NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCE
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

BSc THESIS

The Maximum Independent Set Problem

Ioannis A. Papatsoris

Supervisor: **Panagiotis Stamatopoulos,** Assistant Professor

ATHENS

OCTOBER 2018

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Το Πρόβλημα του Μέγιστου Ανεξάρτητου Συνόλου

Ιωάννης Α. Παπατσώρης

Επιβλέπων: Παναγιώτης Σταματόπουλος, Επίκουρος Καθηγητής

ΑΘΗΝΑ

ΟΚΤΩΒΡΙΟΣ 2018

BSc THESIS

The Maximum Independent Set Problem

Ioannis A. Papatsoris
R.N.: 1115201300137

SUPERVISOR: **Panagiotis Stamatopoulos,** Assistant Professor

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Το Πρόβλημα του Μέγιστου Ανεξάρτητου Συνόλου

Ιωάννης Α. Παπατσώρης
Α.Μ.: 1115201300137

ΕΠΙΒΛΕΠΩΝ: **Παναγιώτης Σταματόπουλος,** Επίκουρος Καθηγητής

ABSTRACT

The maximum independent set problem, to find a maximum set of vertices in a graph such that there is no edge between any two vertices in the set, is one of the basic NP-hard optimization problems and has been extensively studied within the literature, in particular in the line of research on worst-case analysis of algorithms for NP-hard optimization problems. The best currently known running time complexity algorithm utilizes a branch-and-reduce paradigm, by applying several critical reductions on the graph to reduce its size, and solving smaller instances independently, to form a solution for the initial problem. The purpose of this thesis is to study the algorithm and implement an efficient program for computing the Maximum Independent Set on large graphs.

SUBJECT AREA: Graph Theory

KEYWORDS: independent set, exact algorithm, graph, branch and reduce, optimization problem

ΠΕΡΙΛΗΨΗ

Το πρόβλημα του μέγιστου ανεξάρτητου συνόλου, που πρόκειται για την εύρεση ενός μέγιστου συνόλου κόμβων ενός γράφου δεδομένων, τέτοιο ώστε να μην υπάρχει καμία ακμή μεταξύ οποιονδήποτε δυο κόμβων εντός του συνόλου, είναι ένα από τα βασικά NP-hard προβλήματα βελτιστοποίησης και έχει ερευνηθεί εκτενώς εντός της βιβλιογραφίας, συγκεκριμένα στο ερευνητικό πεδίο της ανάλυσης της χείριστης περίπτωσης αλγορίθμων για NP-hard προβλήματα βελτιστοποίησης. Ο τρέχων καλύτερος αλγόριθμος όσον αφορά χρονική πολυπλοκότητα ακολουθεί μια branch-and-reduce τακτική, εφαρμόζοντας ορισμένες ελαττώσεις στον γράφο για να μειώσει το μέγεθός του, λύνοντας τα μικρότερα στιγμιότυπα ανεξάρτητα, ώστε να δημιουργήσει τελικά τη λύση του αρχικού προβλήματος. Ο σκοπός αυτής της πτυχιακής εργασίας είναι η μελέτη του αλγορίθμου, και η υλοποίηση ενός προγράμματος που να επιλύει το πρόβλημα αποδοτικά.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Θεωρία Γράφων

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: ανεξάρτητο σύνολο, ακριβής αλγόριθμος, γράφος δεδομένων, branch and reduce, πρόβλημα βελτιστοποίησης

To my family.

ACKNOWLEDGMENTS

I would like to thank my supervisor Prof. Panagiotis Stamatopoulos for providing his support and guidance on shaping and successfully completing this thesis.

October 2018

CONTENTS

PREFACE 11

1. INTRODUCTION 12

2. RELATED WORK 13

2.1 Exact solutions 13

2.2 Linear time 13

2.3 Memory efficient 14

3. THE ALGORITHM 15

3.1 Description 15

3.2 Implementation 17

4. PROGRAM USAGE 21

4.1 Execution 21

4.2 Input 21

4.3 Output 22

5. EXPERIMENTAL EVALUATION 23

6. CONCLUSIONS AND FUTURE WORK 27

ACRONYMS AND ABBREVIATIONS 28

ANNEX 29

REFERENCES 31

LIST OF FIGURES

Figure 1: Maximum Independent Set algorithm 16

Figure 2: A graph with the maximum independent set of it highlighted 16

Figure 3: EdgeBuffer for graph on Figure 2 17

Figure 4: NodeIndex mapping for graph on Figure 2 17

Figure 5: Expand *node*'s neighbors with *newNeighbors* algorithm 18

Figure 6: Find a maximum independent set within a line graph reduction algorithm 20

Figure 7: Program execution commands 21

Figure 8: Execution time for 500 vertices and varying number of edges . . . 23

Figure 9: Execution time for 1000 vertices and varying number of edges . . . 24

Figure 10: Execution time for 1 million vertices and varying number of edges . 24

Figure 11: Memory in megabytes for 500 vertices and varying number of edges 25

Figure 12: Memory in megabytes for 1 million vertices and varying number of edges 25

Figure 13: Size of maximum independent set for 500 vertices and varying number of edges 26

Figure 14: Traversing through all the active edges of the graph 29

Figure 15: Reductions step for graph instances of max degree 3 29

Figure 16: Input file for the graph on Figure 2 30

Figure 17: Graph generation example using SNAP 30

PREFACE

This project was developed in Athens, Greece between the months January and October of the year 2018. At its initial stages, research was made to find and evaluate current literature related to the problem. After the publication on which the thesis program implementation would be based on was established, it was important to thoroughly study it and understand it. This would lead to the next step, which was implementing an efficient program, according to the algorithm described in the publication. Finally, experiments were performed using it, and this presentation of the work was written.

1. INTRODUCTION

In the context of graph theory, an independent set is a set of vertices in a graph, such that no two of which are adjacent. That is, each edge in the graph has at most one endpoint in the set. An independent set is classified as maximal, when adding any other vertex to the set introduces an edge between its vertices. A graph may have many maximal independent sets of different sizes; the largest of which is referred to as the maximum independent set, and its size represents the independence number of the graph. The maximum independent set of a graph is also not necessarily unique.

While the Maximum Independent Set problem is closely related to a number of fundamental graph problems, such as maximum common induced subgraphs, minimum vertex covers, graph coloring, etc, it also holds of great significance to numerous real-world application, outside of graph theory. Some of these include indexing techniques for shortest path and distance queries, automated labeling of maps, social network analysis, information coding, and more.

The problem of finding the maximum independent set of graph is an NP-Hard optimization problem, thus it is unlikely that there exists an efficient algorithm for an exact solution to it. In this thesis, we briefly present the currently best worst-case complexity algorithm that exists in the literature, and implement a program for computing the maximum independent set, according to that algorithm.

This thesis is based on the algorithm of the publication Exact Algorithms for Maximum Independent Set [1]. That publication uses work presented in 3 more publications ([2], [3], [4]) by the same authors, although, the core logic of the algorithm is the same between all of them.

The rest of the thesis is organized as follows:

1. In Chapter 2 we give an overview of the research that has been done in the literature about the Maximum Independent Set.
2. In Chapter 3 we describe the algorithm we used for solving the problem, and we demonstrate program implementation remarks.
3. In Chapter 5 we present our experimental evaluation.
4. In Chapter 6 we summarize our conclusions.

2. RELATED WORK

2.1 Exact solutions

The Maximum Independent Set is one of the basic NP-hard optimization problems, and thus has been extensively studied within the literature. A trivial brute force algorithm that computes all possible subsets of the graph, checks whether their independent or not, and picks the maximum one, would require $O(2^n n^2)$-time. However, the problem can be solved faster. The first nontrivial exact algorithm for Maximum Independent Set dates back to Tarjan and Trojanowski's $O(2^{n/3} n^{O(1)})$-time algorithm in 1977 [5]. However, despite of a large number of contributions on exact algorithms and their worst-case analyses for Maximum Independent Set during the last 30 years, no published algorithm ran faster than the $O(1.2109^n n^{O(1)})$-time exponential-space algorithm by Robson in 1986 [6], Until the algorithm which this thesis focuses on was proposed, which runs in $O(1.1996^n n^{O(1)})$-time.

Tarjan and Trojanowski's algorithm uses a recursive, or backtracking scheme, and depends upon a somewhat complicated case analysis. In summary, it selects a set of vertices, determines a set of dominating independent sets in it, and recursively solves one subproblem for each dominating set. Robson's algorithm studies cases in the neighborhood of a chosen vertex, and removes a lot of awkward cases by discounting low degree regular graphs.

2.2 Linear time

While great effort has been devoted on research on finding an exact solution to the Maximum Independent Set, due to the hardness of such a task, many algorithms resort to heuristic techniques, and instead opt in to finding an approximate solution in a significantly shorter amount of time. The goal in that case is to develop an algorithm which maintains the right balance between performance and solution accuracy. A recent publication [7] proposes fast linear and near-linear time complexity algorithms, that can efficiently generate a high quality independent set from a graph in practice.

To begin with, a Reducing-Peeling baseline framework is utilized, which iteratively reduces the graph size by applying reduction rules on vertices with very low degrees (Reducing), and temporarily removing the vertex with the highest degree (Peeling), if the reduction rules cannot be applied. Based on this framework, algorithms BDOne and BDTwo are introduced, which can generate high quality independent sets by making use of existing reduction rules for handling degree-one and degree-two vertices, respectively. Finally, a linear-time algorithm, LinearTime, and a near-linear time algorithm, NearLinear, are proposed by designing new reduction rules and developing techniques for efficiently and incrementally applying those. In practice, LinearTime takes similar time and space to BDOne

but computes a higher quality independent set, similar in size to that of an independent set generated by BDTwo, whereas NearLinear has a good chance to generate a maximum independent set, and it often generates near-maximum independent sets. These algorithms are efficient and able to generate much larger independent sets than other existing linear-time algorithms, while having a similar running time.

2.3 Memory efficient

Even though such exact or approximate algorithms are successful to a certain degree, they require memory space at least linear in the size of the input graph. This requirement, however, is often unrealistic for large graphs (e.g., social networks, web graphs) commonly seen in modern applications. An I/O efficient semi-external algorithm [8] addresses this issue, with a greedy approach that utilizes very limited main memory. According to the semi-external setting, it is assumed that the main memory can accommodate all vertices of the graph, but not all edges.

Firstly, a semi-external greedy algorithm which constructs an independent set is utilized, by performing exactly one sequential scan of the disk file of the input graph , avoiding expensive random disc access. The high-level idea of it is to repeatedly add vertices with small degrees to the independent set if none of their neighbours are already in it, until no more vertices can be added. Afterwards, the One-k-Swap algorithm is used, which takes the independent set computed earlier from the greedy algorithm, and swaps vertices to enlarge it. The challenges here under the semi-external setting are to efficiently decide whether a swap operation can be performed correctly to guarantee an independent set, and to develop a mechanism that allows only one of two swaps to be performed, between two swaps that are correct on their own, but conflict when used together. The procedure runs iteratively until no more swaps can be performed. The One-k-Swap algorithm can be extended to Two-k-Swap, which swaps two vertices for three or more at a time, to provide a better solution to the problem. All of these procedures compute an independent set, which is closely to the theoretical optimal bound for massive real-life graphs, using very limited main memory.

3. THE ALGORITHM

3.1 Description

At its core, the algorithm follows a branch and reduce paradigm, typically reducing the graph to two different smaller instances on each step, and solving those first independently. This iterative process creates a search tree, with each search node representing a reduced instance of the graph. After enough branching is performed, we reach a trivial graph instance, on which a maximum independent set can be easily computed. The solution of said trivial instance is stored on the leaves of the search tree, and is used to recursively provide a solution to the parents. The solution to the root node is also the solution to the original graph instance.

The most common way of branching is called 'branching on a vertex', and is to either include a vertex in the solution set, or exclude it. In the former case, the vertex is removed from the graph, along with all its neighbors. Since they are adjacent to a node inside the independent set, they cannot be in the solution set as well. In the latter case, the vertex is simply removed, reducing the size of the graph by 1. It is proven by the authors of the algorithm that the largest maximum independent set between those two reduced instances of the graph, is also the maximum independent set of the original graph.

Although branching on vertices is the most typical scenario of branching, there are a few other methods that are utilized as well. Such methods include branching on edges, branching on a complete bipartite subgraph, branching on a funnel, and branching on a 4-cycle. Choosing the right branching rule, or the most suitable vertex to branch on depends on the properties and form of the current graph instance, and is a crucial part for the performance of the algorithm. It massively affects the size of the solution search tree, as sophisticated branching can prune many instances. Details on how this decision is made, and a full description of the rest of the branching rules, can be found in the algorithm publication.

There is a special case involving vertex cuts. A partition (V_1, Z, V_2) of the vertex set $V(G)$ of a graph G is called a separation, if $V(G)$ is a disjoint union of nonempty subsets V_1, Z and V_2, and there is no edge between V_1 and V_2, where Z is called a vertex cut. At some point in the branching decision part of the algorithm, the graph is searched for minimal vertex cuts of size 1, and then 2. If at least one is found, the maximum independent set search is split between the two subgraphs induced by removing the vertex cut. A maximum independent set is found in both of those, and those maximum independent sets together with the vertex cut, form the maximum independent set of the original graph. Vertex cuts of size 3 also need to be detected in another part of the algorithm.

To reduce the size of that search tree, good branching rules are required. Although, before these branching rules are applied, certain reduction rules are performed to reduce some local structures of the graph, branching on which may lead to a bad performance. Such reduction rules can be applied in polynomial time to either find a part of the solution, or

decrease the size of the instance directly. A high level represtation of the algorithm is shown on Figure 1.

1: **procedure** MIS(G)
2: Reduce local structures on G, resulting in G^*
3: Apply a branching rule on G^*, creating subgraphs G_1 and G_2
4: **return** max(MIS(G_1), MIS(G_2))

Figure 1: Maximum Independent Set algorithm

Here is a list of the reductions applied. Detailed description of each can be found in the publication.

- Eliminating easy instances
- Removing unconfined vertices
- Removing dominated vertices
- Removing line graphs
- Folding twins
- Folding funnels
- Folding degree-2 vertices
- Folding 2-3 and 3-4 structures
- Folding complete k-independent sets
- Folding desks

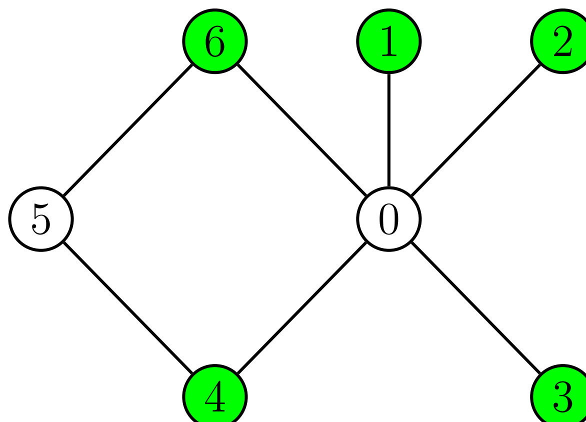

Figure 2: A graph with the maximum independent set of it highlighted

3.2 Implementation

In order to implement a performant program that executes the demanding task of finding the maximum independent set of a graph based on the algorithm described above, we need to utilize the right data structures. One of the most important factors is the internal representation of the graph.

In our program, each node is represented by its id, and the neighbors of a node are stored in a matrix, called EdgeBuffer. They are kept sorted, so as to allow the use of binary search for fast edge lookup. Instead of having an EdgeBuffer for each vertex, there is a global one for every vertex, and specific sections of it refer to the neighbors of a particular vertex. This is used to achieve spacial locality. According to spatial locality, placing data elements within relatively close storage locations can decrease their access time, and speed up the program.

In order to be able to use this global EdgeBuffer, we need to be able to identify which part of it belongs to which vertex. For this, we use a hash table which we will call NodeIndex, which maps vertex id to a data structure. That data structure holds the starting position in the EdgeBuffer segment that corresponds to this node's neighbors, the number of neighbors, and whether the node is removed or not. The purpose of the last property is explained below. We also hold the inverse, a mapping between an EdgeBuffer section starting position and its corresponding node id.

1 2 3 4 6	0	0	0	0 5	4 6	0 5

Figure 3: EdgeBuffer for graph on Figure 2

id	$\{pos, edges, removed\}$
0	$\{0, 5, false\}$
1	$\{5, 1, false\}$
2	$\{6, 1, false\}$
3	$\{7, 1, false\}$
4	$\{8, 2, false\}$
5	$\{10, 2, false\}$
6	$\{12, 2, false\}$

Figure 4: NodeIndex mapping for graph on Figure 2

During the reductions step of the algorithm, multiple nodes are potentially removed from the current graph instance, before the algorithm proceeds to the branching step. If we were to actually remove each node from the EdgeBuffer, it would be very inefficient, as each node removal would result in either a huge number of element shifts, or a complete reallocation of it from scratch. Instead, we mark the *removed* property of the nodes as *true*, and consider them as removed, even though they still remain in the EdgeBuffer. And after all the reductions are finally applied on the current graph instance, we then rebuild the EdgeBuffer and NodeIndex, by excluding the removed nodes. We do this so as to reduce

unnecessary steps on removed nodes during binary search for edge existence, and to increase spatial locality by decreasing the amount of memory required for the EdgeBuffer.

For certain operations such as branching on edges, folding funnels and desks, it is required to add new edges to the graph. Typically, we need to add a batch of new neighbors of a node at a time. Since the EdgeBuffer is big, we need to do this in a performant way, by avoiding shifting elements or reallocating, if possible. In order to do this, we take advantage of any inactive nodes that are marked as removed. The algorithm is described in Figure 5.

1: **procedure** addEdges(NodeIndex, EdgeBuffer, *node, newNeighbors*)
2: Calcuate the size of the EdgeBuffer section corresponding to the neighbors of *node*, store it in *availableSpace*
3: Store the removed neighbors of *node* to *removedNeighbors*
4: Add the non-removed neighbors of *node* to *newNeighbors*, while maintaining sorting
5: **if** $|newNeighbors| \leq availableSpace$ **then**
6: Neighbor addition can be done in place; fill EdgeBuffer with $availableSpace - |newNeighbors|$ number of nodes from *removedNeighbors*, then with the *newNeighbors*
7: **else**
8: Shifting is required; fill EdgeBuffer with *availableSpace* number of nodes from *newNeighbors*
9: Shift the remaining of the EdgeBuffer by the number of nodes in *newNeighbors* left to include, place those
10: Update the *pos* NodeIndex property of the nodes whose neighbors have been shifted
11: Update the number of edges of affected nodes in NodeIndex

Figure 5: Expand *node*'s neighbors with *newNeighbors* algorithm

As mentioned earlier, the core of the algorithm is recursively reducing the graph to two smaller instance, and finding a maximum independent set in those first, to form the one for the original graph. This process creates a search tree. In our implementation, the search tree of the different graph instances generated by branching, is represented as an array of SearchNode structures. Each SearchNode structure consists of a graph instance, a maximum independent set for it (initially empty), the type of branching rule to be applied on it, whether or not it contains a minimal vertex cut, and indexes to the array position of its parent, and its potential left and right children. The search tree is traversed in a depth-first search manner. First, reductions are applied on the graph, and the branching type is decided. Then the left child of it is generated, in which a reduced graph of the current one is assigned to. This goes on recursively, until the graph of a left child is trivial enough for a maximum independent set to be calculated on it. Afterwards, backtracking is performed to the parent of that trivial graph instance, and the right child of it is generated,

according to the branching rule that has been determined for it earlier on the first visit. This process repeats recursively, until backtracking reaches a SearchNode whose both children have had their maximum independent sets calculated. At that point, depending on whether there was an appropriate minimal vertex cut in that graph instance or not, the maximum independent set of it is either the largest set between the maximum independent set of its children, or the union of them in the case of a cut.

Getting the neighbors of a node is one of the most fundamental operations on a graph structure, and a very common requirement for graph algorithms. For performance reasons, many times in the program we prefer to iterate through the neighbors of a node one by one, and potentially stop iterating once a condition is met, instead of collecting them all at once in an array data structure, and then iterating the array. Since our data structure of choice for the graph is slightly complex, and the EdgeBuffer also includes nodes that may be potentially removed, we introduce a GraphTraversal structure, to simplify the process and keep our code clean. It consists of two fields: the current node being processed, and the position of its current neighbor being processed on the EdgeBuffer. At a GraphTraversal object construction, the current node is initialized as the first non removed node with at least one non removed neighbor, and its neighbor is also initialized as the first non removed one. The graph then exposes two functions: getNextNode and getNextEdge. They both mutate a GraphTraversal object and iterate through the nodes of the graph as their respective names suggest, ignoring removed or zero degree nodes. They encapsulate the code for traversing through the data structures, and simplify it down to simple function calls. An example of the simplicity of traversing through all the active edges of a graph can be seen on Figure 14.

When it comes to identifying and applying reductions on a graph instance before proceeding to the branching step, we need to keep in mind that even a single mutation in the graph, demands that we check for all the reductions again from scratch, even for ones we've already checked. This can be achieved by nesting the reduction calls in loops. An example for this is shown on Figure 15, which is a simplified version of the reductions code for graph instances of max degree 3.

If an easy instance or a line graph reduction is applicable, a maximum independent set is calculated on it on the spot. The publication does not specify how. We use brute force for an easy instance, by calculating all possible independent sets within it and keeping the maximum one, but a maximum independent set can be found much faster within a line graph reduction. Due to the structure of the graphs applicable for that reduction, we can just pick arbitrary nodes to include in the set until we can no longer pick any, and we will always result in a maximum independent set, regardless of our initial or subsequent choices. A simple linear algorithm for it is shown on Figure 6.

Certain other operations, like folding complete k-sets, folding twins and folding degree-2 vertices, require to contract some nodes into a new one. The new node is called a

```
1: procedure findMisInLineGraphReduction
2:     while there are nodes do
3:         Insert an arbitrary node in the maximum independent set
4:         Remove that node from the graph
5:         Remove the neighbors of it from the graph
```

Figure 6: Find a maximum independent set within a line graph reduction algorithm

Hypernode, and although the old nodes that formed it are now marked as removed, we still need to keep track of them. This is because, depending on whether the Hypernode will be included in the maximum independent set or not, we will need to decompose it and choose the correct inner nodes of it to include, since we want our solution set to be referring to the original graph, which had no Hypernodes. When decomposing a Hypernode and choosing to include some of its inner nodes in the maximum independent set, some of those might be Hypernodes themselves, so decomposition must be performed recursively.

Regarding identifying minimum vertex cuts, the publication does not specify a specific algorithm. We implement Tarjan's algorithms [9, 10] for vertex cuts of size 1 and 2, which are based on depth-first search and run in $O(n+m)$ time, where n and m are the number of vertices and edges of the graph respectively. Efficient algorithms for spotting vertex cuts of size 3 [11] require substantially extra time and effort to implement, thus we use a $O(n^3)$-time brute force approach, by generating all vertex triplets of the graph and checking whether they form a separation.

4. PROGRAM USAGE

4.1 Execution

The program implementation can be found on Github [13]. It is written in C++ under the C++11 standard, and can be compiled using the included Makefile, with the command *make*.

It can be ran in the following ways:

```
1        ./mis <input_graph>
2        ./mis <input_graph> -check
```

Figure 7: Program execution commands

Replace *<input_graph>* with the input file location. The first run option executes the algorithm, and outputs the maximum independent set to a new file named *<input_graph>.mis*. The second one verifies whether the vertex set at *<input_graph>.mis* is an independent set or not, according to the respective graph at *<input_graph>*.

4.2 Input

The program is designed to easily work with graphs generated by the graph processing library SNAP [12], and so the input format is based on the one used by SNAP. The first two lines are ignored, and the third one contains the number of nodes x, and the number of edges y, as follows: # *Nodes: x Edges: y*. Then on the next line follows a list of edges, one edge on each line, according to the format below:

$$\begin{matrix} X_1 & Y_1 \\ X_2 & Y_2 \\ \vdots \\ X_n & Y_n \end{matrix}$$

This indicates that there is an edge between nodes X_1 and Y_1, X_2 and Y_2, etc. The nodes on the left column are sorted in ascending order, the nodes on the right column that correspond to a specific node on the left are also locally sorted in ascending order, and are larger than the corresponding left column node. Any nodes that belong within the total node count but don't appear on the list, are considered zero degree nodes. An example of an input file using this format is shown on Figure 16.

A graph that uses this format can be easily generated with SNAP, by using a command like the one on Figure 17.

4.3 Output

At its core, the algorithm follows a branch and reduce paradigm, typically reducing the graph to two different smaller instances, and solving those first independently, creating a search tree. During execution, the program prints on the screen several numbers in descending order. These correspond to the id of the closest to the root problem instance that has been solved. When the instance with id 0 (root) is solved, a solution to the initial problem is found, thus the program terminates. This is a useful indication for predicting the time remaining for instances that may take a long time to solve.

The maximum independent set size is also displayed. As mentioned earlier too, the actual content of the maximum independent set is placed in *<input_graph>.mis*.

5. EXPERIMENTAL EVALUATION

In this section, experiments are performed and evaluated using the program. These include the running times on graphs of varying number of nodes and edges, as well as the size of the maximum independent set. All experiments are performed on a 64-bit Intel® Core™ i7-7500U CPU @ 2.70GHz / 3.50Ghz Turbo × 4 machine with 16GB memory, running under Ubuntu 18.04.1 LTS.

To begin, we run the program on a graph of 500 nodes and 1000 edges, and gradually increase the number of edges to inspect how it scales. In Figure 8, we observe that the program runs instantly for 1200 edges or less, and suddenly substantially slows down for more. However, more edges do not necessarily mean a slower running time, as the shape and properties of the graph are of great importance. Thus, we do not always see an increase in running time as we add more edges.

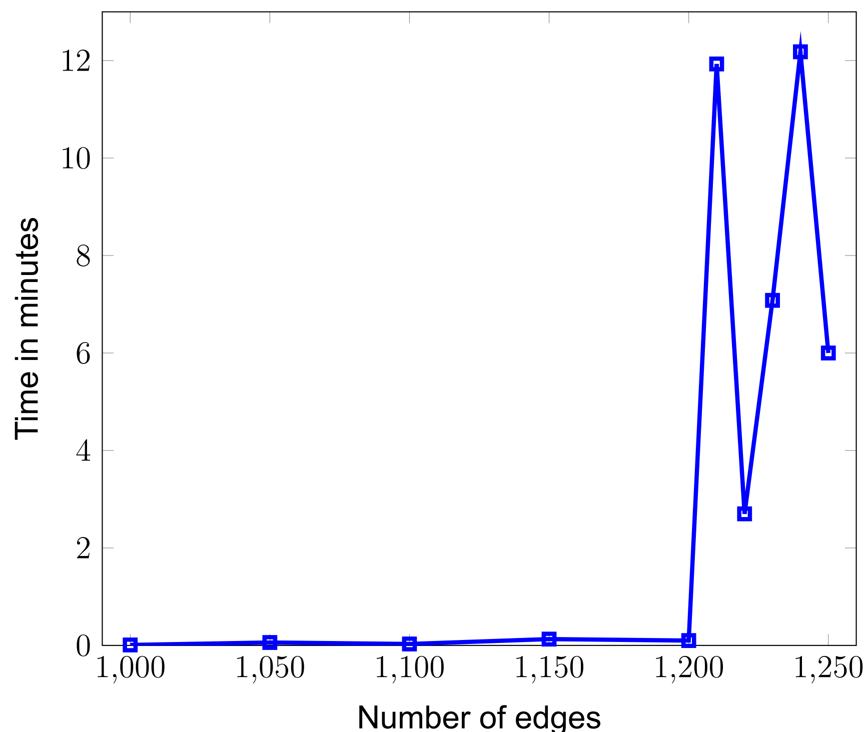

Figure 8: Execution time for 500 vertices and varying number of edges

In Figure 9, we up the number of nodes to 1000, and gradually increase the number of edges in a similar manner. Again, with see that the running time is as short as instant or less than 5 minutes, with a steep leap to around 160 and 20 minutes.

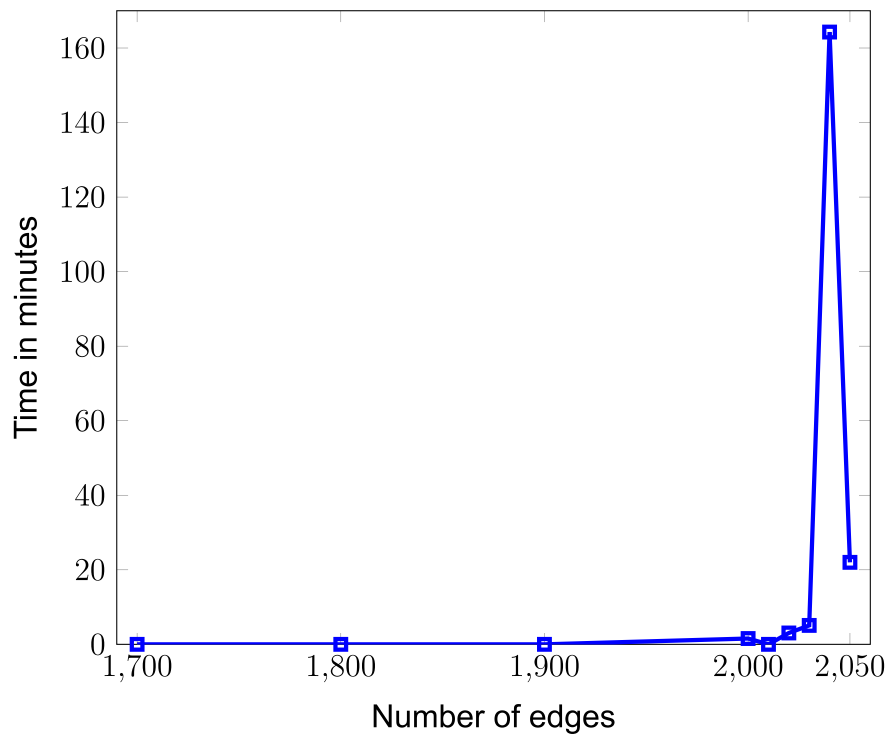

Figure 9: Execution time for 1000 vertices and varying number of edges

In Figure 10, we aim to inspect how the number of nodes affects performance, by testing the program on a graph of 1 million nodes. Again, we observe very fast running times with the right amount of edges, until we add 1380000 edges, where we reach 14 minutes. We attempted 1390000 edges as well, and the running time was estimated to be more than 5 hours.

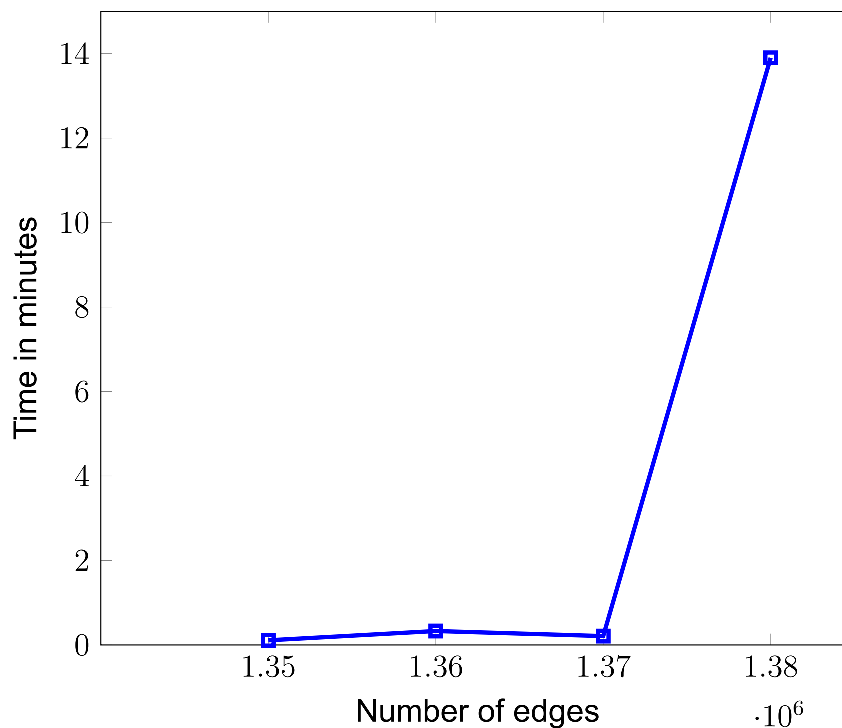

Figure 10: Execution time for 1 million vertices and varying number of edges

Regarding the memory usage of the program, Figures 11 and 12 show the memory consumed for graphs of 500 vertices and 1 million vertices respectively, for varying number of edges. We observe that memory increases dramatically and suddenly, very similarly to the behaviour seen in the time performance plots. Memory usage is expected to be high, since each SearchNode in the search tree consists of a whole graph instance. Although, since we're utilizing depth-first search, we only need to keep one SearchNode for each depth level of the tree. So the maximum memory used is bounded by the depth of the search tree, and not the total number of SearchNodes.

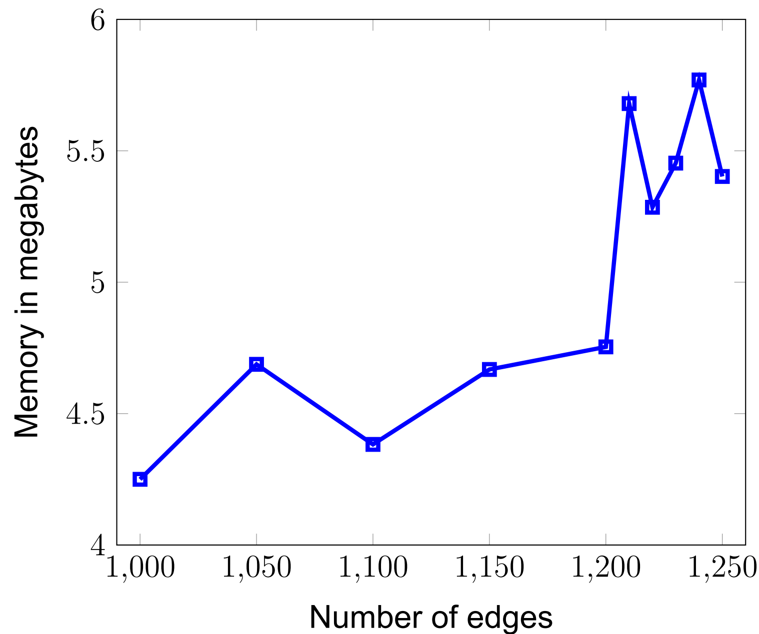

Figure 11: Memory in megabytes for 500 vertices and varying number of edges

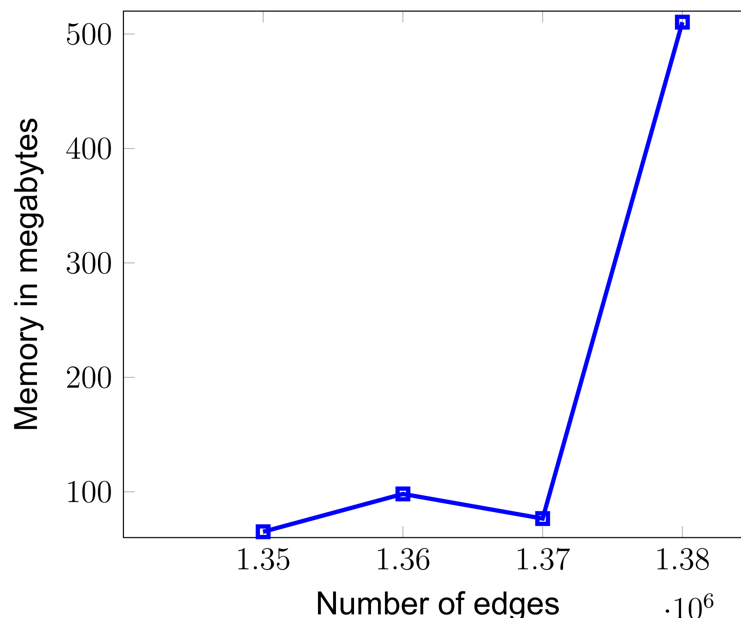

Figure 12: Memory in megabytes for 1 million vertices and varying number of edges

From the above time and memory experiments, we conclude that our program runs fast in sparse graphs, regardless of the number of nodes. However, even the slight increase in edges can quickly result in much longer times and memory usage. Time and memory are expected to be proportional, since the bigger the search tree is, the longer it will take for a solution to be found, and more memory will be required to hold all those graph instances. An exception to this rule is complete or near-complete graphs, on which the program can run very fast, due to the reductions step of the algorithm. For instance, a complete graph of 10000 nodes is solved in 8 seconds.

Finally, in Figure 13 we check the size of the maximum independent set for graphs of 500 nodes. As expected, we see that adding edges results either in a decrease in its size or in no change, since more edges constrain the problem further more. Although, we do not use the same graph and add edges on it, instead we generate a new graph every time. This explains why at some point we see an increase in the maximum independent size, instead of a decrease or no change.

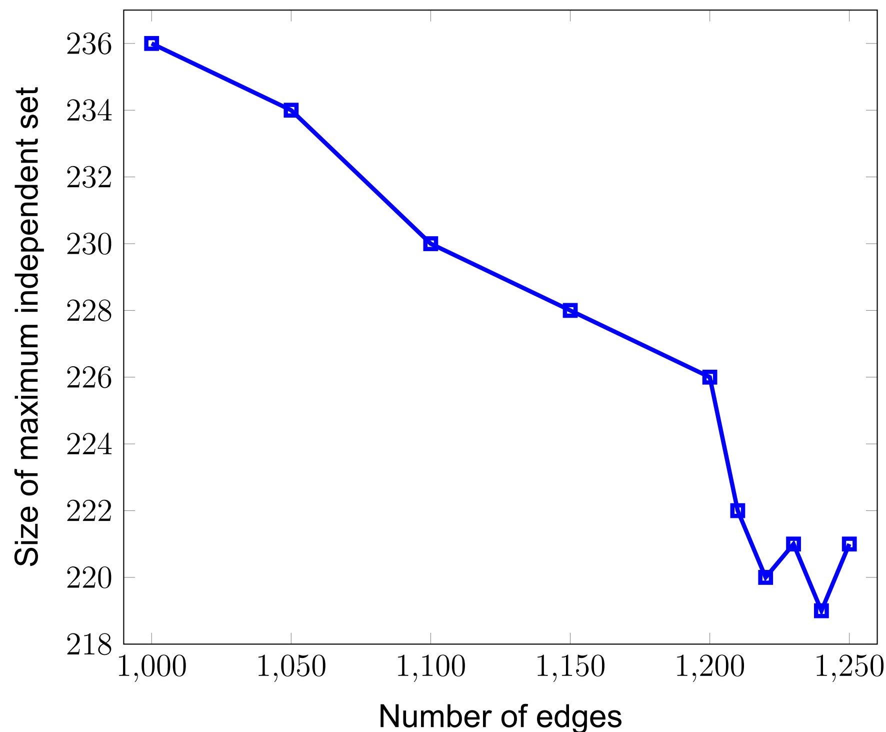

Figure 13: Size of maximum independent set for 500 vertices and varying number of edges

6. CONCLUSIONS AND FUTURE WORK

Exponential algorithms for NP-hard problems have been extensively studied throughout the last decades. The Maximum Independent Set problem is undoubtedly one of the most fundamental problems in that regard. In this thesis, we implemented an efficient program to provide a solution to the problem, based on the currently best time complexity algorithm. We described how we approached implementing an efficient graph data structure suited for such an algorithm, and mentioned several key points of interest, regarding the conversion of the algorithm from pseudo code to an actual program. From our experimental evaluation we observed that our program's performance drops steeply as the number of edges increases, proving the hardness of the problem and the algorithm's exponential nature. The density of the graph is of much more importance than its number of nodes, when it comes to performance. As future work, the program implementation could be studied and improved upon even more, as there is plenty of space for optimization on such a big program as this one.

ACRONYMS AND ABBREVIATIONS

NP	Non-deterministic polynomial time

ANNEX

```
1   GraphTraversal graphTraversal(graph);
2   while (graphTraversal.curNode != NONE) {
3       while (graphTraversal.curEdgeOffset != NONE) {
4           uint32_t neighbor =
                graph.edgeBuffer[graphTraversal.curEdgeOffset];
5           ...
6           graph.getNextEdge(graphTraversal);
7       }
8       graph.getNextNode(graphTraversal);
9   }
```

Figure 14: Traversing through all the active edges of the graph

```
1   void Reductions::reduce3() {
2       do {
3           do {
4               do {
5                   do {
6                       do {
7                           removeDominatedNodes();
8                       } while (foldCompleteKIndependentSets());
9                   } while (removeUnconfinedNodes());
10              } while (foldTwins());
11          } while (removeShortFunnels());
12      } while (removeDesks());
13      buildConnectedComponents();
14      removeEasyInstances();
15      removeLineGraphs();
16      graph.rebuild();
17  }
```

Figure 15: Reductions step for graph instances of max degree 3

```
1   # Undirected graph (each unordered pair of nodes is saved once):
       datasets/example.txt
2   # Undirected Erdos-Renyi random graph.
3   # Nodes: 7 Edges: 7
4   # NodeId NodeId
5   0  1
6   0  2
7   0  3
8   0  4
9   0  6
10  4  5
11  5  6
```

Figure 16: Input file for the graph on Figure 2

```
1     Snap-4.0/examples/graphgen/graphgen -g:e -n:100 -m:500 -o:graph.txt
```

Figure 17: Graph generation example using SNAP

REFERENCES

[1] Mingyu Xiao, Hiroshi Nagamochi, *Exact Algorithms for Maximum Independent Set*, Information and Computation 255(1) (2017) 126-146

[2] Mingyu Xiao, Hiroshi Nagamochi, *An Exact Algorithm for Maximum Independent Set in Degree-5 Graphs*, Discrete Applied Mathematics 199 (2016) 137-155

[3] Mingyu Xiao, Hiroshi Nagamochi, *A Refined Algorithm for Maximum Independent Set in Degree-4 Graphs*, J. of Combinatorial Optimization 34(3) (2017) 830–873

[4] Mingyu Xiao, Hiroshi Nagamochi, *Confining Sets and Avoiding Bottleneck Cases: A Simple Maximum Independent Set Algorithm in Degree-3 Graphs*, Theoretical Computer Science 469 (2013) 92-104

[5] Robert Endre Tarjan, Anthony E. Trojanowski, *Finding a Maximum Independent Set*, SIAM J. Comput. 6(3) (1976) 537–546

[6] J. M. Robson, *Algorithms for maximum independent sets*, J. of Algorithms 7(3) (1986) 425–440

[7] Lijun Chang, Wei Li, Wenjie Zhang, *Computing a Near-Maximum Independent Set in Linear Time by Reducing-Peeling*, Proceedings of the 2017 ACM International Conference on Management of Data 1181-1196

[8] Yu Liu, Jiaheng Lu, Hua Yang, Xiaokui Xiao, Zhewei Wei, *Towards Maximum Independent Sets on Massive Graphs*, Proceedings of the VLDB Endowment - Proceedings of the 41st International Conference on Very Large Data Bases 8(13) (2015) 2122-2133

[9] Robert Tarjan, *Depth-First Search and Linear Graph Algorithms*, SIAM J. Comput. 1(2) (1972) 146–160

[10] J.E. Hopcroft and R.E. Tarjan, *Finding the Triconnected Components of a Graph*, Technical Report, Cornell University (1972)

[11] Arkady Kanevsky, VijayaRamachandran, *Improved Algorithms for Graph Four-Connectivity*, 42(3) (1991) 288-306

[12] *SNAP, the network analysis platform*, accessed 24 October 2018, `<http://snap.stanford.edu/snap/>`

[13] *Maximum Independent Set program implementation on Github*, accessed 24 October 2018, `<https://github.com/iPapatsoris/Maximum-Independent-Set>`