length l as C_l.

Notice, that a cycle is not a path (nor is a path a cycle). If we add an edge v_1v_4 to the path G_0 it becomes an even cycle C_4.

Definition 0.9 (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*.

Definition 0.10 (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 0.11 (anticonnected graph, subset)**.** *A graph G is anticonnected if $\overline{G}$ is connected. A subset X is anticonnected if $\overline{G}[X]$ is connected.*

Definition 0.12 (anticomponent)**.** *An* anticomponent *of a graph G is an induced subgraph whose complement is a component in $\overline{G}$.*

Definition 0.13 (antihole)**.** *An* antihole *is a subgraph of G whose complement is a hole in G.*

Definition 0.14 (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 0.15 (clique number)**.** *A* clique number *of a graph G, denoted as $\omega(G)$, is a cardinality of its largest induced clique.*

Definition 0.16 (anticlique)**.** *An* anticlique *is a graph in which there are no edges. We will also call anticliques* independent sets*.*

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

Definition 0.18 (coloring)**.** *Given a graph G, its* coloring *is a function $c : V(G) \to \mathbb{N}^+$, such that $c(x) \neq c(y)$ for every edge $xy \in E(G)$. A k-coloring of G (if exists) is a coloring, such that $c(x) \leq k$ for all vertices $x \in V(G)$.*

Definition 0.19 (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.*

Defs below to ch. 1

Chapter 1

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 triangle and requiring arbitrarily large numbers of colors (e.g. construction by Mycielski [**Mycielski1955**]).

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

Definition 1.1 (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 [**CB61**] 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 [**grotschel1993**]. Other classical NP-complete problems are still NP-complete in perfect graphs e.g. Hamiltonian path [**Mller1996**], maximum cut problem [**Bodlaender1994**] or dominating set problem [**Dewdney81**].

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 1.1) stating that a graph is perfect if and only if it is Berge and an algorithm for recognizing Berge graphs in polynomial time (section 1.2).

all theese citations are about subclasses e.g. biparite graphs. Should we mention some subclasses?

Definition 1.2 (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 [**Chvtal1975**, **Chudnovsky2003**], radio channel assignment problem [**McDiarmid99**, **McDiarmid2000**] and appear in the theory of Shannon capacity of a graph [**Lovasz1979**]. Also, as pointed out in [**alfonsinPerfect2001**, **Chudnovsky2003**] algorithms to solve semi-definite programs grew out of the theory of perfect graphs. We will take a look at semi-definite programs and at perfect graph's relation to Shannon capacity in section 2.1.1.

1.1 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 [**CB61**] and then proven by Lovász in 1972 [**LL72**].

Theorem 1.1.1 (Perfect graph theorem)**.** *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 1.1.2. *A graph G is perfect if and only if for every induced subgraph H, the number of vertices of H is at most $\alpha(H)\omega(H)$.* proof!

Then, since $\alpha(H) = \omega(\overline{H})$ and $\omega(H) = \alpha(\overline{H})$ theorem 1.1.2 implies theorem 1.1.1.

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 1.2) 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 **MC06** [**MC06**].

Theorem 1.1.3 (Strong perfect graph theorem)**.** *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 [**GC03**].

Basic classes of perfect graphs

Bipartite graphs are perfect, since we can color them with two colors. From the theorem of König we get that line graphs of bipartite graphs are also perfect [**Knig1916**, **GC03**]. From the perfect graph theorem (theorem 1.1.1) 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 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 [**Cornujols1985**].

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 [**Chvtal1987**].

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 1.1.3 is a consequence of the *Decomposition theorem*:

Theorem 1.1.4 (Decomposition theorem)**.** *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 [**MC06**] for the proof of theorem 1.1.3 and theorem 1.1.4.

1.2 Recognizing Berge Graphs

The following is based on the paper by Maria **MC05 MC05** [**MC05**]. We will not provide full proof of its correctness, but 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 1.2.1). 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 (section 1.2.2).

1.2.1 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:

- $\forall_{1 \leq i \leq 3}$ P_i is a path between a and b_i,

[1] A triangle is a clique K_3.

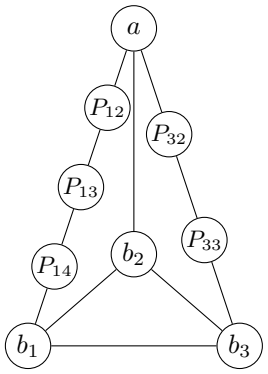


Figure 1.1: An example of a pyramid.

- $\forall_{1 \le i < j \le 3}$ 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 can be *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.

Finding Pyramids

Algorithm 1.2.1 (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 b_1, b_2, b_3 do
3:         for each s_1, s_2, s_3, such that for 1 ≤ i < j ≤ 3, {b_i, s_i} is disjoint
               from {b_j, s_j} and b_i b_j is the only edge between them do
4:             if there is a vertex a, adjacent to all of s_1, s_2, s_3, and to at most
                   one of b_1, b_2, b_3, such that if a is adjacent to b_i, then b_i = s_i
                   then
5:                 M ← V(G) \ {b_1, b_2, b_3, s_1, s_2, s_3}
6:                 for each m ∈ M do
7:                     S_1(m) ← the shortest path between s_1 and m such that
                         s_2, s_3, b_2, b_3 have no neighbors in its interior, if such a
                         path exists.
8:                     calculate S_2(m), S_3(m) similarly
9:                     T_1(m) ← the shortest path between m and b_1, such that
                         s_2, s_3, b_2, b_3 have no neighbors in its interior, if such a
                         path exists
10:                    calculate T_2(m), T_3(m) similarly
11:                end for
12:                if s_1 = b_1 then                    \\ calculate all possible P_1 paths
13:                    P_1(b_1) ← the one-vertex path with vertex b_1
14:                    for each m ∈ M do  P_1(m) ← UNDEFINED
15:                else
16:                    P_1(b_1) ← UNDEFINED
17:                    for each m ∈ M do
18:                        if  m is nonadjacent to all of b_2, b_3, s_2, s_3  and
                               S_1(m) and T_1(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)
                color all other vertices white.
30:           for each m_2 ∈ M ∪ {b_2} do
31:             if P_2(m_2) exists and contains no black vertices
                    then
32:               add (m_1, m_2) to good_pairs_{1,2}
33:             end if
34:           end for
35:         end if
36:       end for
37:       calculate good_pairs_{1,3} and good_pairs_{2,3} in similar way
38:       for each triple m_1, m_2, m_3 such that m_i ∈ M ∪ {b_i} do
39:         if ∀_{1≤i<j≤3}: (m_i, m_j) is a good (i,j)-pair then
40:           return TRUE
41:         end if
42:       end for
43:     end if
44:   end for
45: end for
46: return FALSE
47: 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 40 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 38. The proof of the converse is rather technical and we refer

to [**MC05**] for it.

dokadny nr twierdzenia

Now we will proove the time complexity.

Theorem 1.2.1. *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 4-43 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.

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 31.

Finally, there are $O(|V|^3)$ pairs m_1, m_2, m_3 and checking each takes $O(1)$ time. □

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.

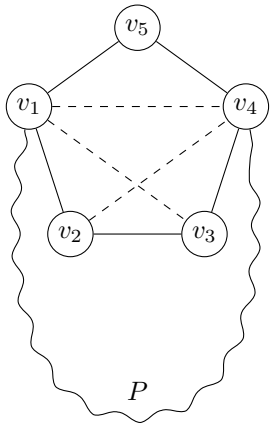


Figure 1.2: An example of a jewel.

Most obvious way to find a jewel would be 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, 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 3.1 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.

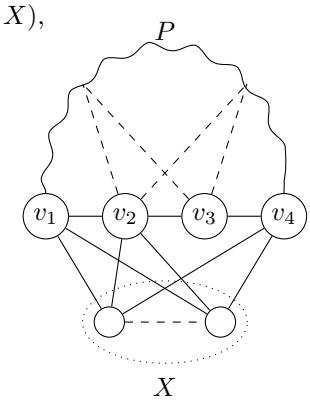


Figure 1.3: An example of a $\mathcal{T}_2$.

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 existence of $\mathcal{T}_2$ configuration implies that the graph is not Berge, we will need the following Roussel-Rubio lemma:

Lemma 1.2.2 (Roussel-Rubio Lemma [**RR01**, **MC05**])**.** *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.*

We may use this lemma quite often, might want to provide proof if so.

Now, we shall prove the following:

Theorem 1.2.3. *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 1.2.2, 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 1.2.2 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: $v_2, x, p_2, \dots, p_{n-1}, y, v_2$.

□

I merged a couple of proofs from [**MC06**], check in the morning if this is correct.

check in the morning

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.

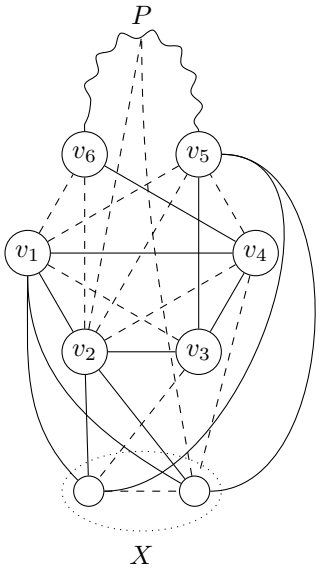


Figure 1.4: An example of a $\mathcal{T}_3$.

The following algorithm with running time of $O(|V|^6)$ checks whether G contains a configuration of type $\mathcal{T}_3$:

Algorithm 1.2.2
Input: A graph G.
Output: Returns whether G contains a configuration of type 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 nonneibhbor 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                \\ v_1, ... v_6, P, X is a T_3
15:  |  |  |  |  |  |  end if
16:  |  |  |  |  |  end for
17:  |  |  |  |  end if
18:  |  |  |  end for
19:  |  |  end for
20:  |  end for
21:  |  return FALSE
22: end procedure
```

We will skip the proof that each graph containing $\mathcal{T}_3$ is not Berge, as it is quite technical. See section 6.7 of [**MC05**] for the proof.

Theorem 1.2.4.

To see that the algorithm 1.2.2 has a running time of $O(|V|^6)$, let us note that for each triple v_1, v_2, v_5 we examine, of which there are $O(|V|^3)$, there are linear many choices of X, each taking $O(|V|^2)$ time to process and generating a linear many choices of v_4 which take a linear time to process in turn. This gives us the total running time of $O(|V|^6)$.

.

1.2.2 Amenable holes.

First, let us introduce a few new definitions.

Definition 1.3 (C-major vertices)**.** *Given a shortest odd hole C in G, a node $v \in V(G) \setminus V(C)$ is C-major if the set of its neighbors in C is not contained in any 3-node path of C.*

a picture of this, clean odd hole, amenable hole

Definition 1.4 (clean odd hole)**.** *An odd hole C of G is* clean *if no vertex in G is C-major.*

Definition 1.5 (cleaner)**.** *Given a shortest odd hole C in G, a subset $X \subseteq V(G)$ is a* cleaner for C *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 1.6 (near-cleaner)**.** *Given a shortest odd hole C in G, a subset $X \subseteq V(G)$ is a* near-cleaner for C *if X contains all C-major vertices and $X \cap V(C)$ is a subset of vertex set of some 3-node path of C.*

Definition 1.7 (amenable odd hole)**.** *An odd hole C of G is* amenable *if it is a shortest off hole in G, it is of length at least 7 and for every anticonnected set X of C-major vertices there is a X-complete edge in C.*

Theorem 1.2.5. *Let G be a graph, such that G and $\overline{G}$ contain no Pyramid, no Jewel and no configuration of types $\mathcal{T}_1, \mathcal{T}_2$ or $\mathcal{T}_3$. Then every shortest hole in G is amenable.*

The proof of this theorem is quite technical and we will not discuss it here. See section 8 of [**MC05**] for the proof.

With Theorem 1.2.5 we can describe the rest of the algorithm.

Algorithm 1.2.3 (List possible near cleaners, 9.2 of [**MC05**])
Input: A graph G.
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.

Begin.
We will call a triple (a, b, c) of vertices *relevant* if $a \neq b$ (but possibly $c \in \{a, b\}$) and $G[\{a, b, c\}]$ is an independent set.

Given a relevant triple (a, b, c) we can compute the following:

- $r(a, b, c) \leftarrow$ the cardinality of the largest anticomponent of $N(a, b)$, that contains a nonneibhbor of c, or 0, if c is $N(a, b)$-complete.
- $Y(a, b, c) \leftarrow$ the union of all anticomponents of $N(a, b)$ that have cardinality strictly greater than $r(a, b, c)$.
- $W(a, b, c) \leftarrow$ the anticomponent of $N(a, b) \cup \{c\}$ that contains c.
- $Z(a, b, c) \leftarrow$ the set of all $Y(a, b, c) \cup W(a, b, c)$-complete vertices.
- $X(a, b, c) \leftarrow Y(a, b, c) \cup Z(a, b, c)$.

For every two adjacent vertices u, v compute the set $N(u, v)$ and list all such sets. For each relevant triple (a, b, c) compute the set $X(a, b, c)$ and list all such sets.

Output all subsets of $V(G)$ that are the union of a set from the first list and a set from the second list. *End*

To prove the correctness of algorithm 1.2.3 we will need the following theorem.

Theorem 1.2.6 (9.1 of [**MC05**])**.** *Let C be a shortest odd hole in G, with length at least 7. 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.*

Let us suppose that C is an amenable hole in G. By theorem 1.2.6, 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 1.2.6 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 $N(u, v) \cup X(a, b, c)$ is a near-cleaner for C. Therefore the output of the algorithm 1.2.3 is correct.

Algorithm 1.2.4 (Test possible near cleaner, 5.1 of [**MC05**])
Input: A graph G containing no simple forbidden structure, 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.*

Begin.
For every pair $x, y \in V(G)$ of distinct vertices find shortest path $R(x, y)$ between x, y with no internal vertex in X. If there is one, let $r(x, y)$ be its length, if not, let $r(x, y) = \infty$.

For all $y_1 \in V(G) \setminus X$ and all 3-vertex paths $x_1 - x_2 - x_3$ of $G \setminus y_1$ we check the following:

- $R(x_1, y_1), R(x_2, y_2)$ both exist – define y_2 as the neighbor of y_1 in $R(x_2, y_1)$.
- $r(x_2, y_1) = r(x_1, y_1) + 1 = r(x_1, y_2)$ $(= n$ say$)$
- $r(x_3, y_1), r(x_3, y_2) \geq n$

moze raczej $min(r(x_3, y_1), r(x_3, y_2)) = r(x_2, y_1) = r(x_1, y_2) = r(x_1, y_1) + 1$?

If there is such a choice of x_1, x_2, x_3, y_1 then we output that there is an odd hole. If not, we report that there is no shortest odd hole C such that X is a near-cleaner for C. *End*

Below we will prove that if the algorithm 1.2.4 reports an odd hole in G, there indeed is one. The proof of the correctness of the other possible result is more complicated, see section 4 and theorem 5.1 of [**MC05**].

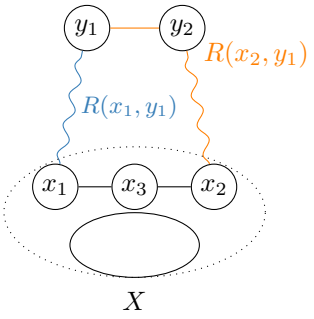


Figure 1.5: An odd hole is found

Proof. Let us suppose that there is a choice of x_1, x_2, x_3, y_1 satisfying the three conditions in the algorithm 1.2.4 and let y_2 and n be defined as in there. We claim that G contains an odd hole.

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

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 $r(x_3, y_1), r(x_3, y_2) \geq n (= r(x_1, y_2))$ 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 $r(x_1, y_2) = n$ 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 $2 \leq i \leq n-1$ and $2 \leq j \leq n$ 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, that is $j = i + 1$. So $p_1 - \ldots - p_1 - q_{j+1} - \ldots - q_n$ contains a path between x_1, y_2 of length $\leq n-2$, contradicting that $r(x_1, y_2) = n$. So $R(x_1, y_1)$ and $R(x_2, y_1)$ have no common vertex 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 $x_1 - x_3 - x_2$ form an odd hole, so the answer is correct.

Suppose that $p_i q_j$ is an edge for some $1 \leq i \leq n-1$ and $1 \leq j \leq n$. 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 $\{p_1, \ldots, p_i, q_j, \ldots q_n\}$ which has length $\leq n - j + 1$ and has no internal vertex in X (since $j > 1$); and since $r(x_1, y_2) = n$, it follows that $n - j + i \geq n$, that is, $i \geq j$ as claimed.

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

1.2.3 Summary

todo?

Chapter 2

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 [**Grtschel1993**]. We consider it in section 2.3. However due to its use of the ellipsoid method this algorithm has been usually considered unpractical [**coloringSquareFree**, **Chudnovsky2003**, **coloringArtemis**].

There has been much progress on the quest of finding a more classical algorithm coloring perfect graphs, without the use of ellipsoid method (see section 2.4), however there is still no known polynomial combinatorial algorithm to do this.

better wording of this paragraph

2.1 Information theory background

Section 2.1 and section 2.2 are based on lecture notes by Lovász [**Lovasz95**].

The 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 semi-definite matrices. The connection of coloring graphs and the cones of semi-definite 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 [**Chudnovsky2003**].

2.1.1 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

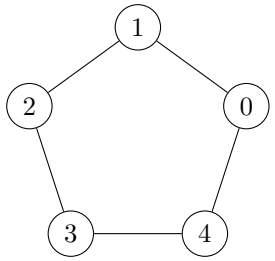


Figure 2.1: An example of a noisy channel

C_5 (fig. 2.1) in which vertices correspond to signal values and two vertices are connected if and only if 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.

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 (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 2.3.1) we could have a lower bound on $\Theta(G)$. Let us now turn into search for some usable upper bound.

2.1.2 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 $\alpha(C_5^2)$).

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, that is 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 nodes are represented with orthogonal vectors. We can calculate $e_1^\intercal u_i = 5^{-1/4}$.

picture

It turns out, that we can obtain a similar representation of the nodes of C_5^k by unit vectors $v_i \in \mathbb{R}^{3k}$, so that any two non-adjacent nodes are labeled

with orthogonal vectors (this representation is sometimes called the *orthogonal representation* [**Lovsz1989Orthogonal**]). 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 get

why?

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

(if v_i formed a basis then this inequality would be an equality).

On the other hand each term on the left hand side is $5^{-1/4}$, 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 find a orthogonal representation that will give us the best bound. So, we can define the *Lovász number* of a graph G as :

this equation does not really follow from the thought process above, it is a slightly different definition

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

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

Contrary to Lovász's first hope [**Lovasz1979**] $\vartheta(G)$ does not always equal $\Theta(G)$, 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 2.1.1 (Lovász "sandwich theorem" [**Knuth1994**])**.** *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 good) orthogonal representation? It turns out that it can be computed in polynomial time using semidefinite optimization.

2.2 Computing ϑ

First, let us recall some definitions, with [**gilbertstrang2020**] as a reference for linear algebra.

Definition 2.1 (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 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.

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

Definition 2.2 (positive semidefinite matrix)**.** *Let A be an $n \times n$ symmetric matrix. A is* positive semidefinite *if all of its eigenvalues are nonnegative. We denote it by $A \succeq 0$.*

We have equivalent definitions of semidefinite matrices.

Theorem 2.2.1. *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 2.3 (convex cone)**.** *A subset C of $\mathbb{R}^n$ is a* convex cone*, 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 vertex 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.*

The special case when $A_1, \ldots A_n, B$ 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 [**grotschel1993**] for reference. 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 nonedges ij. 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 2.2.2 (Proposition 5.1 of [**Lovasz95**]). *The minimum angle ϕ of any orthogonal representation of G is given by $\cos^2 \phi = 1/\vartheta(G)$.*

mark all theorems everywhere accordingly

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

Theorem 2.2.3 (Proposition 5.3 of [**Lovasz95**]). *$\vartheta(G)$ 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 sual 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.

2.3 Coloring perfect graph using ellipsoid method

The following is based on **Laurent2005** by **Laurent2005** [**Laurent2005**].

2.3.1 Maximum cardinality stable set

Given graph G, recall that stability number of G is equal 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 $\alpha(G)$.

Algorithm 2.3.1 (maximum cardinality stable set in a perfect graph)
Input: A perfect graph $G = (V, E)$.
Output: A maximum cardinality stable set in G.

Begin.
Let $v_1, \dots v_n$ be an ordering of vertices of G. We will construct a sequence of induced subgraphs $G = G_0 \supseteq G_1 \supseteq \dots \supseteq G_n$, so that G_n is a required stable set.

Let $G_0 \leftarrow G$. Then, for each $i \geq 1$, compute $\alpha(G_{i-1} \setminus v_i)$. If $\alpha(G_{i-1} \setminus v_i) = \alpha(G)$, then set $G_i \leftarrow G_{i-1} \setminus \{v_i\}$, else set $G_i \leftarrow G_{i-1}$.

Return G_n. *End*

Let us prove that G_n is indeed a stable set. Suppose otherwise and let $v_i v_j$ be an edge in G_n 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 $v_i v_j$ is not an edge of G_n. Therefore there are no edges in G_n.

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

The running time of algorithm 2.3.1 is polynomial, because we construct n auxiliary graphs, each requiring calculating α once plus 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 in the following manner. Create graph G' by replacing every node v by a set W_v of $w(v)$ nonadjacent nodes, making two nodes $x \in W_v$, $y \in W_u$ adjacent in G' if and only if the nodes v, u are adjacent in G. Then 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. We will use this technique later on.

2.3.2 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.

Algorithm 2.3.2
Input: A perfect graph $G = (V, E)$.
Output: A stable set which intersects all the maximum cardinality cliques of G.

Begin.
We will create a list $Q_1, \dots Q_t$ of all maximum cardinality cliques of G.

Let $Q_1 \leftarrow$ a maximum cardinality clique of G. We calculate this by running algorithm 2.3.1 on $\overline{G}$.

Now suppose $Q_1, \dots, Q_t$ have been found. We show how to calculate Q_{t+1} or see that we are done.

Let us define a weight function $w : V \to \mathbb{N}$, so that for $v \in V$, $w(v)$ is equal to the number of cliques $Q_1, \dots Q_t$ that contain v.

Assign $S \leftarrow$ the maximum w-weighted stable set, as described in a remark to algorithm 2.3.1. It is easy to see that S has weight t, which means that S meets each of $Q_1, \dots Q_t$.

If $\omega(G \setminus S) < \omega(G)$, then S meets all the maximum cardinality cliques in G so we return S. Otherwise we find a maximum cardinality clique in $G \setminus S$ (it will be of size $\omega(G)$, because $\omega(G \setminus S) = \omega(G)$), add it to our list as Q_{t+1} and continue with longer list.

End

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 2.3.1 on it. Therefore total running time is polynomial.

2.3.3 Minimum coloring

Algorithm 2.3.3
Input: A perfect graph $G = (V, E)$.
Output: A coloring of G using $\chi(G)$ colors.

Begin.
If G is equal to its maximum cardinality stable set, color all vertices one color and return.

Else find S intersecting all maximum cardinality cliques of G (algorithm 2.3.2). Color recursively all vertices of $G \setminus S$ with $\chi(G \setminus S) = \omega(G \setminus S) = \omega(G) - 1$ colors and all vertices of S with one additional color. *End*

We will call recursion at most $O(|V|)$ times, each step of recursion is polynomial in time. Therefore the running time of algorithm 2.3.3 is polynomial.

2.4 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.

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 hole, all three paths in a prism must have the same parity. A prism with all three paths odd is called an *odd prism*.

In 2005 Maffray and Trotignon a coloring algorithm that colors graphs containing no odd hole, no antihole and no prism (sometimes called Artemis graphs) in $O(|V|^4|E|)$ time [**Maffray2006**]. They later improved the time complexity to $O(|V|^2|E|)$ [**Lvque2009**].

In 2015 Maffray showed an algorithm for coloring Berge graphs with no squares (a square is a C_4) and no odd prism [**Maff2015**].

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 [**Chudnovsky2017**].

A most recent advancement (2018) is an algorithm by Chudnovsky et al. that colors any square-free Berge graphs in time of $O(|V|^9)$ [**Chudnovsky2019**]. Before proving strong perfect graph conjecture, a similar conjecture for square-free Berge graphs has been proven by Conforti et al. [**Conforti2004**] During one of her lectures, Maria Chudnovsky expressed hope that discovery of full algorithm for coloring Berge graphs might follow a similar pattern. We analyze this algorithm and provide its pseudocode in appendix A.

Dabym ze 2-3 zdania braggowania, ze algorytm jest duzo bardziej zlozony niz rozpoznawanie + na czym sie opiera.

Chapter 3

Implementation

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

make repo public and add readme on howto run, some script creating and running default tests

3.1 Berge graphs recognition: naïve approach

When implementing an algorithm with the running time of $O(n^9)$ (especially one as complex as CCLSV), one of the first questions that pop up is whether this algorithm is applicable at all. 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.

well.. there's this java impl

We will 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 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 last vertex and continue. Also notice, that we can abandon "paths" that have chords.

This backtracking algorithm is very simple, but has a great potential for optimizations in implementation. We also use path enumeration algorithm in 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 chord

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 to choose from. We could simply enumerate all sequences of vertices without repetition and for each one of them check if it is a path (that is if all pairs of vertices next to each other are connected and that there are no chords), but this would be too slow and render our algorithm unusable.

We notice that returning a new path for each call of the 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 vertices unique and no chords). 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 better data structure. For a given graph we create a data structure that suits best the goal of generating paths. 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 vertices a, b if a and b are connected, 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 data structure design gave us a speedup of overall running time of our algorithm in the range of 20x. check this

Another crucial optimization is to eliminate partial solutions that have chords right away. This is especially useful in graphs that have many chords and reduces running time dramatically.

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 a 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 a 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 the $areUnique$ method we increment $counter$.

This optimization alone is crucial for performance. In our testing, all calls of $areUnique$ took abour 70% of total running time of the CCLSV algorithm, and virtually all running time of the naïve algorithm . After using static $stamp$ array, it fell to 5.2% in CCLSV and to X% in naïve. The overall speedup of the CCLSV algorithm was about 6x. The speedup of naïve algorithm was even greater – about 20x.

check

check if for sure it didn't use path speedup

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 generate all possibly chordal paths or holes and therefore has much use in the CCLSV algorithm. For example, when searching for jewels we generate with it all possibly chordal paths of length 5 and for each check if it has required properties of a jewel. This proved to be much faster than generating vertices that don't have forbidden chords one by one. We use the same algorithm for generating starting vertices for a possible $\mathcal{T}_2$ and $\mathcal{T}_3$.

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 3.4 for running times.

3.2 Berge graphs recognition: polynomial algorithm

rewrite/rearrange text below

The Berge recognition algorithm's running time is $O(n^9)$, 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 (section 3.2.1) make a very big difference and make it more viable than naïve algorithm, at least on some test cases. Also, we explore a new frontier of implementing its most time consuming part on massively parallel GPU architecture, with some good results (section 3.3).

Anything interesting about algo/data structure?

3.2.1 Optimizations

When implementing a complicated algorithm that has a time complexity of $O(n^9)$ 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 lest 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* [**callgrind**].

Callgrind is a profiling tool that records the call history of a program and presents it as a call-graph. When profiling a program, event counts (data reads, cache misses etc.) are attributed directly to the function that they occurred in. In addition to that, the costs are propagated to functions upwards the call stack. For example, say internal function *worker* consumes most of programs running time. It is called from *run*1 and *run*2, with *run*2 calls taking twice as much time. *run*1 and *run*2 are in turn called from main. Total attribution of *worker* would be nearly 100%, of *run*1 about 33% and of *run*2 about 66%. The contribution of main would also be near 100%. We used a tool *gprof2dot* [**gprof2dot**] to generate visual call graphs from callgrind's output.

example of a call graph?

callgrind

first result to best result gains (like: we are 20x faster than what first comes to mind)

regenerate speedup results to make sure what the gains were

Before any optimizatios, 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 hole by itself, find a simple structure or check a possible near cleaner for amenable odd hole. Therefore optimizations of the path enumerating algorithm were also helpful in speeding up CCLSV implementation.

Another algorithm for which we found major speedups is the algorithm to generate all possible near-cleaners (Algorithm 1.2.3). At its very end, it returns a set that is an union . We used a `dynamic_bitset` from the boost library [**boost**] for this purpose. 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. By using it, the speedup of Algorithm 1.2.3 was about x.

after rewriting alg, point to a line

After these optimizations, checking each potential near-cleaner (Algorithm 1.2.4) is by far the biggest bottlenecks of the CCLSV algorithm. We didn't find any major optimizations there, but Algorithm 1.2.4 has a good potential for parralelization, which we explore in Section 3.3.

todo

3.2.2 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 3.1). 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 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 3.3).

Also, we test the algorithm 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 [**graphRepo**] as a source of perfect graphs.

update to 11?

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

In total over 100 cpu-hours of end to end tests were performed without any errors.

update the munber

3.3 Parallelism with CUDA

CUDA background

moderngpu - allows us to run simple transforms

Implementing get NC is tough – we rely on set¡bitset¿ to do the work.
Maybe we could use a dynamic set from Adam's link? As a "open question"

We posed a question if it would be profitable to use GPU for this problem. 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 the algorithm is complex and not easy to parallelise. Therefore we decided to analyze the CCLSV algorithm and parrallelize parts that would benefit from it the most.

a good name for it?

To identify parts of CCLSV to implement, we used callgrind and found potential bottlenecks. The initial tests showed that with growing size of a graph some parts take more and more relative time. Because of callgrind's slow execution times compared to just running the program, we used manual timers for bigger tests. Using this method we identified testing all possible near-cleaners (algorithm 1.2.4) as the biggest bottleneck for larger graphs. We

considered two approaches: run whole algorithm 1.2.4 on a single CUDA core, testing multiple Xs in parallel, or parallelize algorithm 1.2.4 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 1.2.4. 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 parallelise the work after calculating R. Let us notice, that for all Xs all 3-vertex paths are the same. We calculate them beforehand, then each thread receives a 3-vertex path $x_1 - x_2 - x_3$ and an additional vertex y_1 and performs the required checks. This is almost perfect scenario for the GPU – we do a simple SIMD work in parallel, without a lot scattered memory access.

It turns out that this optimization alone speeds up algorithm 1.2.4 almost 7x on bigger tests, which gives us a speedup of about 6x for overall algorithm.

check

implement other CUDA opts and describe them

check

3.4 Experiments

Data sets

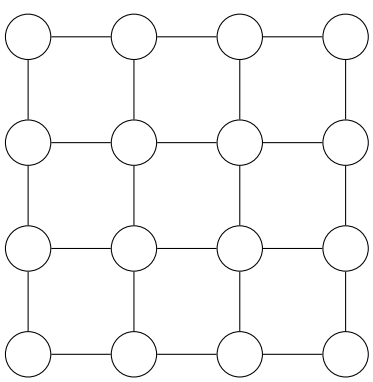
Let us describe experiments and their results. All of our algorithms search for an odd hole or an odd antihole and stop when find one, or a evidence of one. Therefore, their running times are greatest with perfect graphs on the input and we decided to use mainly perfect graphs for our performance benchmarks. For every test, the vertex number were shuffled.

run some smart non-perfect tests

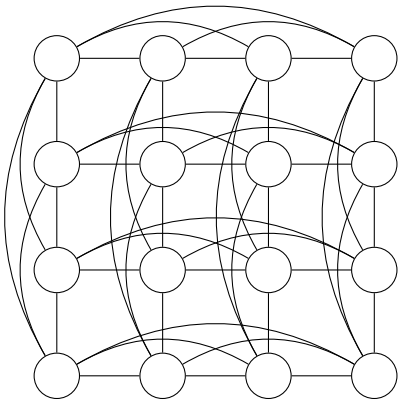
We ran tests on X classes of perfect graphs, see Figure 3.1 for some examples:

- Random perfect graphs,
- Random bipartite graphs,
- Line graphs of random bipartite graphs,
- Lattice graphs (fig. 3.1a),
- Rook graphs (fig. 3.1b),
- Knight graphs (fig. 3.1c),
- Hypercube graphs (fig. 3.1d),
- Split graphs (fig. 3.1e).

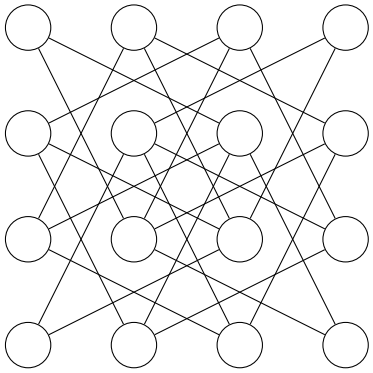
In later paragraphs we describe each class one by one and benchmark our implementations on them.



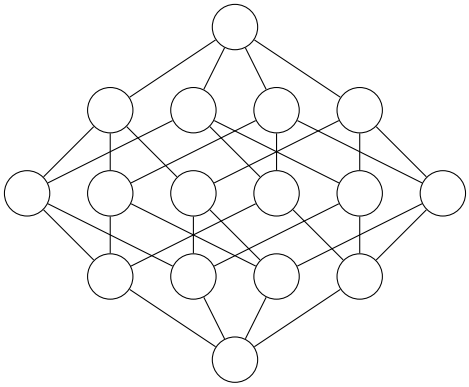
(a) Lattice graph



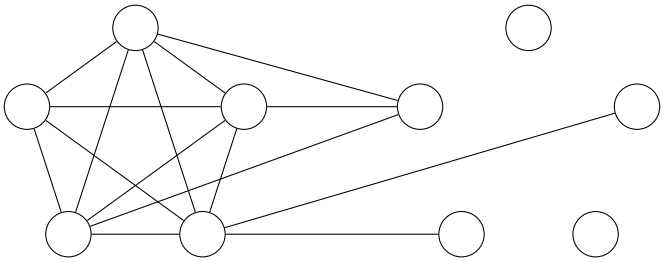
(b) Rook graph



(c) Knight's graph



(d) Hypercube, $n = 16$



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

Figure 3.1: Graph classes

Random perfect graphs Our first method of generating graphs is to set $|V|$ and for each pair of vertices u, v let uv be the edge with probability of $1/2$. Then we check if generated graph is perfect and continue as long as we don't get sufficient number of perfect graphs.

First, we note that this method of generating perfect graphs is very inefficient and we couldn't generate any graphs with $|V| \geq 20$. Second, those graphs favour the naïve algortihm, because there is very low probability of long chordless paths to appear (each chord has a $1/2$ chance to appear).

When looking at the results (Figure 3.2a), while naïve algorithm's time is almost zero, we see the CCLSV running time climbing with the growth of $|V|$, with only moderate improvement by GPU CCLSV. Slight GPU improvement is explained by anylysis of the components of the overall time (Figure 3.2b). Testing all near cleaners takes up around half of total time, so we cannot speed up the overall time by much. These are also relatively small tests, so the latency from copying data to the GPU is significant.

Better plot by a lot WIP

Random bipartite graphs

TODO

Line graphs of random bipartite graphs

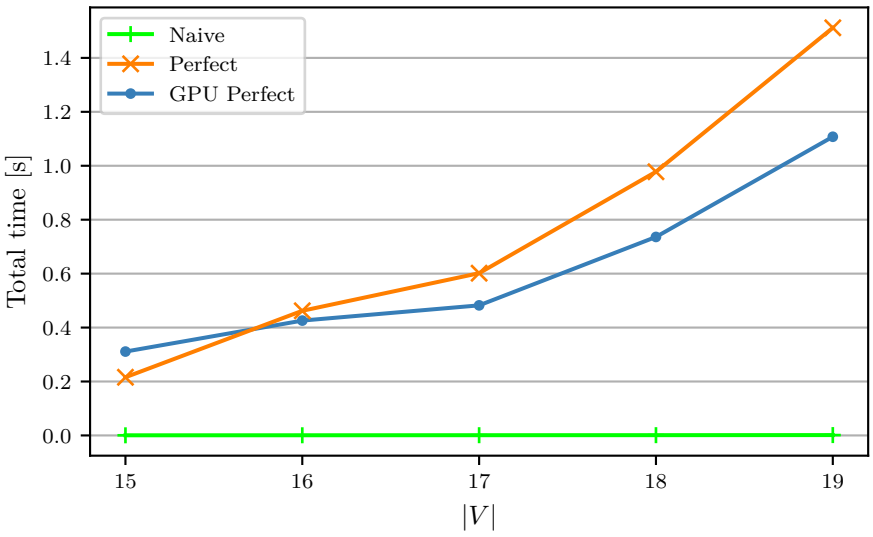
data with bigger N

Next, we generated a random bipartite graphs (see paragraph above) and then calculated their line graphs. We repeated the process until we had sufficieng number of graphs for each $|V|$. On fig. 3.3a we can see GPU gives us much better improvement than in random graphs. This is due to the fact, that testing near cleaners takes over 80% of the time for bigger graphs (Figure 3.3b).

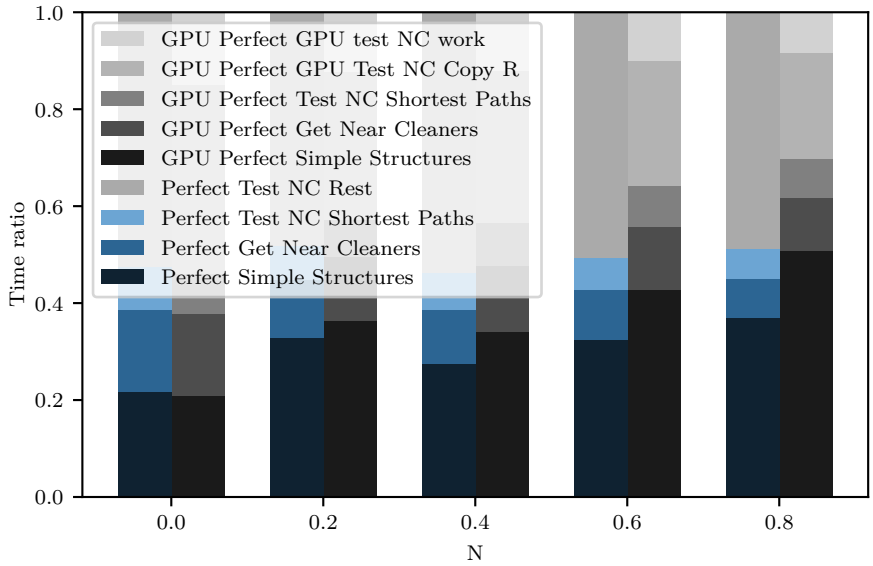
Lattice graphs Next, we turn our attention to graphs generated on a grid, or a checker board (see later paragraphs). First, we take lattice graphs. In lattice graph, each vertex is connected to vertices that are above, below, to the left and to the right of it, if such vertices exist. See fig. 3.1a for an example of a 4×4 lattice graph.

Let us take a look at fig. 3.4a. It is a clear example of why polynomial algorithms can be useful, even the ones with time complexity as high as CCLSV. For $|V| = 48$ the running time for the CCLSV is 22.2s, for GPU CCLSV is 7.7s and for naïve is 1.5s. Naïve is almost 15x faster. But with growing $|V|$ its running time grows very fast, reaching 41s for $|V| = 60$ and 210s for $|V| = 66$. CCLSV growth is much gentler and for $|V| = 66$ it is 2.5x faster than naïve.

With larger $|V|$ the speedup gained from utilizing GPU is also more significant. For $|V| = 78$ the running time for the whole algorithm is almost 3x better.

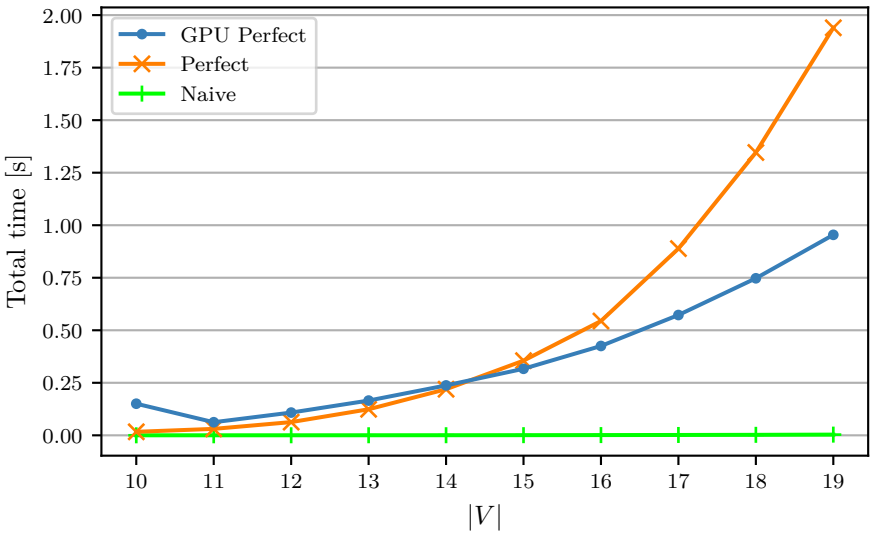


(a) Random perfect graphs

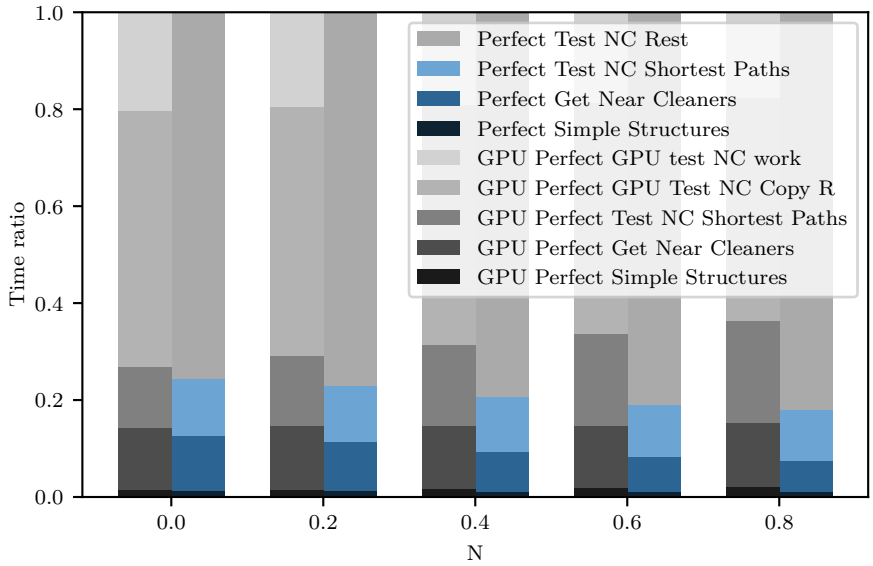


(b) Random perfect graphs

Figure 3.2: Random perfect graphs

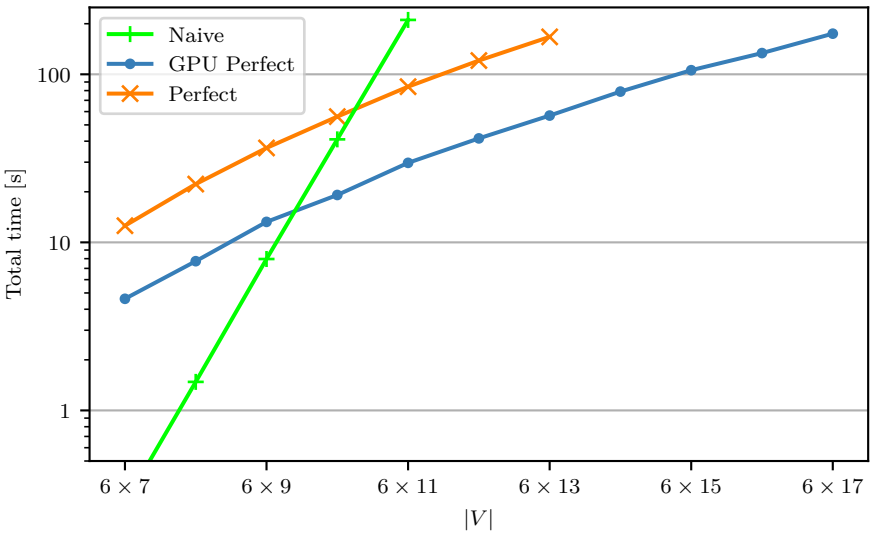


(a) Line graphs of random bipartite graphs

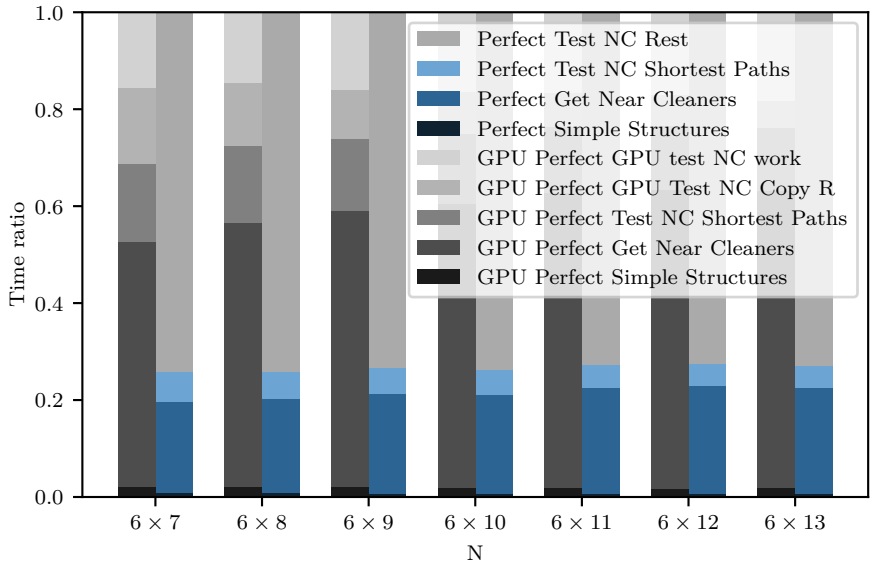


(b) Line graphs of random bipartite graphs

Figure 3.3: Line graphs of random bipartite graphs



(a) Overall running times



(b) Percentage of time taken be each part

Figure 3.4: Lattice graphs

Rook graphs If we imagine our grid to be a chessboard we can define a Rook graph to have an edge, when the rook from chess can move from one vertex to another in one move. See fig. 3.1b for an example of 4×4 Rook graph.

Rook graphs have many more chords than Lattice graphs, so the naïve's performance is much better here. What is more interesting, the overall performance of CCLSV is much worse (fig. 3.5a). Looking at fig. 3.5b we can see that the majority of CCLSV time is spent checking all near cleaners. This can be explained by the fact that rook graphs are much denser than lattice graphs and have many more 3-vertex paths (see line TODO of algorithm 1.2.4). This is confirmed by greater CCLSV GPU speedup of around 6x on graphs with $|V| = 35$.

Knight graph A knight graph is constructed in a similar way to rook graph, but by taking potential moves of a knight.

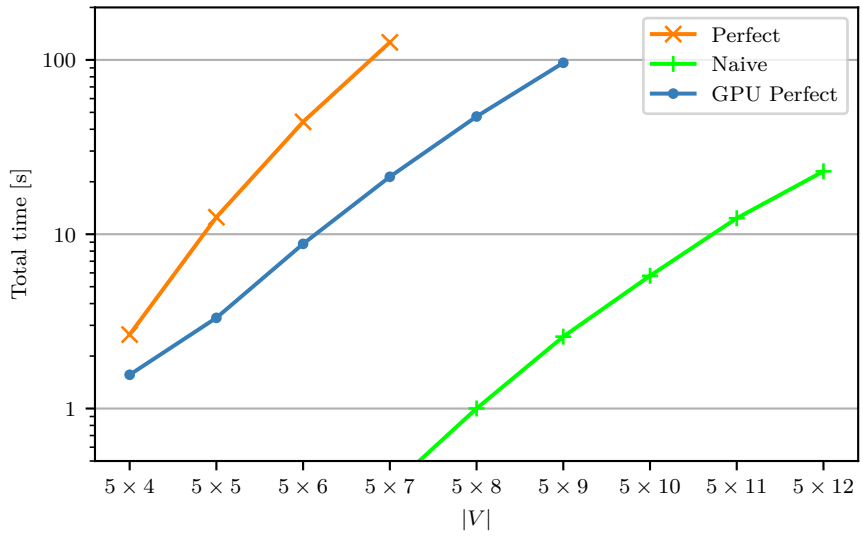
add 800s result for naive

Hypercube graphs 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.

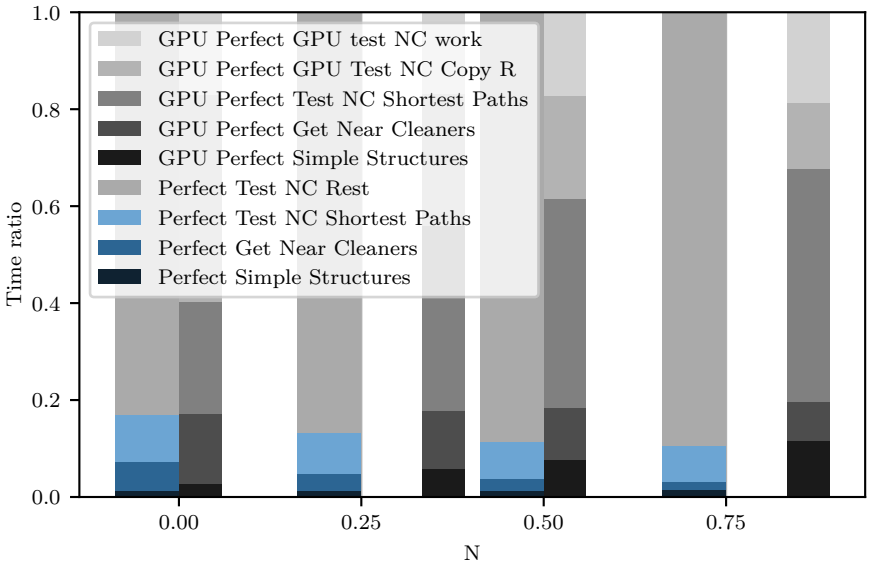
Hypercube graphs tell a similar story to lattice graphs and knight graphs. Naïve is very fast at first, but becomes unbearably slow for $|V|$ higher than 55. Unfortunately, CCLSV doesn't fare much better, although it grows slower. Again, the speedup of GPU CCLSV is noticeable, being around 4.8x for $|V| = 55$.

Split graphs 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 $1/2$ chance to appear.

There are no long chordless paths here, so naïve fares great (fig. 3.8a). What is interesting, when looking at detailed times of GPU CCLSV, we can see that checking the existence of simple forbidden structures takes more than 30% of the time. This opens a possibility of further GPU speedups, not explored by us.

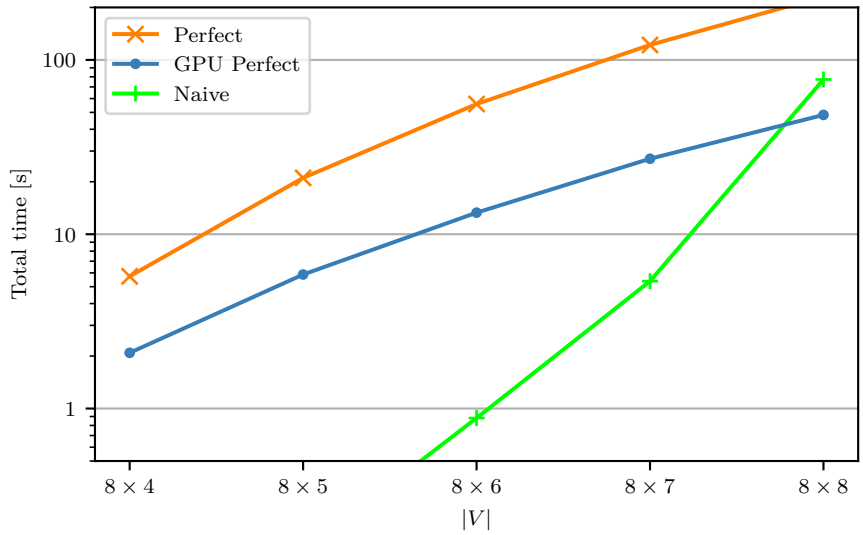


(a) Overall running times

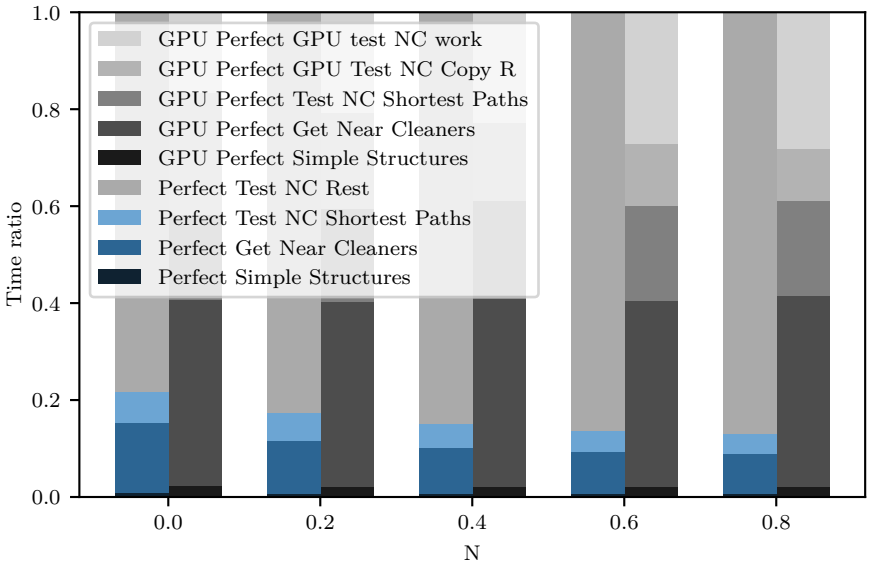


(b) Percentage of time taken be each part

Figure 3.5: Rook graphs

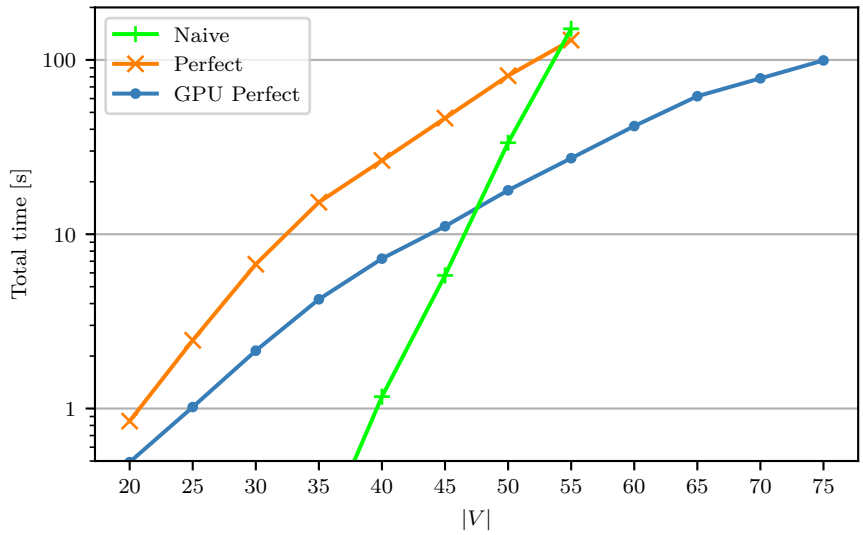


(a) Overall running times

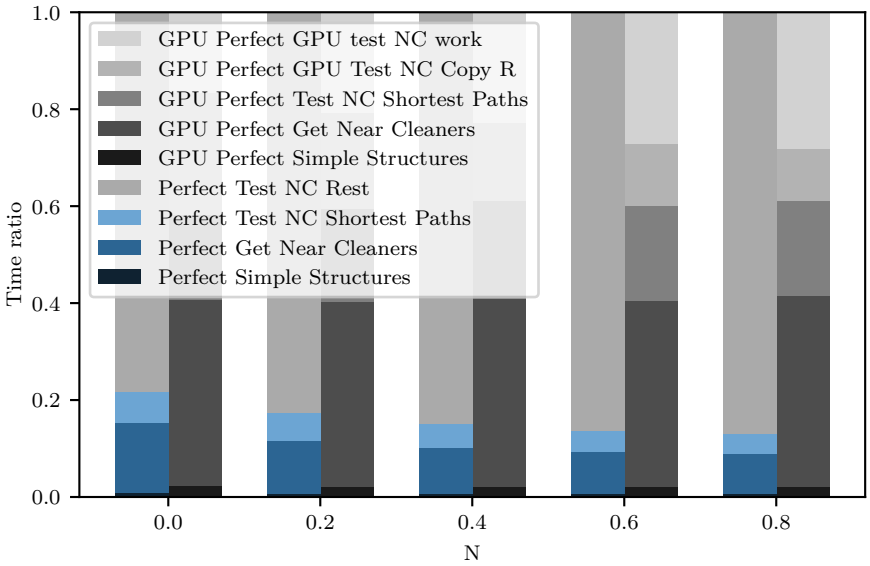


(b) Percentage of time taken be each part

Figure 3.6: Knight graphs

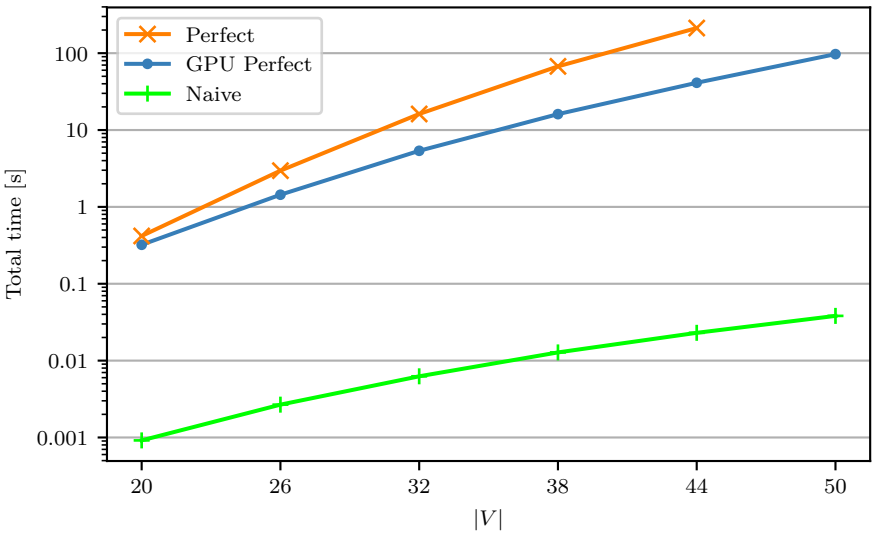


(a) Overall running times

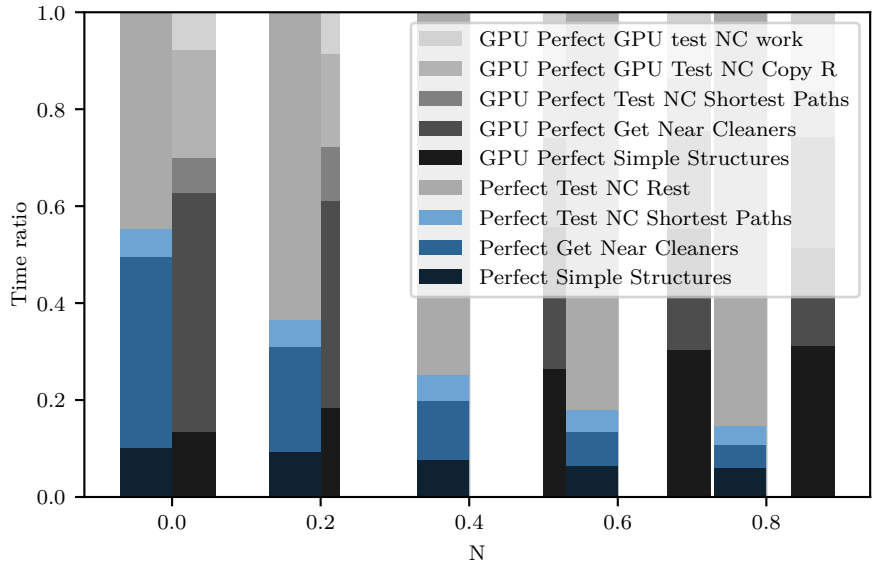


(b) Percentage of time taken be each part

Figure 3.7: Hypercube graphs

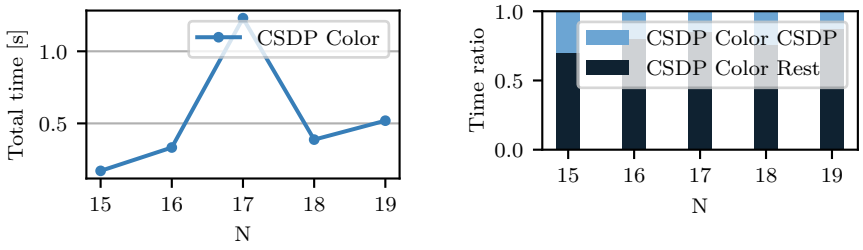


(a) Overall running times



(b) Percentage of time taken be each part

Figure 3.8: Split graphs



(a) Overall running times (b) Percentage of time taken be each part

Figure 3.9: Random perfect graphs

TODO

Figure 3.10: Line graphs of random bipartite graphs

3.5 Coloring Berge Graphs

3.5.1 Ellipsoid method

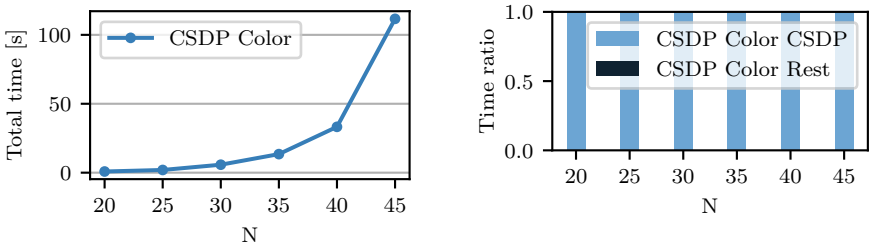
We used an open source CSDP [**csdpRepo**, **csdp1999**] library, that implements predictor corrector variant of the semidefinite programming algorithm to calculate $\vartheta(G)$, given G on the input. The CSDP library has been used in many recent publications across different fields, such as [**Ampountolas˙2017**, **Adasme˙2011**].

setting $\epsilon = 1/2$ in csdp helped a lot – how much?

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 2.3. 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 specifying the precision of ϑ we want to calculate to be 1/3 sped up the algorithm by TODO. 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.

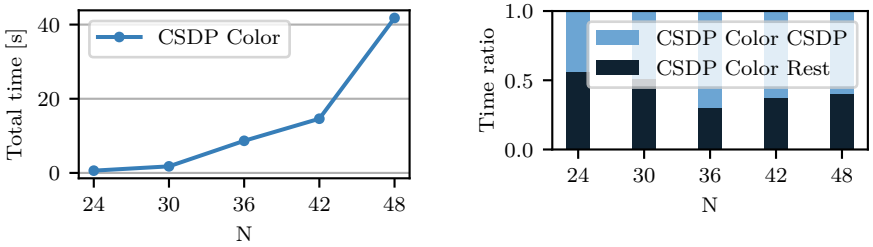
Experiments and results

A few things of note. We notice that in rook graphs the graph manipulation other than calculating CSDP takes close to half of overall running time (fig. 3.12b). We didn't particularly optimize this, other than by use of methods developed for Berge graph recognition algorithm. There are surely some optimizations to be made here if need be. Altough, as we can see on other plots, it won't help in majority of the cases.



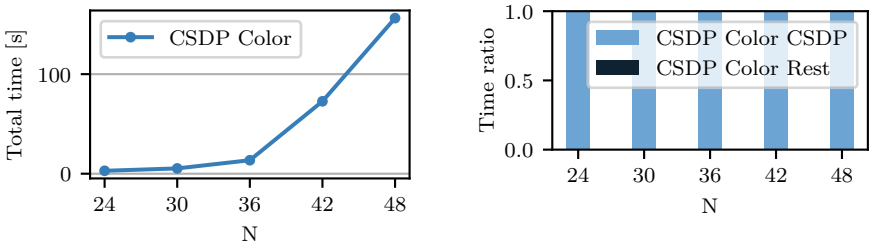
(a) Overall running times (b) Percentage of time taken be each part

Figure 3.11: Lattice graphs



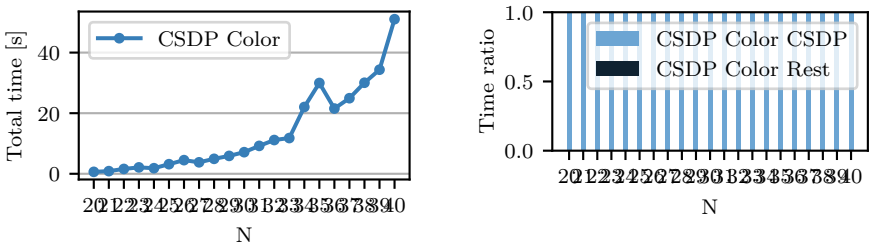
(a) Overall running times (b) Percentage of time taken be each part

Figure 3.12: Rook graphs



(a) Overall running times (b) Percentage of time taken be each part

Figure 3.13: Knight graphs



(a) Overall running times (b) Percentage of time taken be each part

Figure 3.14: Hypercube graphs

TODOa

Figure 3.15: Split graphs

Appendices

Appendix A

Perfect Graph Coloring algorithm

A.1 Notes

In appendix 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.

- $V(X)$ – vertices of structure X. Will be written as X when obvious.
- $a - b$, when a and b are nodes – a and b are neighbors.
- $a \cdots b$, when a and b are nodes – a and b are not neighbors.
- $a - X$, when a is a node and X is a set of nodes – a has a neighbor in X.
- $a \cdots X$, when a is a node and X is a set of nodes – a has a nonneighbor in X.
- $a \blacktriangleleft X$, when a is a node and X is a set of nodes – a is complete to X.
- $a \not\blacktriangleleft X$, when a is a node and X is a set of nodes – a is anticomplete to X.
- $X \blacksquare Y$, when X and Y are set of nodes – X is complete to Y.
- $X \not\blacksquare Y$, when X and Y are set of nodes – X is anticomplete to Y.
- $[n]$ – $\{1, \ldots, n\}$.
- $L(BS(K_4))$– a line-graph of a biparite subdivision of K_4.
- $a \leftarrow b$ – let a be equal b.
- $a :\in X$ – let a be equal to any element of X

- $a \veebar b$ – a xor b
- Strip S_{uv} is *rich* if and only if $S_{uv} \setminus T_{uv} \neq \emptyset$

Introduction on unknown structures - J-strip etc etc

A.2 Algorithms

Algorithm A.2.1 (Color Graph)

COLOR-GRAPH (G)
Input: G – square-free Berge graph
Output: A $\chi(G)$-coloring of G
1 TODO
2 TODO
3 **return** Todo

---

aaa

Algorithm A.2.2 (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 $c_1(w) = 1, c_2(w) = 2$
  $P \leftarrow$ chordless path $w - p_1 - \ldots - p_k - a$ in $C_1^{1,2}(w)$ so that
    $k \geq 1$, $p_1 \in K_3 \cup L$, $p_2 \ldots p_k \in L$, $a \in K$, $c_1(a) \in [2]$
  $Q \leftarrow$ chordless 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   // ∄u ∈ B, h ∈ [2] : C_h^{c_1(u),c_2(u)}(u) ∩ K_1 = ∅
24  if p_1 ∈ K_3 then
        ASSERT: c_2(p_1) ∉ [2]
25      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) ∩ K_1 = ∅
26      c'_2 ← c_2 with colors j and 3 swapped in C_2^{j,3}(p_1)
        ASSERT: j = 2
27      return Color-Good-Partition(G, K_1, K_2, K_3, L, R, c_1, c'_2)
28  else
29      relabel c_1, so that c_1(q_1) = 3
30      if 3 does not appear in K_1 then
            ASSERT: C_1^{j,3}(q_1) ∩ K_1 = ∅
            ASSERT: j = 1
31          c'_1 ← c_1 with colors j and 3 swapped in C_1^{j,3}(q_1)
32          return Color-Good-Partition(G, K_1, K_2, K_3, L, R, c'_1,
              c_2)
33      else
            ASSERT: q_1 ⋪ {a, a_3}
            ASSERT: C_1^{i,3}(q_1) ∩ K_1 = ∅
            ASSERT: i = 1
34          c'_1 ← c_1 with colors i and 3 swapped in C_1^{i,3}(q_1)
35          return Color-Good-Partition(G, K_1, K_2, K_3, L, R, c'_1,
              c_2)
```

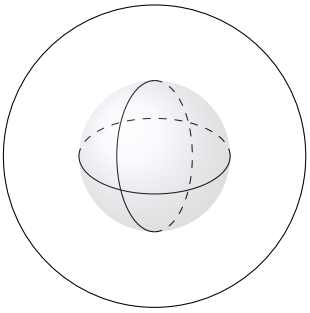


Figure A.1: 3d test