Exact Exponential Algorithms to solve NP-Hard Problems

A Thesis

submitted to
Indian Institute of Science Education and Research Pune
in partial fulfillment of the requirements for the
BS-MS Dual Degree Programme

by

T I Darsan

Indian Institute of Science Education and Research Pune
Dr. Homi Bhabha Road,
Pashan, Pune 411008, INDIA.

April, 2024

Supervisor: Prof. Soumen Maity

Certificate

This is to certify that this dissertation entitled Exact Exponential Algorithms to solve NP-Hard Problems towards the partial fulfilment of the BS-MS dual degree programme at the Indian Institute of Science Education and Research, Pune represents study/work carried out by T I Darsan at Indian Institute of Science Education and Research under the supervision of Prof. Soumen Maity, Professor, Department of Mathematics , during the academic year 2023-2024.

Prof. Soumen Maity

Committee:

Prof. Soumen Maity

Dr. Prafullkumar Tale

This thesis is dedicated to Vedikettu FC.

Declaration

I hereby declare that the matter embodied in the report entitled Exact Exponential Algorithms to solve NP-Hard Problems are the results of the work carried out by me at the Department of Mathematics, Indian Institute of Science Education and Research, Pune, under the supervision of Prof. Soumen Maity and the same has not been submitted elsewhere for any other degree.

T I Darsan

Acknowledgments

Many people have helped me and supported me throughout my 5th year. First among them is my parents and my sisters. They have always been with me throughout my life giving constant support. I need to thank my project supervisor, Prof.Soumen Maity who have given me valuable suggestions and helped me throughout my thesis. His course on graph theory was the greatest inspiration and greatest course for me in IISER. I also need to thank Dr.Prafulkumar Tale who has helped me a lot in my midyear presentations and also in my semester project on Cops and Robbers game. He has given me instructions on report writing how to approach research which was all very valuable to me.
This acknowledgement cannot end without saying thanks to my friends at IISER without whom I couldn't have survived this 5 years. This includes my batchmates, seniors and some juniors. I also thank Vedikettu FC and other football, chess and kho-kho players at IISER who helped me get through tough times.

Abstract

The first part of the thesis is mainly a literature study of the book, "Exact Exponential Algorithms" by Fomin and Krastch. We explain different techniques to solve computational problems through exact algorithms. The techniques are branching, dynamic programming and measure and conquer and also treewidth. These are explained using examples and different algorithms.

In the second part of the thesis, we introduce the problem, Component order connectivity using some examples and ILP formulation. We also introduce it's edge version, component order edge connectivity and parameterized version and give some results. Then we give algorithms to find the solution when the instance is tree or a bounded degree graph.

Contents

Chapter 1

Preliminaries

This chapter mainly contains an introduction to particular topics of graph theory and algorithms. An understanding of these topics is needed to comprehend the later chapters in this thesis.

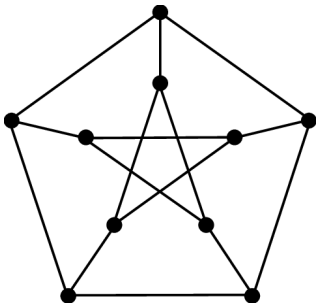
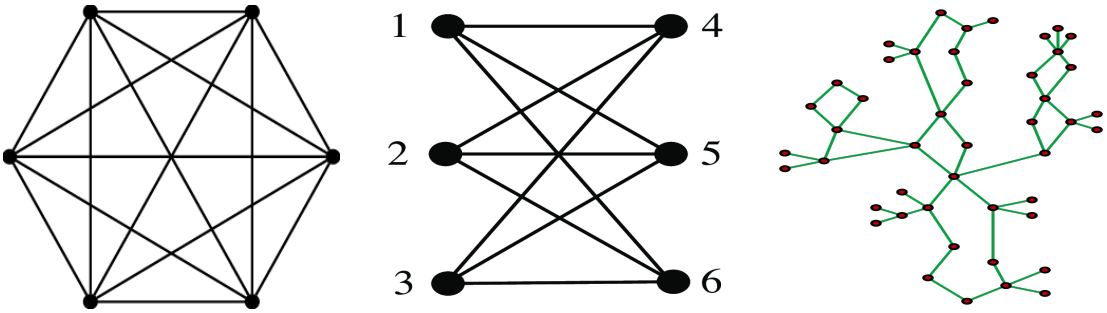


Figure 1.1: Peteresen Graph

1.1 Graphs

Graphs in the case of graph theory is often confused with graph or plotting of a function. But here, we are dealing with graphs which are mathematical structure to show the relation between objects. The objects are represented via vertices or nodes and the relation is represented via edges between these vertices. A simple graph is a graph which has no multiedges (there is at most one edge between any pair of vertices). A simple graph can represented as $G(V, E)$ where V is the set of vertices and E is the set of edges. An example of a graph is a

cycle with 5 vertices which can be represented as $G(V, E)$ where $V = \{v_1, v_2, v_3, v_4, v_5\}$ and $E = \{(v_1, v_2), (, v_2, v_3), (v_3, v_4), (v_4, v_5), (v_5, v_1)\}$ or a Petersen graph which has all the vertices with degree 3. A walk in a graph is a set of vertices, where each consecutive vertices has an edge between them. A path is a walk with no vertices is present more than one time. More properties of graphs including connectivity, coloring, ramsey theory and hamilton cycles can be found in the book, 'Graph Theory' by Reinhard Diestel [1].



(a) Complete Graph (b) Bipartite Graph (c) Cactus Graph

Definition 1.1.1. *A graph is called connected if there is a path between any pair of vertices in the graph. Disconnected graphs have more than one connected component which itself are connected subgraphs of a graph.*

Some of the common graphs which we will deal with are:

Complete Graphs Every pair of vertices has an edge between them.

Bipartite Graphs The vertex set can be divided into two subsets of vertices that have no edge between them.

Tree A connected acyclic graph.

Unicycle A graph containing exactly one cycle.

1.2 Algorithms

An algorithm is a set of instructions that are to be followed to solve a particular problem. Most of the computational problems we deal with can be treated as graph problems. Graph

algorithms mainly deal with finding graphs that belong to a specific class like graphs that have no cycles in them (trees). Some basic graph algorithms are Breadth First Search or BFS and Depth First Search or DFS. These are graph traversing or searching algorithms that find or search for a node with specific criteria or subgraphs with specific features [2]. In DFS, the algorithm starts at the root node and traverses along a path as long as possible. Backtracking happens only when there are no unvisited vertices in the neighbours of the last visited vertex. Backtracking is done to a vertex with an unvisited neighbour and the algorithm is continued similarly. In BFS, the algorithm first visits all the unvisited vertices at the lowest distance from the root node. This classifies vertices into different levels according to their distance from the root vertex.

1.3 Computational Complexity

Definition 1.3.1. *Given two functions f and g defined on $\mathbb{N}$, we denote $f \in \mathcal{O}(g)$ if there is some positive real constant c such that $f(n) \leq c * g(n)$ for every n. Similarly, we denote $f \in \mathcal{O}^*(g)$ if there is some c such that $f(n) \leq c * g(n) * poly(n)$ for every n.*

The complexity of an algorithm is usually measured in time complexity which is the required time needed for the algorithm to compute the solution to the problem. The main quantity used to classify the hardness of a problem is the number of basic operations such as addition, multiplication, logical OR and logical AND required to solve it. A problem is said to be solvable in time $T(n)$ if it can be solved using at most $T(n)$ basic operations, where n is the size of the input. For example, matrix multiplication of two $n \times n$ matrices take $\mathcal{O}(n^3)$ time with standard matrix multiplication algorithm. This is because the algorithm takes $\mathcal{O}(n)$ time to calculate each of the n^2 elements of the required matrix.

Some of the complexity classes are:

P is the class of problems that can be solved in polynomial time.

> **Input:** A graph G, a number k
> **Question:** Does G contain a vertex of degree k?

Here the algorithm would be to go through each vertex of the graph G and find it's degree in the graph. This would take $\mathcal{O}(n^3)$ time which is polynomial with the number of vertices as n.

NP is the class of decision problems that can be solved in polynomial time on a nondeterministic machine. In other words, it is the class of problems whose solutions can be verified in polynomial time.

> **Input:** A graph G, a number k
> **Question:** Does G contain an independent set of vertices of size greater than or equal to k

Here, given a vertex set of size greater than or equal to k, we can verify if there exists an edge between any two vertices in the vertex set by going through the edge set which takes at most $\mathcal{O}(n^2)$ time.

NP-hard A lot of times you can solve a problem by reducing it to a different problem. I can reduce problem B to Problem A if, given a solution to Problem A, I can easily construct a solution to Problem B. (In this case, "easily" means "polynomial time"). A problem is **NP-hard** if all problems in **NP** are polynomial time reducible to it.

NP-complete A problem is **NP-complete** if it is both **NP-hard** and **NP**

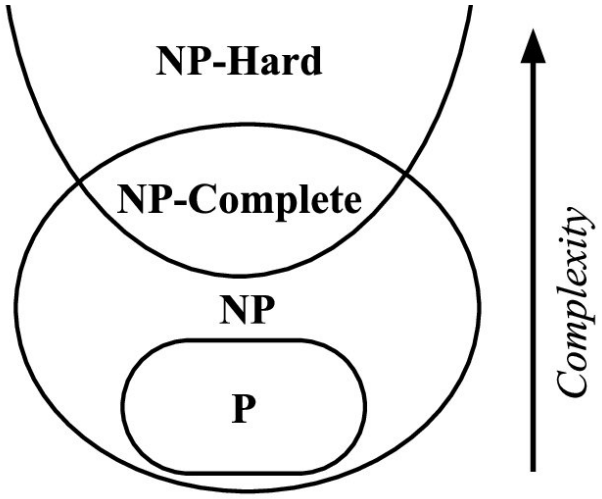


Figure 1.3: Complexity

These classes satisfy the relation $\mathbf{P} \subseteq \mathbf{NP}$ as an algorithm solvable in poly time can also be verified simply by finding all the solutions. The question whether $\mathbf{P} = \mathbf{NP}$ is hardest and unsolved questions in complexity theory and whole of the theory is built under the hypothesis that $\mathbf{P} \neq \mathbf{NP}$. More details on complexity and different complexity classes can be found in [3].

Definition 1.3.2 (Polynomial-Time Reductions:)**.** *Let A,B be two decision problems. We say that A reduces to B, denoted by $A \leq_p B$ if there exists a polynomial time computable function which takes an input instance x of A and converts it into an instance $f(x)$ of B such that x is a Yes instance of A if and only if $f(x)$ is a Yes instance of B.*

Theorem 1. *If X is an **NP-complete** problem, then the existence of a polynomial time algorithm for X implies **P = NP***

Theorem 2. *If Y is **NP-complete** and X is a problem in NP satisfying $Y \leq_p X$ then X is also **NP-complete***

There are multiple ways to approach **NP-hard** problems. Compromising on accuracy gives us approximation algorithms, weakening the requirement to solve every in stance leads to parameterised algorithms and algorithms for special classes of input. In this thesis, we mainly deal with exponential algorithms which we get when we compromise on time complexity or efficiency and go for algorithms running on exponential time but finds the exact solutions to the problems.

Chapter 2

Exact Exponential Algorithms

This chapter is mainly a literature review of the book, 'Exact Exponential Algorithms' by Fomin and Kratsch[4]. The book provides techniques and ways of approaching graph problems through exact algorithms. The techniques are shown through different examples and algorithms. Some of the main techniques to design exponential algorithms are branching, dynamic programming, measure and conquer, inclusion-exclusion, split and list, subset convolution, etc. The book also provides insights into treewidth.

2.1 Branching

Branching is one of the most used and basic techniques to design algorithms running in exponential time. Branching is done by breaking down the input instance into small instances and solving them. The key is as we are considering all the smaller instances that can happen, solving these instances simultaneously solves the input instances. This is done recursively till the algorithm ends or we finds a trivial instance. Branchig consists of *reduction rules* and *branching rules*. *Reduction rules* are those rules which take care of trivial instances and simplify the problem instance. *Branching rules* break down each instance into smaller instances.

Branching algorithms are evaluated using it's worst case time complexity. This is found by treating such algorithms as search trees. The input instance is taken as the root node

and *reduction rules* applied if possible. Then the root node gives rise to children nodes according to the *branching rules*. The no. of child nodes depends upon the number of smaller instances the algorithm branches into. The *reduction rules* are applied to these nodes until an instance is solved (gives a yes or no answer). A positive answer in one of the branches(smaller instance) tells that the input is a yes instance. The time complexity is by assuming each *reduction rules* and *branching rules* is done in polynomial time and therefore, the algorithm is running in time proportional to the number of nodes in the search tree. The number of nodes in the search tree is further bounded by the number of leaves in the search tree, which is taken as $T(n)$. Let us take the size of the input instance to be n and the algorithm branches the input instance into d smaller instances of size $n-n_1, n-n_2, ..., n-n_d$ respectively. The branching vector of such an algorithm is $(n_1, n_2, ..., n_d)$ Then the recursive relation is given by

$$T(n) \leq T(n-n_1) + T(n-n_2) + ... + T(n-n_d)$$

This can be solved by assuming the An base case solution is always of the form c^n where c is the solution of the equation:

$$x^n = x^{n-n_1} + x^{n-n_2} + ...x^{n-n_d}$$

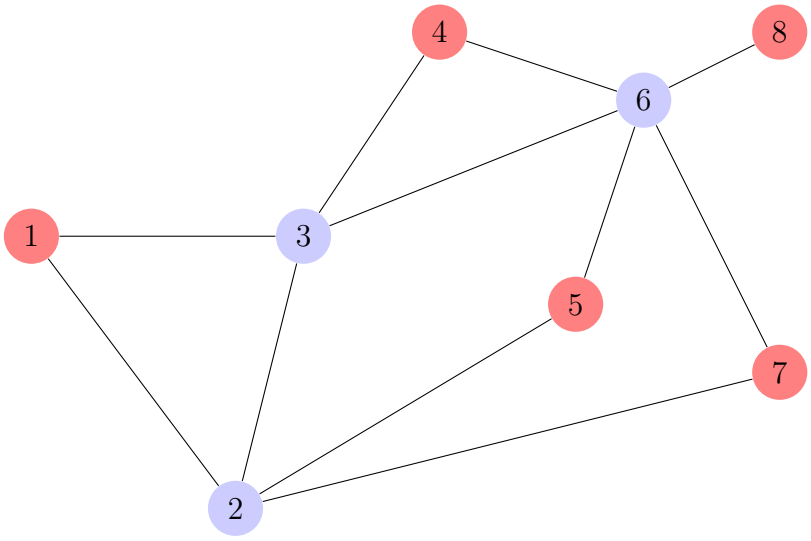
2.1.1 Independent Set

Independent set is a set of vertices such that there is no edge between any pair of vertices in the set. It is also called a stable set. Independent set problem is one of the most discussed problems in graph algorithms research. It is also treated as a vertex cover problem because the vertex cover problem can be easily reduced into independent set problem. In the vertex cover problem, we need to find a set of vertices such that all the edges in the graph are incident to at least one vertex from this set. It is very easy to prove that an independent set of size $\geq k$ exists in a graph if and only if there is a vertex cover of size $\leq (n-k)$.

Maximum Independent Set (MIS)
INPUT: A graph G
QUESTION: What is the maximum size of an independent set in G?

Minimum Vertex Cover (MVS)
INPUT: A graph G
QUESTION: What is the minimum size of a vertex cover in G?

In the following example {1,4,5,7,8} (red colored vertices) is an independent set of maximum size and {2,3,6} is a vertex cover of minimum size.



Branching can be easily applied to obtain the maximum independent set. Following observations or rules are applied when solving via branching:

- Isolated vertices (degree 0 vertices) can be directly added to the solution set.

$$MIS(G) = 1 + MIS(G \setminus v)$$

- Vertices of degree one, say v can be directly added to the solution set and it's neighbour can be removed from the graph. This is because there is always some solution of maximum size which contains v. If a solution doesn't contain v, it should contain it's neighbour u, otherwise it is not of maximum size. We can change this independent set by adding v and removing u creating a new solution containing v.

$$MIS(G) = 1 + MIS(G \setminus \{v, u\})$$

- Now, the remaining graph has minimum degree at least 2. So following *branching rule* can be applied.

$$MIS(G) = \max(1 + MIS(G \setminus N[v]), MIS(G \setminus v))$$

Here, we branch on the highest degree vertex, say v. There are two possibilities: either v is in the solution or not. If v is in the solution, it's neighbours cannot be in the solution. This gives rise to the first branch where v is added to the solution and all it's neighbours including v ($N[v]$)is removed from the graph. Second branch takes the case where v is not in the solution in which case it is just removed from the graph.

Algorithm 1 MIS(G)

```
1: if V(G) = φ then
2:     return 0
3: else if ∃v such that d(v) = 0 then
4:     return MIS(G \ v)
5: else if ∃v such that N(v) = {u} then
6:     return 1 + MIS(G \ {v, u})
7: else
8:     Pick a vertex v with largest degree
9:     return max(1 + MIS(G \ N[v]), MIS(G \ v))
```

In the algorithm above, the first three rules are reduction rules which simplify the instances. The algorithm goes into branching only if no more reduction rules can be further applied. After removing vertices of degree 0 and 1, the graph is left with vertices of degrees greater than or equal to 2. Now, the branching rule is applied. The algorithm branches on the vertex with the highest degree. The branching factor is always greater than (1,3) as the lowest degree is 2. This is because there are two branches: in first branch, removing $N[v]$ gives a reduction of at least 3 and in second branch removing v gives reduction of 1. A branching factor of (1,3) corresponds to solving this recurrence equation:

$$x^3 - x^2 - 1 = 0$$

Solving the equation gives us x = 1.4657 and the worst time complexity of this branching algorithm is $\mathcal{O}(1.4657^n)$. More complex algorithms with better running time for independent set problem can be found at [5].

2.2 Dynamic Programming

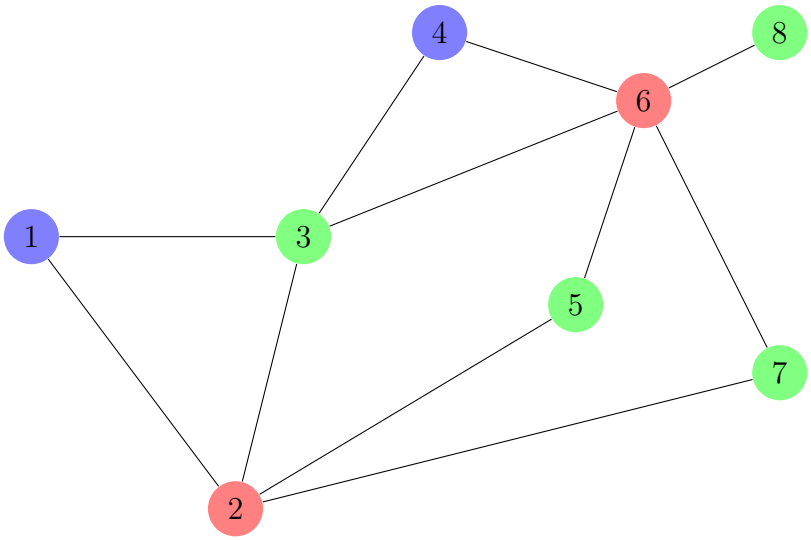
Dynamic Programming is one of the basic techniques used to solve computational problems. It can used to design algorithms running in exponential and also in some cases polynomial time. In dynamic programming, we first solve smaller instances and use these solutions to build solutions for larger instances. This is quite the opposite of branching where we try to decompose the problem. Most often dynamic programming takes exponential space while branching can be done in polynomial space.

2.2.1 Graph Coloring

A $k-coloring$ of a graph gives one of the k colors to each vertex of the graph such that no two adjacent vertices have same colors. This can also be thought as assigning one number $c : V \to \{1, 2, \ldots, k\}$ to each vertex of the graph. The least number k where this colouring is possible is called the chromatic number of G, denoted by $\chi(G)$. A coloring c of G using $\chi(G)$ colors is called an optimal coloring. In the Coloring Problem, the task is to find the chromatic number of G. More details and algorithms can be found in [6].

$\chi(G)$
Input: A graph G
Question: What is the chromatic number of G?

The following graph is coloured using 3 colours, which is basically partitioning the vertex set into 3 subsets: {1,4}, {3,5,7,8} and {2,6}



The brute force algorithm would be to try assigning each color to each vertex of the graph. Chromatic number is always less than n. So, this would take $\mathcal{O}^*(n^n)$ time, which is $2^{nlog(n)}$ time.
In the dynamic programming method, we try to find the largest independent set in the graph and assign it one color and remove the set from the graph and solve for the remaining graph.

- For every $X \subseteq V$ we set $OPT[X] = \chi(G[X])$ where $G[X]$ is the subset induced by X.
- We set $OPT[\phi] = 0$
- We now define the dynamic programming recursion which assigns one color to some maximal independent set.

$$OPT[X] = 1 + min\{OPT[X \setminus I] : \text{I is a maximal independent set of G[X]}\}$$

- We now set $\chi(G) = OPT[V]$

The algorithm runs on all subsets $X \subseteq V$ and for all such X, we need to find all the maximal independent sets. If we do this by brute force it takes $2^{\text{ mod } X}$ time. So total time is given by:

$$\sum_{i=1}^{n} \binom{n}{i} .2^i = 3^n$$

If we consider only maximal independent sets of X at each instance, we can reduce the running time as number of maximal independent sets of a graph with i vertices is upper bounded by $3^{i/3}$. The running time is given by:

$$\sum_{i=1}^{n} \binom{n}{i} .3^{i/3} = (1 + \sqrt[3]{3})^n < 2.4423^n$$

2.2.2 Set Cover

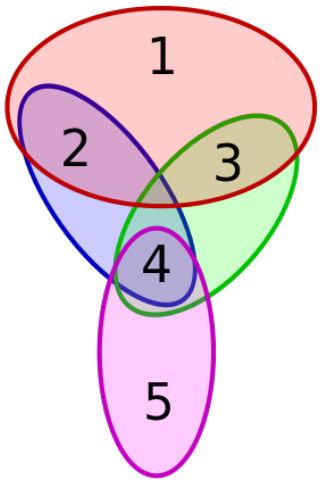
In the MINIMUM SET COVER problem, we are given an universe of elements, $\mathcal{U}$ and a collection of sets, $\mathcal{S}$ of non empty sets. The task is to find the minimum size of a subset $\mathcal{S}' \subseteq \mathcal{S}$ such that $\mathcal{S}'$ covers $\mathcal{U}$. Covering of a universe by a set can be given as:

$$\cup_{S \in \mathcal{S}'} S = \mathcal{U}$$

$MSC(G)$
INPUT: An universe $\mathcal{U}$ and a collection of sets $\mathcal{S}$
QUESTION: What is the minimum cardinality of a sub collection $\mathcal{S}' \subseteq \mathcal{S}$ which covers $\mathcal{U}$?

In the following example,
$\mathcal{U} = \{1, 2, 3, 4, 5\}$ and
$\mathcal{S} = \{\{1, 2, 3\}, \{2, 4\}, \{3, 4\}, \{4, 5\}\}$
The required subset is,

$$\mathcal{S}' = \{\{1, 2, 3\}, \{4, 5\}\}$$



- For every set $U \subseteq \mathcal{U}$ and every $j = 1, 2, \ldots, m$, we define $OPT[U, j]$ as the minimum size subset of $\{S_1, S_2, \ldots, S_j\}$ which covers U. If $\{S_1, S_2, \ldots, S_j\}$ doesn't cover U, then

set $OPT[U, j] = \infty$

- Now we set for every $U \subseteq \mathcal{U}$, $OPT[U, 1] = 1$ if $U \subseteq S_1$ and $OPT[U, 1] = \infty$ otherwise.
- In step $j + 1$, we compute $OPT[U, j + 1]$ as follows:

$$OPT[U, j + 1] = \min(\, OPT[U, j],\, OPT[U \setminus S_{j+1}, j] + 1)$$

- $OPT[U, m]$ is the cardinality of the minimum set cover of $\mathcal{U}$

In this Dynamic Programming algorithm, each step takes $\mathcal{O}(n)$ time and this is to be calculated for every set $U \subseteq \mathcal{U}$ and every $j = 1, 2, \ldots, m$. This gives us the total running time of $\mathcal{O}(nm2^n)$.

Minimum Dominating Set
INPUT: A graph G
QUESTION: What is the minimum size of a dominating set in G?

The algorithm for minimum set cover can be further used to find the minimum dominating set in a graph where the dominating set problem can be reduced to minimum set cover. This is by defining the set of vertices as universe $\mathcal{U}$ and the set of edges as a collection of sets $\mathcal{S}$. A dominating set is a set of vertices in a graph such that all the vertices in the graph are at most at a distance of 1 from the dominating set. More algorithms for dominating sets can be found in [7].

2.3 Measure and Conquer

Measure and conquer is one of the most powerful tools to analyse the running time of a branching algorithm. Instead of creating more complex branching and reduction rules, we introduce a new parameter called measure which makes the analysis of running time more simple. The measure should have the following features:

- Measure of an instance should be always nonnegative.

- Measure of the input instance should be bounded by some polynomial function of the size of the input.
- Measure of the instance of a subproblem obtained by applying reduction or branching rules on the input instance should be less than the measure of the input instance.

For calculation of running time of an algorithm, we make use of measure drop vector similiar to branching vector. This is given as$\{t_1, t_2, \ldots t_n\}$ and the corresponding recurrence relation is given by:

$$T(\mu) = T(\mu - t_1) + T(\mu - t_2) + \cdots + T(\mu - t_n)$$

This is solved to obtain an upper bound on running time as: $T(\mu(G)) \leq c^{\mu(G)} \leq c^{f(n)}$

2.3.1 Independent set

We have already done a branching algorithm for the independent set problem. Now we modify the algorithm little bit and try to analyse the running time using measure and conquer.

Algorithm 2 MIS(G)

if $\exists v$ such that $d(v) = 0$ **then**
  return $1 + MIS(G \setminus v)$
else if $\exists v$ such that $N(v) = \{u\}$ **then**
  return $1 + MIS(G \setminus \{v, u\})$
else if $\Delta(G) \geq 3$ **then**
  Pick a vertex v with largest degree
  return $\max(1 + MIS(G \setminus N[v]), MIS(G \setminus v))$
else if $\Delta(G) \leq 2$ **then**
  Compute $\alpha(G)$ ing polynomial time
  return $\alpha(G)$

- First line is a reduction rule which removes isolated vertices and adds them to the solution set.
- The Second line is also a reduction rule that removes vertices of degree 1 and adds them to the independent set while removing its neighbour from the graph.

- The Third line is a branching rule which is done only if there is at least a vertex with degree ≥ 3. This helps in increase the running time of the algorithm. Measure and conquer analysis is done in this part of the algorithm.
- Fourth line is computes only if all the vertices in the graph has degree ≤ 2 which simple means the graph is a collection of lines and cycles. Maximum independent set size for this graphs can be calculated in polynomial time easily.

For the branching rule, we define two variables, IN which is the decrease in measure when a vertex is taken into the solution set and OUT when the vertex is not in the solution and removed from the graph.

First, we consider the case where the measure is simply the size of input or $\mu(G) = n$. A simple analysis of the branching algorithm gives us $IN \geq 4$ and $OUT \geq 1$ as if the vertex is in the solution. all its neighbours are removed which are at least 3 in number and there is a decrease of one in the measure if the vertex is removed from the graph. So we get a branching vector of (1,4) which would be put into the polynomial equation to obtain the corresponding worst time complexity as $\mathcal{O}^*(1.3803^{\mu(n)}) = \mathcal{O}^*(1.3803^n)$

Second, we define more complicated measures. As we need more and more vertices of higher degree in the graph so that we can get good running time, we give less weight to lesser degree vertices, ie vertices of degree less than 2. The new measure is given by $\mu(G) = v_{\geq 3} + \frac{v_2}{2}$. Here $v_{\geq 3}$ is the number of vertices with a degree greater than or equal to 3 and v_2 is the number of vertices with degree 2. We don't give any measure to lesser degree vertices as these would get removed in the reduction rule itself. Now, we observe than if v has neighbours with degree 2, their degree get reduced by 0.5 if v is not in the solution also. So this gives us that any neighbour vertex contributes a drop in measure by 1 when both the branches taken together. This gives $IN \geq 1, OUT \geq 1, IN + OUT \geq 1 + d(v)$. To calculate the running time, we consider two cases. First when $d(v) \geq 4$, now we get the worst running time for the branching factor (1,5) which corresponds to $\mathcal{O}^*(1.3248^n)$. In the second case, $d(v) = 3$. Now we know that even if we remove or add v to the solution, the measure due to any neighbouring vertex decreases by 0.5. If the neighbouring vertex is of degree 3 it gets reduced to 2 which decreases the measure and if the degree is 2, it gets reduced to 1. This gives $IN, OUT \geq 1 + \frac{d(v)}{2} = 2.5$. This gives branching factor as (2.5,2.5) corresponding to running time of $\mathcal{O}^*(1.3196^n)$. Comparing the two cases, we get the worst case running time as $\mathcal{O}^*(1.3248^n)$ for this measure. We can further improve the running time by optimize the

weight given to each degree vertices [8]. In fact, setting the weight of degree 2 as 0.5966 and degree 3 as 0.9287 while keeping the rest the same, gives an improved bound of $\mathcal{O}^*(1.2905^n)$.

2.4 Treewidth

Treewidth[9] is one of the fundamental properties of a graph. This is very useful when designing exact and also parameterized algorithms. A tree decomposition of a graph is creating a tree and including some features of the given graph inside the vertices of this tree.

A tree decomposition of a graph $G = (V, E)$ is the pair $(\{X_t, t \in T\}, T = (T, F))$ where $\{X_t, t \in T\}$ is called bags which are collection of subsets of V and T is the required tree which has the following properties:

- Each vertex v of graph G is present in some bag X_t.
- Vertices of each edge, (u, v) of G is present in some bag X_t
- The tree vertices whose bag contains a particular vertex v of G make a connected subtree. For every $v \in G$, the set $\{t \in T : v \in X_t$ makes a connected subtree of T.

The width of the tree decomposition is given as $max_t|X_t| - 1$. The treewidth of a graph G, $tw(G)$, is the minimum width of a tree decomposition of G.

- The treewidth of a tree, T is 1. This is by forming the same tree as the tree decomposition and forming the bag of vertex t as $X_t = \{t, parent(t)\}$. This gives us a tree decomposition with width 1.
- The treewidth of a cycle,C_n is 2. This is by forming a path with $n - 1$ vertices as tree decomposition of T_{n-1} and putting v_n to all the bags. This satisfies the necessary conditions of a tree decomposition and the width is 2.

The path decomposition of a graph is the same as tree decomposition but the tree should be path. A path decomposition can be denoted as a set of successive bags:

$$(X_1, X_2, \ldots, X_d)$$

The minimum width of a path decomposition is given as pathwidth or $pw(G)$.

A path decomposition $(X_1, X_2, \ldots, X_d)$ is nice iff $|X_1| = |X_d| = 1$ and for every i from 1 to d-1, there is a vertex v such that either $X_{i+1} = X_i \cup \{v\}$ or $X_{i+1} = X_i \setminus \{v\}$. Any path decomposition can be converted to another nice path decomposition in polynomial time.

2.4.1 Maxmimum Cut

A cut of a graph takes two disjoint vertex subsets from a graph as input and gives the number of edges between the vertex subsets as the output. $CUT(X, Y)$ where $X, Y \subseteq V$ is the number of edges between X and Y.

Maximum Cut
INPUT: A graph G
QUESTION: Find $X \subseteq V$ which maximises $CUT(X, V \setminus X)$

For the algorithm, we first find the nice path decomposition of the graph G with width k in polynomial time:

$$P = (X_1, \ldots, X_r)$$

We now define $V_i = \cup_{j=1}^{i} X_j$ and $c_i(A, B)$ where $A, B \subseteq V_i$ is the maximum number of edges between some $A' \subseteq A$ and $B' \subseteq B$ in V_i. Now, the required solution is

$$max\{c_r(A, B) : (A, B) is a partition of V\}$$

We use dynamic programming to compute $c_i(A, B)$. In the case where $i = 1$, we have only one vertex, v in X_1. Therefore, $c_1(v,) = c_1(, v) = 0$. For $i > 1$, we consider two cases:

- $X_i = X_{i-1} \cup \{v\}$ for some v: Now, all neighbours of v must be in $X_i \cap X_{i-1}$. This gives:

$$c_i(A, B) = \begin{cases} c_{i-1}(A \setminus \{v\}, B) + CUT(\{v\}, B) & v \in A \\ c_{i-1}(A, B \setminus \{v\}) + CUT(\{v\}, A) & \notin A \end{cases}$$

- $X_i = X_{i-1} \setminus \{v\}$ for some v:

$$c_i(A, B) = max\{c_{i-1}(A \cup \{v\}, B), c_{i-1}(A, B \cup \{v\})\}$$

The running time for computing $c_i(A, B)$ is $\mathcal{O}(2^{|X_i|}|X_i|)$. Therefore the total running time is

$$\mathcal{O}(\sum_{i=1}^{r} 2^{|X_i|}|X_i|) = \mathcal{O}(2^k \cdot k \cdot n)$$

Chapter 3

Component Order Connectivity Problem

3.1 Problem

Connectivity is one of the main properties of graphs. Restricting the size of each connected component of a graph comes up in various situations. One example could be the case of computer networks. Each system can be thought of as a node and connected systems are thought to have an edge between them. We aim to find the minimum number of systems to be removed from the network so that each connected component of the network is of limited size. Such measures would be useful when something like a virus attack in a network. The l-Component Order Connectivity or l-COC problem is to find the minimum number of vertices to be deleted from the graph so that each of the connected components of the graph is of size less than or equal to l.

l-COC
Input : A graph G
Question: What is the minimum size of a vertex subset S such that all the connected components of $G \setminus S$ are of size $\leq l$?

3.2 Examples

We use $\mu(G)$ to denote $l-\mathrm{COC}(G)$ in this section. The examples provided are mostly taken from [10].

Path: This is by removing every $(l+1)^{th}$ vertex from the graph.

$$\mu(P_n) = \lfloor \frac{n}{l+1} \rfloor$$

Cycle: This is by removing every $(l+1)^{th}$ vertex and also the last vertex if it is not already removed.

$$\mu(C_n) = \lceil \frac{n}{l+1} \rceil$$

Complete Graphs This is by removing all the vertices except any l vertices from the graph.

$$\mu(K_n) = n - l$$

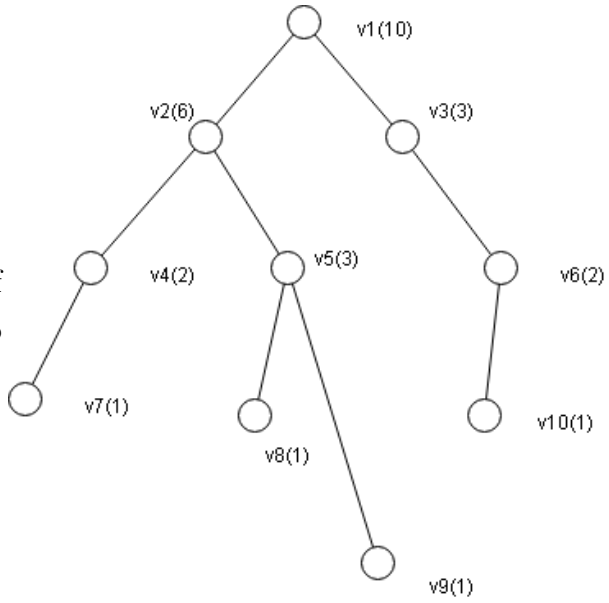
Star Graph: This is by removing only the central vertex which leaves $n-1$ isolated vertices.

$$\mu(K_{1,n}) = 1$$

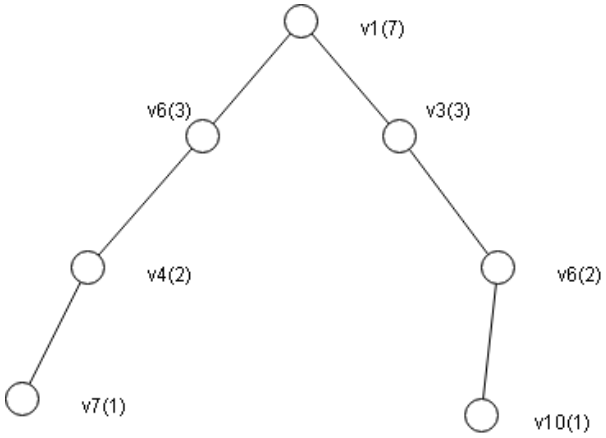
Trees

- First we define a function $c : V(G) \to n$ for every vertex in G starting from leaf vertex towards the root. This is given by $c(v) = |T_v(G)|$ where $T_v(G)$ is the subtree with v as root.
- Now, we go from any leaf of the tree towards the root and find the first vertex v with $c(v) > l$ and remove that vertex from the graph and add it to the solution set.
- We also remove all components with $\leq l$ vertices and then repeat the procedure until no vertex is left.

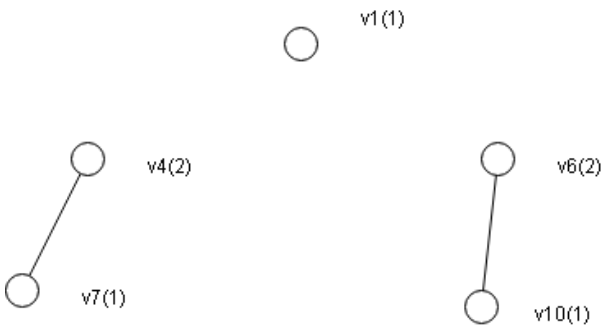
In the following example,
We have a tree with 10 vertices and $c(v)$ of each vertex is given in bracket. We need to find $2 - COC(T)$.



First we find a vertex with $c(v) > 3$ and we remove it's subtree. v2 is such a vertex. Removing vertex v2 will give us new tree with new $c(v)$ values.



Now, we have two vertices, $v6$ and $v3$ with $c(v) = 3$. As these two are in different branches, we can remove them simultaneously. By doing so, we get the next graph which obeys 2 component order connectivity. So, {v5, v3, v6} is the required set of vertices and $3 - COC(T) = 3$.



3.3 ILP Formulation

$$
\begin{aligned}
\text{Maximize} \quad & \sum_{i \in V} y_i \\
\text{subject to} \quad & x_{ij} + x_{ik} - x_{jk} \leq 1 && i, j, k \text{ are distinct in V} \\
& \sum_{j \in V \setminus \{i\}} x_{ij} \leq l - 1 && \forall i \in V \\
& y_i + y_j - x_{ij} \leq 1 && \forall \{i, j\} \in E \\
& x_{ij} \in \{0, 1\} \\
& y_i \in \{0, 1\}
\end{aligned}
$$

In the problem, our aim is to maximise the number of vertices in the graph while following component order connectivity, i.e. each component in the graph should have $\leq l$ vertices. More details on ILP formulation can be found at [11] and details about the given ILP formulation can be found at [12].

- Each vertex is taken as a variable $y_i \in \{0, 1\}$ according to whether the vertex i is in the final graph or not. Another variable x_{ij} takes into consideration whether the vertices i and j are connected in the final graph.
- The aim of the ILP is to maximise the number of vertices in the final graph or maximise $\sum_{i \in V} y_i$.
- The first constraint is the one about connectivity which says that if i and j are connected and j and k are connected, then i and k are connected.
- The second constraint says that each component is of size $\leq l$ by saying each vertex is connected to at most $l - 1$ another vertices.
- The third constraint says that if two adjacent vertices are present in the final graph, they should belong to the same component.

3.4 3-COC Algorithm for Bounded Degree Graphs

We design a simple branching algorithm for $3 - COC$ problem on bounded degree graphs. We start from the lower degree vertices and define branching rules to such vertices and then

move on to the higher degree vertices. We select the vertex with minimum degree, say v and do the following branching:

- $v \in S$**:** We put v in solution, thereby removing v from the graph. Here we get a drop of 1.
- $v \in C_1$**:** Here, we branch on the possibility that v belongs to a component of size 1 at the end of the algorithm. So, we remove v and all it's neighbours from the graph. We get a drop of $1 + d(v)$.
- $v \in C_2$**:** Here, we branch that v belongs to a component of size 2 at the end of the algorithm. In the worst case we can have $d(v)$ branches here each having a drop of around $2d(v)$.
- $v \in C_3$**:** Here, we branch that v belongs to a component of size 3 at the end of the algorithm. In the worst case, we can have $d(v)(d(v) - 1)$ branches each having a drop of $2d(v) + 1$

This drop gives a recurrence relation of

$$x^{2d+1} - x^d - dx - d(d-1) = 0$$

So, for a bounded degree graph $3 - COC$ problem with degree bound say 5, the worst case running time is obtained by putting $d = 5$ for the above equation. For $d = 5$ we get $x = 1.369$ which corresponds to running time of $\mathcal{O}(1.369^n)$.

3.5 2-COC or Dissociation Problem

$2 - COC$ problem is removing some vertices and dividing the remaining graph into components of size 2 and 1. This is given in the literature as a 3-path vertex cover problem or dissociation problem. k-Path Vertex Cover problem is to find the minimum size of the vertex subset that hits all the paths of length k in the given graph. When it comes to the case where $k = 3$, this is equivalent to our problem. More details and algorithms can be found in [13]. Here, they use a branching algorithm with several branching rules for vertices with different degrees. By using these complex branching rules, they get a worst-case running time

of $\mathcal{O}^*(1.51^n)$. The NP-completeness and polynomial-time solvable cases for the dissociation set problem can be found in [14].

3.6 Edge Version

l-COEC
Input: A graph G
Question: What is the minimum size of an edge subset S such that all the connected components of $G \setminus S$ are of size $\leq l$?

In the edge version of the problem, we need to delete the minimum number of edges from the graph, so that the resulting graph has all its components of limited size. This is a well-studied problem both from exact exponential perspective and a parameterized perspective.

We use $\lambda(G)$ to denote $l - \text{COEC}(G)$ in this section. The examples provided are mostly taken from [10].

Path: This is by removing edge after every l^{th} edge.

$$\lambda(P_n) = \lfloor \frac{n-1}{l} \rfloor$$

Cycle: This is by removing every $(l)^{th}$ edge and also the last edge if it is not already removed.

$$\lambda(C_n) = \lceil \frac{n}{l} \rceil$$

Complete Graphs This is by removing all the vertices except any l vertices from the graph.

$$\mu(K_n) = n - l$$

Star Graph: This is by removing all edges except any $l - 1$ edges.

$$\mu(K_{1,n}) = n - l$$

3.7 Parameterized Version

- In Parameterized algorithms, instead of expressing running time as a function $T(n)$ of n, we express it as a function $T(n, k)$ of the input size n and some parameter k of the input.
- In other words, we do not want to be efficient on all inputs of size n, only for those k is small.
- **FPT**: A parameterized problem is fixed-parameter tractable if there is an $f(k)n^c$ time algorithm for some constant c.

Kernalization: Kernelization is a polynomial time transformation that maps an instance (I, k) to an instance (I', k') such that:

- (I, k) is an yes instance if and only if (I', k') is an yes instance.
- $k' \leq k$.
- $|I'| \leq f(k)$ for some function $f(k)$

For example, the vertex cover problem has a kernel of $k(k+1)$ vertices when parameterized by the solution size k. The parameterized verion of l-**COC** and l-**COEC** problem is given below:

l-**COC** Input : A graph G and an integer k Question: Does there exist a vertex subset $S \subseteq V$ such that $\|S\| \leq k$ and $G \setminus S$ has all components with size $\leq l$.

l-**COEC** Input : A graph G and an integer k. Question: Does there exist an edge subset $S' \subseteq E$ such that $\|S'\| \leq k$ and $G \setminus S$ has all components with size $\leq l$.

Theorem 3. *Parameterised version of* $l -$ Component order connectivity *is solvable in* $\mathcal{O}^*(l+1)^k$

Proof: This is obtained by taking any connected component of size $= l+1$ and branching on the case that at least one of the vertex from this connected component should be in the solution. This creates $(l+1)$ branches and the depth of the search tree should be less than or equal to k as this is the solution size. Thereby we get an algorithm with running time $\mathcal{O}^*(l+1)^k$.

Theorem 4. *For every constant* l, $l -$ Component order connectivity *admits a kernel with atmost* $2lk$ *vertices that takes* $n^{\mathcal{O}(l)}$ *time.[15]*

Theorem 5. $l -$ Component order edge connectivity *is fixed parameter tractable (*FPT*) when parameterized by vertex cover number of the input graph. It also admits a kernel with* $2kl$ *vertices and* $2kl^2 + k$ *edges when parameterized by* $k + l$*.[16]*

Chapter 4

Conclusion

In the first part of the thesis, we studied different methods to design exact exponential algorithms to solve NP hard problems. This is done by reading the book, "Exact Exponential Algorithms" by Fomin and Krastch. We mainly focused on Branching, Dynamic Programming, Measure and Conquer and Treewidth. We learned when and where can we use these methods and tried to implement these techniques on different problems.
The problems we mainly deal with in this thesis is Component order connectivity and the edge version of the same problem. An introduction to the problem using different examples and ILP formulation is given. Various algorithms on these problems can be found in the literature in the name of path vertex cover or dissociation set problem. This includes exact algorithms and also some parameterized algorithms. Some exact algorithms are provided and also an intuition into how to solve different problems using branching is also shown. The parameterized version and some results are also given.

Bibliography

[1] R. Diestel, *Graph Theory, 4th Edition*, ser. Graduate texts in mathematics. Springer, 2012, vol. 173.

[2] J. Kleinberg and E. Tardos, *Algorithm design*. Pearson Education India, 2006.

[3] S. Arora and B. Barak, *Computational complexity: a modern approach*. Cambridge University Press, 2009.

[4] F. V. Fomin and P. Kaski, "Exact exponential algorithms," *Communications of the ACM*, vol. 56, no. 3, pp. 80–88, 2013.

[5] M. Xiao and H. Nagamochi, "Exact algorithms for maximum independent set," *Information and Computation*, vol. 255, pp. 126–146, 2017.

[6] D. W. Matula, G. Marble, and J. D. Isaacson, "Graph coloring algorithms," in *Graph theory and computing*. Elsevier, 1972, pp. 109–122.

[7] J. M. Van Rooij and H. L. Bodlaender, "Exact algorithms for dominating set," *Discrete Applied Mathematics*, vol. 159, no. 17, pp. 2147–2164, 2011.

[8] F. V. Fomin, F. Grandoni, and D. Kratsch, "Measure and conquer: A simple o (20. 288 n) independent set algorithm," in *SODA*, vol. 6, 2006, pp. 18–25.

[9] H. L. Bodlaender, "Treewidth: Algorithmic techniques and results," in *International Symposium on Mathematical Foundations of Computer Science*. Springer, 1997, pp. 19–36.

[10] D. Gross, M. Heinig, L. Iswara, L. W. Kazmierczak, K. Luttrell, J. T. Saccoman, and C. Suffel, "A survey of component order connectivity models of graph theoretic networks," *WSEAS Transactions on Mathematics*, vol. 12, no. 895-910, p. 2, 2013.

[11] A. Schrijver, *Theory of linear and integer programming*. John Wiley & Sons, 1998.

[12] W. Ben-Ameur, M.-A. Mohamed-Sidi, and J. Neto, "The k k-separator problem: polyhedra, complexity and approximation results," *Journal of Combinatorial Optimization*, vol. 29, pp. 276–307, 2015.

[13] B. Brešar, F. Kardoš, J. Katrenič, and G. Semanišin, "Minimum k-path vertex cover," *Discrete Applied Mathematics*, vol. 159, no. 12, pp. 1189–1195, 2011.

[14] Y. Orlovich, A. Dolgui, G. Finke, V. Gordon, and F. Werner, "The complexity of dissociation set problems in graphs," *Discrete Applied Mathematics*, vol. 159, no. 13, pp. 1352–1366, 2011.

[15] M. Kumar and D. Lokshtanov, "A 2\ell k $kernel for \ell - component order connectivity$," *arXiv preprint arXiv:1610.04711*, 2016.

[16] A. Gaikwad and S. Maity, "Further parameterized algorithms for the f-free edge deletion problem," *Theoretical Computer Science*, vol. 933, pp. 125–137, 2022.