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

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*. The size of the vertex set of a graph G will be called the *cardinality* of G.

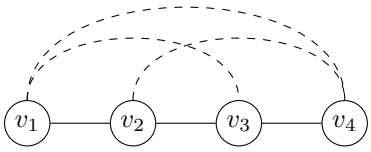

Figure 1: An example graph G_0

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 *unordered*). 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 **??** 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 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. $\{x_1, \ldots x_{l-1}\}$ 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 (sometimes we call such an edge a *chord*).

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.*

formal enough?

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 $\overline{G}$ iff 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 G.*

Definition 0.13 (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 of n vertices as K_n.*

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

Definition 0.15 (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) iff $G[V']$ is an anticlique.

Definition 0.16 (stability number)**.** *A* stability number *of a graph G, denoted as $\alpha(G)$, is a cardinality of its largest induced stable set.*

Definition 0.17 (coloring)**.** *Given a graph G, its* coloring *is a function $c : V(G) \to \mathbb{N}^+$, such that for every edge $xy \in E(G)$, $c(x)$ is different from $c(y)$. A $k-$coloring of G (if exists) is a coloring, such that for all vertices $x \in V(G)$, $c(x) \leq k$.*

Definition 0.18 (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 0.19 (Berge graph)**.** *A graph G is* Berge *if both G and $\overline{G}$ have no odd hole.*

Definition 0.20 (perfect graph)**.** *A graph G is* perfect *if for its every induced subgraph G' we have $\chi(G') = \omega(G')$*

Definition 0.21 (C-major vertices)**.** *Given a shortest off 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 0.22 (clean odd hole)**.** *An odd hole C of G is* clean *if no vertex in G is C-major.*

Definition 0.23 (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 0.24 (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 0.25 (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.*

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' equal to the union of G and a clique K_n. We can see that indeed $\chi(G') = n = \omega(G')$, but it gives us no indication of the structure of G or G'.

A definition of perfect graphs (definition 0.20) states that given a graph G, the chromatic number and cardinality of the largest clique of *its every induced subgraph* should be equal. 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 (definition 0.19) and an algorithm for recognizing Berge graphs in polynomial time (section 1.2).

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 ca-

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

pacity 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 1.3.1.

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 iff 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 0.19) after Claude Berge who studied perfect graphs.

In 1961 Berge conjured that a graph is perfect iff 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 iff 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 G or $\overline{G}$ has a 2-join.*

See [**MC06**] for the proof of theorem 1.1.4 and theorem 1.1.3.

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 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.

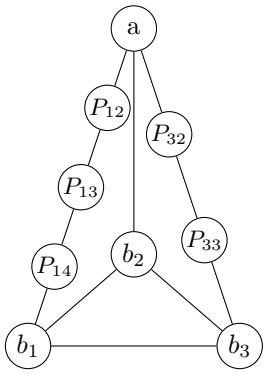

Figure 1.1: An example of a pyramid.

- $\forall_{1 \leq i < j \leq 3}$ 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 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.

Begin.

Enumerate all 6-tuples $(b_1, b_2, b_3, s_1, s_2, s_3)$ such that:

- $\{b_1, b_2, b_3\}$ is a triangle

- for $1 \leq i < j \leq 3$, $\{b_i, s_i\}$ is disjoint from $\{b_j, s_j\}$ and $b_i b_j$ is the only edge between them

- 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 $s_i = b_i$.

There are $O(|V(G)|^6)$ 6-tuples, and for each we check above conditions in $O(|V(G)|)$ time. For each such 6-tuple we follow with the rest of the algorithm.

Let $M \leftarrow V(G) \setminus \{b_1, b_2, b_3, s_1, s_2, s_3\}$.

For each $m \in M$, set $S_1(m)$ equal to 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. Set S_2 and S_3 similarly. Set $T_1(m)$ to be 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, and set T_2, T_3 similarly. It takes $O(|V(G)|^2)$ time to calculate paths $S_i(m)$ and $T_i(m)$ for all i and m.

Now, we will calculate all possible paths P_i. For each $m \in M \cup \{b_1\}$ we will define a path $P_1(m)$ with paths $P_2(m)$, $P_3(m)$ defined in a similar manner.

If $s_1 = b_1$ let $P_1(b_1)$ be the one-vertex path with vertex b_1, and let $P_1(m)$ be undefined for each $m \in M$.

If $s_1 \neq b_1$, then $P_1(b_1)$ is undefined and for all $m \in M$ we will check if all the following are true:

- m is nonadjacent to all of b_2, b_3, s_2, s_3

- $S_1(m)$ and $T_1(m)$ both exist
- $V(S_1(m) \cap T_1(m)) = \{m\}$
- there are no edges between $V(S_1(m) \setminus m)$ and $V(T_1(m) \setminus m)$

If it is true for some $m \in M$, then we assign $P_1(m) \leftarrow s_1{-}S_1(m){-}m{-}T_1(m){-}b_1$, otherwise we let $P_1(m)$ be undefined. It takes $O(|V(G)|^2)$ to check this, for a given m. We assign P_2 and P_3 in a similar manner. Total time of finding all $P_i(m)$ paths for a given 6-tuple is $O(|V(G)|^3)$.

Now we want to check if there is a triple m_1, m_2, m_3, so that $P_1(m_1)$, $P_2(m_2)$, $P_3(m_3)$ form a pyramid. A most obvious approach of enumerating them all would be too slow, so we do it carefully.

For $1 \leq i < j \leq 3$, we say that (m_i, m_j) is a *good* (i,j)*-pair*, iff $m_i \in M \cup \{b_i\}$, $m_j \in M \cup \{b_j\}$, $P_i(m_i)$ 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.

We show how to find the list of all good $(1,2)$-pairs, with similar algorithm for all other good (i,j)-pairs. For each $m_1 \in M \cup \{b_1\}$, we find the set of all m_2 such that (m_1, m_2) is a good $(1,2)$-pair as follows.

If $P_1(m_1)$ does not exist, there are no such good pairs. If it exists, 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. (We can do this in $O(|V(G)|^2)$) Then for each $m_2 \in M \cup \{b_2\}$, test whether $P_2(m_2)$ exists and contains no black vertices. We do this for all m_1 and get a set of all $(1,2)$-good pairs. In similar way we calculate all good $(1,3)$-pair and $(2,3)$-pairs (in $O(|V(G)|^3)$ time).

Now, we examine all triples m_1, m_2, m_3 such that $m_i \in M \cup \{b_i\}$ and test whether (m_i, m_j) is a good (i,j)-pair. If we find a triple such that all three pairs are good, we output that G contains a pyramid and stop.

If after examining all choices of $b_1, b_2, b_3, s_1, s_2, s_3$ we find no pyramid, output that G contains no pyramid. Since there are $O(|V(G)|^6)$ such choices and it takes a time of $O(|V(G)|^3)$ to analyze each one, the total time is $O(|V(G)|^9)$. *End*

It is easy to see, that if algorithm outputs that G contains a pyramid, it indeed does. The proof of the converse is rather technical and we refer to [**MC05**] for it.

Jewels

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

- $v_1, \ldots, v_5$ 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.

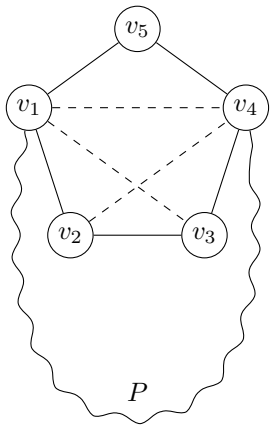

Figure 1.2: An example of a jewel.

Most obvious way to find a jewel would be to enumerate all (possibly chordal) cycles of length 5 as $v_1, \dots v_5$, 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 2.1.1 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.

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.

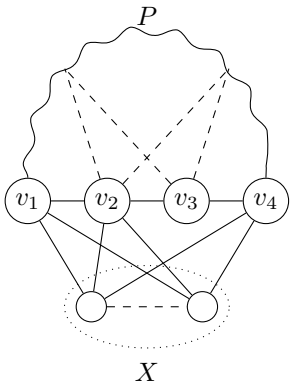

Figure 1.3: An example of a $\mathcal{T}_2$.

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 1.2.1 (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:

Lemma 1.2.2. *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 1.2.1, 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 1.2.1 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: $v_2, x, p_2, \ldots, p_{n-1}, y, v_2$.

□

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.

The following algorithm with running time of $O(|V(G)|^6)$ checks whether G contains a configuration of type 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.

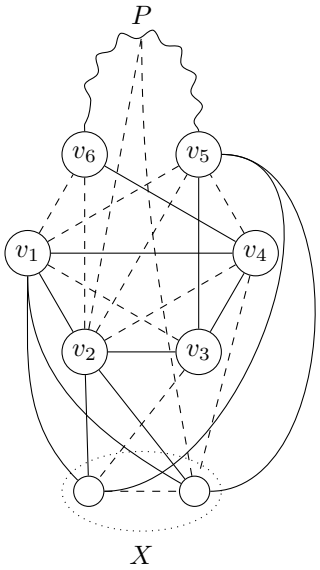

Figure 1.4: An example of a $\mathcal{T}_3$.

Begin.

For each triple v_1, v_2, v_5 of vertices such that v_1v_2 is an edge and v_1v_5, v_2v_5 are nonedges find the set Y of all $\{v_1, v_2, v_5\}$-complete vertices. For each anticomponent X of Y find the maximal connected subset F' containing v_5 such that v_1, v_2 have no neighbors in F' and no vertex of $F' \setminus \{v_5\}$ is X-complete. Let F be the union of F' and the set of all X-complete vertices that have a neighbor in F' and are nonadjacent to all of v_1, v_2 and v_5.

Then, for each choice of v_4 that is adjacent to v_1 and not to v_2 and v_5 and has a neighbor in F (call it v_6) and a nonneibhbor in X, we test whether there is a vertex v_3, adjacent to v_2, v_4, v_5 and not to v_1, with a nonneibhbor in X. If there is such a vertex v_3, find P – a path from v_6 to v_5 with interior in F' and return that $v_1, \ldots v_6, P, X$ is a configuration of type $\mathcal{T}_3$. If we exhaust our search and find none, report that graph does not contain it. *End*

To see that the algorithm 1.2.2 has a running time of $O(|V(G)|^6)$, let us note that for each triple v_1, v_2, v_5 we examine, of which there are $O(|V(G)|^3)$, there are linear many choices of X, each taking $O(|V(G)|^2)$ 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 $O(|V(G)|^6)$.

We will skip the proof that each graph containing $\mathcal{T}_3$ is not Berge. See section 6 6.7 of [**MC05**] for the proof.

1.2.2 Amenable holes.

Theorem 1.2.3. *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 1.2.3 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: $O(|V(G)|^5)$ 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, b are distinct (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 1.2.4 (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.*

Proof.
do we want it here? 1-2 pages long

□

Let us suppose that C is an amenable hole in G. By theorem 1.2.4, 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 1.2.4 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 clener, 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 $r(x,y)$ be infinite.

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$)$
- $r(x_3, y_1), r(x_3, y_2) \geq n$

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**].

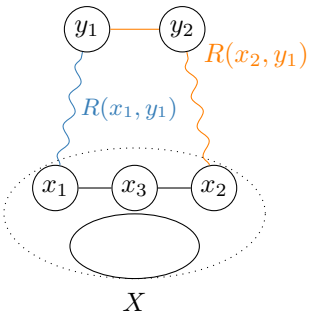

Figure 1.5: An odd hole is found

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?

1.3 Coloring Perfect Graphs

A natural problem for perfect graphs is a problem of coloring them. In 1988 Martin Grötschel et al. published an ellipsoid-method-based polynomial algorithm for coloring perfect graphs [**Grtschel1993**]. We consider it in section section 1.3.1. 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 1.3.4), however there is still no known polynomial combinatorial algorithm to do this.

better wording of this paragraph

1.3.1 Information theory background

Shannon Capacity of a graph

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 is the background which motivated Berge to introduce perfect graphs [**Chudnovsky2003**].

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. 1.6) in which vertices correspond to signal values and two vertices are connected iff values they represent can be confused.

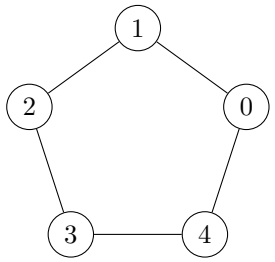

Figure 1.6: Example noisy channel

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 $v_1v_2 \dots v_k$, $w_1w_2 \dots w_k \in G(V)^k$ of length k are non-confoundable iff 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 $(i, u)(j, v) \in E(G \cdot H)$ iff $ij \in E$ 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 (polynomial or not). If we could calculate $\alpha(G^k)$ for a first few values of k (we will show how to do it in algorithm 1.3.1) we could have a lower bound on $\Theta(G)$. Let us now turn into search for some usable upper bound.

Lovász number

The following introduction of what has become known as *Lovász number* is based on excellent lecture notes by Lovász [**Lovasz95**].

rewrite this so its not blatantly copied.

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 $|S| \leq 5^{k/2}$. Thus we get $\alpha(C_5^k) \leq 5^{k/2}$ and $\Theta(C_5) = \sqrt{5}$.

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 :

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

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

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

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 equal in some graphs, including all perfect graphs, as is demonstrated in the Lovász "sandwich theorem" (theorem 1.3.1).

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.

1.3.2 Computing ϑ

TODO

Theorem 1.3.1 (Lovász "sandwich theorem")**.** *For any graph G:*

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

Because in perfect graphs $\omega(G) = \chi(G)$, we get $\omega(G) = \vartheta(G) = \chi(G)$.

TODO

1.3.3 Coloring perfect graph using ellipsoid method

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

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 $< \frac{1}{2}$.

We will now show how to find a stable set in G of size $\alpha(G)$.

Algorithm 1.3.1 (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 1.3.1 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' iff 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.

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 1.3.2
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 1.3.1 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 1.3.1. 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 1.3.1 on it. Therefore total running time is polynomial.

Minimum coloring

Algorithm 1.3.3
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 1.3.2). 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 1.3.3 is polynomial.

1.3.4 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.

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.

Chapter 2

Implementation

link to repo

2.1 Berge graphs recognition

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 2.1.1) make a very big difference and make it usable – or at least much more usable than naïve algorithm could be. Also, we explore a new frontier of implementing its most time consuming part on massively parallel GPU architecture, with good results (section 2.1.3).

Anything interesting about algo/data structure?

2.1.1 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 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 $run1$ and $run2$, with $run2$ calls taking twice as much time. $run1$ and $run2$ are in turn called from main. Total attribution of *worker* would be nearly 100%, of $run1$ about 33% and of $run2$ 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

Generating paths

we also optimize naïve, in fact it makes a bigger difference there.

A major part of Berge recognition algorithm is 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 a quick method for generating paths, a problem that could seem trivial at first, is a crucial part to optimize and we give it much attention.

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 nowhere in the algorithm we need all the paths at once. It would suffice for us to have a path, check if it has the properties we further require of it and then proceed to generate next path (in some ordering of paths). Therefore, we need a method that receives 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). 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 one before last vertex. 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.

This method is significantly faster than simply enumerating all combinations, but still leaves a problem of generating paths a main bottleneck of our algorithm (in our preliminary test it took about 70% of running time for graphs with $|V(G)| = 10$ and growing with bigger graphs.

But we can do much better with some care and better data structure. For

a given graph we create a data structure that suits best the goal of generating paths. Instead of 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, with much bigger speedup for naïve algorithm (around TODOx).

check this

Array unique For each path candidate we call a subroutine 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 *unique* procedure. Let's instead have a static array *stamp* of ints and a static *counter* of how many times we called *unique* 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 *unique* method we increment *counter*.

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

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 use it whenever possible. It can be easily modified to generate all possibly chordal paths or holes and therefore has much use in the 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 T_2 and T_3.

2.1.2 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 2.1.1). 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 2.1.3).

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

2.1.3 Parallelism with CUDA (?)

a section?

CUDA background

moderngpu - allows us to run simple transforms

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 implement whole naïve algorithm and most time consuming parts of the CCLSV algorithm.

a good name for it?

Naïve parallelisation

Naïve algorithm is very simple – generate all possible odd hole candidates, check if any of them is an odd hole and repeat for the complement. But its parallelisation is not trivial.

The biggest problem of implementing naïve algorithm is in splitting the work between CUDA threads. This is very important because while GPU is faster when all its cores are working efficiently, single core performance is much slower than CPU. But it is not obvious how to split the work.

Our first attempt was to switch our slightly more sophisticated way of generating paths (section 2.1.1) in favor of generating all combinations. We could code a prefix of a combination as a number (we use combinations with repetitions for simplicity). This would give us $|V(G)|^k$ codes for all prefixes of length k. Then each CUDA thread would process its part of all codes: for each code it would first check if the encoded prefix is a valid path and then generate all path candidates with that prefix. This approach has two problems: it doesn't split the work evenly enough and it does too much unneeded work (only a few prefix codes are valid). Still, implementing it on GPU gives us a program with speeds comparable to naïve CPU algorithm after all optimizations.

A better approach is to generate all paths of some length k on CPU (using algorithm described in section 2.1.1), copy them on GPU and have each thread process them as described above. This is superior to the previous algorithm in utilizing GPU – each prefix takes similar time to process. We also even out the work done by each thread by having single thread process multiple prefixes. But it puts much more strain on the CPU – it has to generate all valid prefixes. Here we used experiments to determine best value of k (it was 7 for graphs of $n < 15$ and 6 for larger). This yields an algorithm that beats CPU naïve by a factor of 50x. .

check

check

CCLSV parallelisation

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 and an additional vertex and performs the required checks. This is almost

do it also in CPU

acutally do this

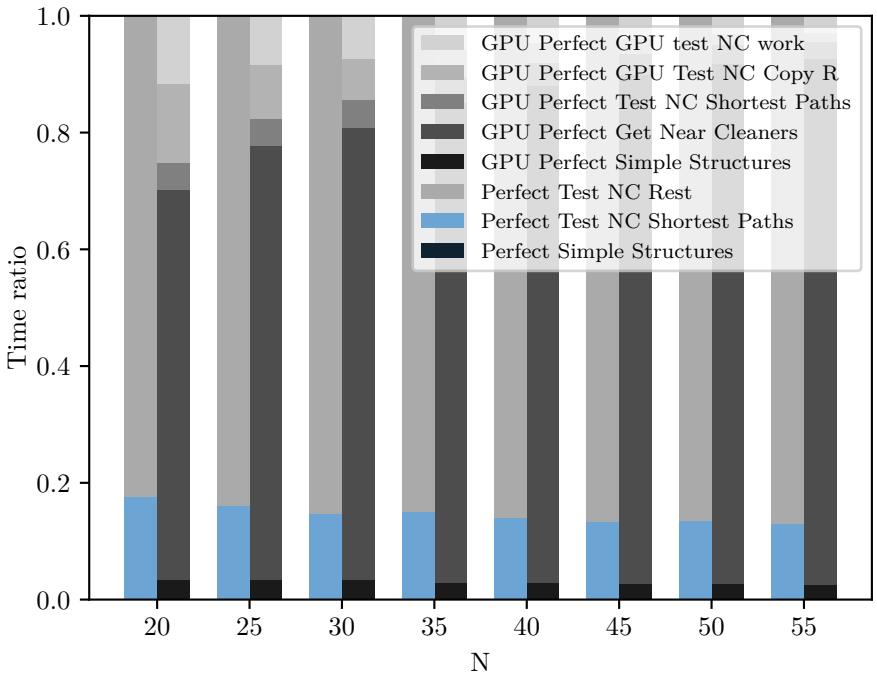

Figure 2.1: testPlot

perfect scenario for 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 30x on bigger tests, which gives us a speedup of about 5x for overall algorithm.

check

implement other CUDA opts and describe them

check

2.1.4 Experiments

Naive algorithm - brief description, bottlenecks optimizations (makes huge difference).

Description of tests used.

Results and Corollary - almost usable algorithm.

2.2 Coloring Berge Graphs

2.2.1 Ellipsoid method

Description.

Implementation.

Experiments and results.

2.2.2 Combinatorial Method

Cite the paper.

On its complexity - point to appendix for pseudo-code.

Appendices

Appendix A

Perfect Graph Coloring algorithm