An in-depth study of the syntactic analysis of graphs using search matching algorithms, to the degree of sub graphs

Student number: 10637291
email: lectonlm@gmail.com
Supervisor: Dr Linda Marshall
Dept Computer Science, University of Pretoria

November 1, 2015

1 Graph Basics

1.1 Graph Overview

A Graph in Mathematics and Computer Science is defined as a pair $G = (V, E)$, where V is the set of vertices and E is the set of edges, formed by pairs of vertices with each other. Figure 1 demonstrates the structural attributes of a simple graph.

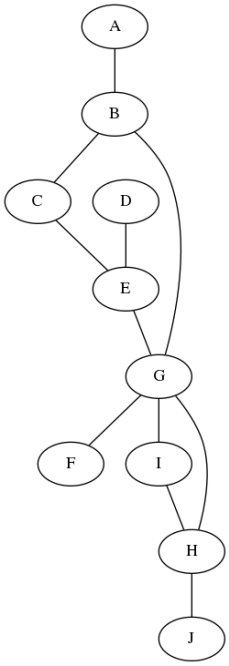


Figure 1: Representation of a graph

Graphs can either directed or they can undirected. This means that the edges in the graph could have an abscence of direction as in the 1 above, or they could have a direction showing from which vertice the edge is coming from, and to which vertice the edge is going to. Figure 2 demonstrates a directed graph, commonly known as a digraph. This characteristic is demonstrated by the edge 3, that goes from node B to node G, and also by edge 2 that goes from node B to node C.

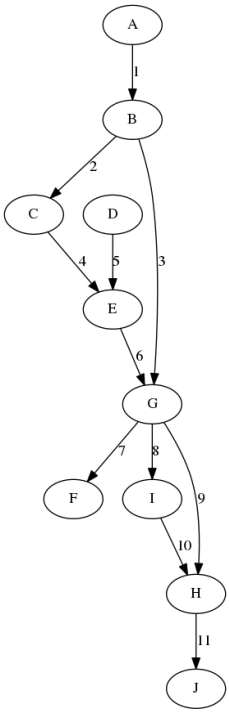


Figure 2: Representation of a digraph

Note that the undirected graph mentioned above is commonly referred to as a bigraph because the direction of its edges could be percieved as to going in both direction as it is not specifed.

In this paper, the focus is primarily on directed graphs, and they shall be referred to as digraphs from here onward.

1.2 Empty graph

An *Empty graph*, is a graph consisting of n vertices and zero edges, where n ¿ 0. An example is demonstrated in the figure 3

Figure 3: Representation of an empty graph with 10 vertices

In the example above, figure 3 has 10 vertices but none of the vertices have a relationship with each other thus.

1.3 Digraph

A Digraph in mathematics is defined as is a pair of disjoint vertices and edges (V, E), and their repective mappings that comprises of two components, namely the initial vertix and terminal vertice of each edge i.e. each edge has a initial vertix:

$$Ei \rightarrow Vi \tag{1}$$

and a terminal vertix:

$$Vj \rightarrow Ej \tag{2}$$

for some vertices Vi,Vj in V and edges Ei,Ej in E [7] refer to the figure 2.

1.4 Graph representation

Graphs are represented in a variaty of ways, from adjacency lists, incident matrices and adjacency matrices. The algorithms that are studied in this paper make us of adjacency matrices and adjacency list representations of graphs.

1.4.1 Adjacency matrices

An adjacency matrices is a nxn matrix A, with $A(i, j) = 1\ iff(i, j)E$ [9]. This means that wherever there is an edge in the graph, it is denoted by a 1 in the matrix, places in the matrix where there is an absence of an edge, are denoted by 0.

Figure 4 depicts the association between the graph in figure 1 and its adjacency matrix.

	A	B	C	D	E	G	F	I	H	J
A	0	1	0	0	0	0	0	0	0	0
B	1	0	1	0	0	1	0	0	0	0
C	0	1	0	0	1	0	0	0	0	0
D	0	0	0	0	1	0	0	0	0	0
E	0	0	1	1	0	1	0	0	0	0
G	0	1	0	0	1	0	1	1	1	0
F	0	0	0	0	0	1	0	0	0	0
I	0	0	0	0	0	1	0	0	1	0
H	0	0	0	0	0	1	0	1	0	1
J	0	0	0	0	0	0	0	0	1	0

Figure 4: Representation of a graph and its associated adjacency matrix

1.4.2 Adjacency list

An Adjacency list is vertices of a graph, of which each vertice is connected to the list. The vertices in an adjacency list point to their own list of edges that they are connected to(i.e. the list contains the edges that connect them to other vertices).

Figure 5 depicts the association between the graph in figure 1 and its adjacency list.

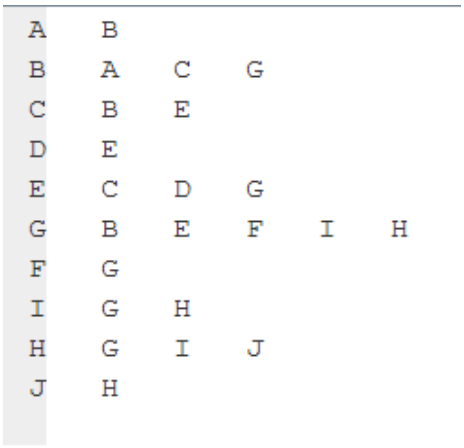


Figure 5: Representation of a graph and its associated adjacency list

1.4.3 Set of Pairs

Another representation of graphs is by the use of triplets $(source, destination, lable)$, as depicted in the thesis of Linda Marshall [17] . Each element of the triplet represents a specific attribute of the graph, the first element represent the source vertice, the second represents the destination vertice and the last element represents the corresponding label for the edge.

1.5 Supergraphs and subgraphs

Let G_A be be graph define as follows $G_A = (V_A, E_A)$ and let G_B be another graph that is defined as follows $G_B = (V_B, E_B)$ where V_A,V_B are sets of vertices and E_A,E_B are sets of edges. In graph theory, a graph G_A is said to be a subgraph of graph G_B, and graph G_B is said to be a supergraph of graph G_A if all the vertices and edges that are in graph G_A are also in graph G_B, that is [3]:

(1) $V_A \subseteq V_B$, and

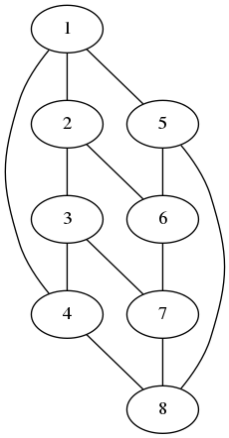
(2) Every edge of G_A is also an edge in G_B.

In figure 1, the graph constructed by vertices D, E, G, F and I is the sub-graph of the entire graph. And thus the graph is a super-graph of the sub-graph constructed by the vertices, namely D, E, G, F and I.

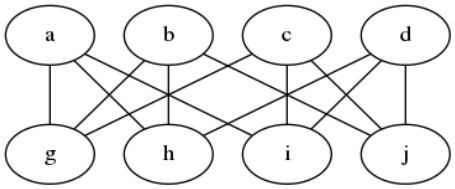
1.6 Graph Isomorphism

Two graphs are said to be isomorphic if they are syntactically similar to each other iff there is a bijection between their respective nodes which make each edge of G_A correspond to exactly one edge

of G_B, and vice versa [12], i.e. the graphs are structurally the same to each other. This property is demonstrated in figures 6b.



(a) Isomorphic graph on the left



(b) Isomorphic graph on the right

Figure 6: A demostration of two isomorphic graphs

The two graphs look very different, but when they are further inspected, it is evident that the two are a representation of the same structural scheme or maybe even the same graph that has been rearranged.

Consider the vertice 1 from the left-most graph, it has three edges going to and from vertices $2, 4$ and 5, and now consider the vertix a from the right-most graph, it also has three vertices going to and from it, namely g, h and i. These two vertices have the same structural, but from two different graphs, the same goes for all the other vertices in both graphs .i.e each vertice in the left-most graph can be associtated with one in the right-most graph as we did for vertices 1 and a.

2 Search Matching Algorithm

2.1 Introduction

Subgraph ismorphis can be determined using a brute-force approach on the tree representation of a graph G_A. Though this technique is effective, it is however not efficient, this is because all the possible permutation subgraphs of a graph G_A are tested against the graph G_B to determine if there are subgraphs in graph G_A that are isomorphic to graph G_B.The number of subgraphs of a graph G_A increase at an exponential rate with every addition of a vertice V_n into the graph, thus the total number of subgraphs that can be evaluated are

$$ST = 2^{n/2} \tag{3}$$

where ST is the total number of subgraphs and n the number if vertices in the graph G_A.

The matching process is computationally expensive due to this very fact, that is the more vertices there are in the graph G_A, the more expensive it becomes to detect the subgraph ismorphisms because of the amount of subgraphs it has and thus must be evaluated.

This paper explores two graphs that are very effective with regards to graph isomorphis and subgraph isomorphism detection. The algorithms that are invesicated are the Ullman Algorithm and the VF2 algorithm.

2.2 Ullman Algorithm

2.2.1 Algorithm

The Ullman algorithm was developed by J.R.Ullman and was published in his paper titled "An Algorithm for Subgraph Isomorphism" [1].The algorithm performs graph matching on an adjacency matrix representation of both the graphs, and uses the depth search first(DSF) recursicve approach to traverse through the graphs and perform the graph matching process. The Ullman algorithm improves the effiency of the brute-force approach at detecting subgraph ismorphisms by deductively eleminating nodes in the tree that are in graph G_A, but are not in graph G_B, thus reducing the number of subgraphs that are matched against graph G_B to determine ismorphism.

The algorithm starts by building a starting adjacency matrix $M0$ using the two adjacency matrix representations of graphs G_A and G_B using the following procedure.

(1) Construct a $n * m$ matrix where n is the number of rows of graph G_B and m is the number of colums of graph G_A.

(2) Set all the entries in the matrix to the value of 1.

(3) Apply the following rule: Set the values in $M0$ to 0 for all $M0_{ij}$ where the degree of a vertice in graph G_A at j is greater then the degree of the same vertice in graph G_B .i.e.

$$deg(Ai) < deg(Bj). \tag{4}$$

A more formal representation of this rule is as follows

$$f(x) = \begin{cases} 1, & \text{if } deg(Ai) \geq deg(Bj) \\ 0, & \text{otherwise } \forall \text{i,j} \end{cases}$$

When the starting matrix has been constructed, we systematically permute matrices M^d from the starting matrix $M0$ where d represents the depth of the generated matrix. The procedure of generating the permuted matrices follows a depth search first (DSF) recursicve approach where the stopping condition (leaf matrices) conform to the following form:

(1) M contains only $0's$ and $1's$.

(2) There is exactly one 1 in each row.

(3) Not more than one 1 in each colum.

An demonstration of how the permutation matrices are generated is demonstrated in figure 7.

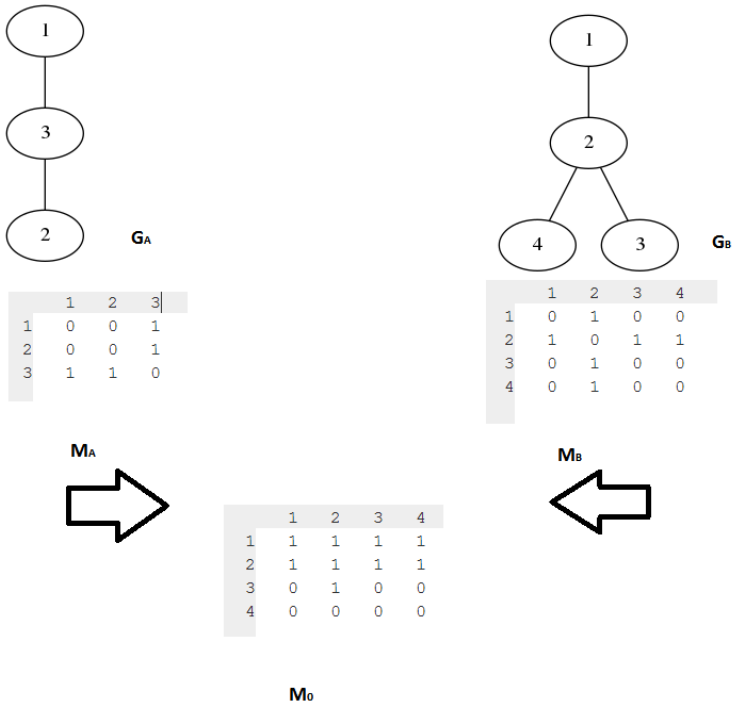


Figure 7: Demonstation of how a permutation matrix is generated from two graphs

Once all the permutation matrices have been generated, each one of the matrices is matched with a graph C, that is obtained from the dot product of the permuted matrix and the graph G_A. The formula for calculating graph C is follows: $C{=}M_n(M_n \ .\ G_A)T$, where G_A = input graph and M_n = permutated matrix M_n in M^d, obtained from the starting matrix $M0$ $(M_n \ .\ G_A)T$ = the transpose of the dot product of the permutated matrix M_n and the graph G_A. If there is a single instance of the matrix C, that is calculated using some permutated matrix M_n obtained from the starting matrix $M0$, that is equal to matrix G_B, then G_B is isomorphic to G_A. Thus G_B is isomorphic to G_A $iff G_B$

$$ij = 1 \rightarrow Cij = 1 \forall i, j \tag{5}$$

If non of the generated permutated matrices can statisfy this condition, then G_B is not isomorphic to G_B.

2.2.2 Ullman Pseudo code

Step 1 $M = M^0$, $d = 1$, $H_1 = 0$,
for all $i = 1, \cdots, p_\alpha$ set $F_i := 0$;
refine M, if exit FAIL then terminate algorithm;
Step 2. If there is no value of j such that $m_{dj} = 1$ and $f_j = 0$ then go to step 7,
$M_d := M$;
if $d = 1$ then $k = H_1$ else $k := 0$;
Step 3 $k = k + 1$,
if $m_{dk} = 0$ or $f_k = 1$ then go to step 3;
for all $j \neq k$ set $m_{dj} = 0$;
refine M; if exit FAIL then go to step 5;
Step 4 If $d < p_\alpha$ then go to step 6 else give output to indicate that an isomorphism has been found;
Step 5 If there is no $j > k$ such that $m_{dj} = 1$ and $f_j = 0$ then go to step 7,
$M = M_d$;
go to step 3,
Step 6 $H_d = k$, $F_k = 1$; $d = d + 1$;
go to step 2,
Step 7. If $d = 1$ then terminate algorithm,
$F_k = 0$; $d = d - 1$; $M := M_d$, $k := H_d$;
go to step 5;

Figure 8: Pseudo code of the Ullman algorith [1]

2.3 VF2 Algorithm

2.3.1 Algorithm

The VF2 algorithm was introduced by L.P.Cordella, P.Foggiaa, C.Sansone and M.Vento [11]. The algorithm is suitable for graph matching and isomorphic determination, including subgraph isomorphic determination on large graphs, this is attributed to the Data structures that the algorithm uses and the manner in which they are used [11], this feature is discussed later in the paper.

The algorithm performs the matching process by attempting to find a mapping M, of vertices in graph G_A which correspond to vertices in graph G_B. The mapping is then used to determine if the two graphs are completely syntactically similar(isomorphic), partially syntactically similar or have no structural similarities at all.

2.3.1.1 Matching and Mapping definition

A mapping M is defined as a set of pairs (n, m), where n is a vertice from G_A and m a vertice from G_B, thus $n \subseteq G_A$ and $m \subseteq G_B$.

The isomorphism determining properties of the mapping are defined as follows, a mapping $M \subset N_A * N_B$ is isomorphic $iff M$ is a bijection, that preserves the branching structure of G_A and G_B, where N_A is a set of vertices from G_A and N_B a set of vertices from G_B.

The mapping $M \subset N_A * N_B$ is subgraph isomorphic $iff M$ is isomorphic to G_A and a subgraph of G_B.

2.3.1.2 Mapping Procedure

The mapping M comprises of state based partial solution morphisms $M(s)$ for each state s.The process of finding the mapping M that is described above uses State Space Representations (SSR) [12]. The partial solution morphisms $M(s)$ selects two subgraphs from G_A and G_B, namely $G_A(s)$ and $G_B(s)$ respectively. The subgraphs comprises of only vertices that are present in the partial solution $M(s)$ for the state s as well as the edges joining them together.

The algorithm starts with an initial state $s0$ that has no mapping between the two graphs, thus $M(s0) = NULL$. The algorithm then computes a set of candidate pairs $P(s)$. Each candidate p in the set is checked against the feasibility function that is discussed in the following chapter, if p is successful then it is added to the state s. And the successor s' is computed using a combination of the predecessor state and the candidate p, thus:

$$s' = s \cup p \tag{6}$$

The process of generating successor states is a recursive procedure that makes use of the depth first traversal for graphs. When a path has been exhausted and a solution has not yet been found, the algorithm uses backtracking to explore the alternative paths [11,13].

2.3.1.3 Definition of the set $P(s)$ and of the feasibility function $F(s, n, m)$

The VF2 algorithm generates the states with close consideration that only some of the states are consistant with the desired morphisms [12]. The algorithm avoids inconsistant states by making use of a set of rules in it's state generation procedure, thus ensuring that only consistant state are generated, these rules are refered to as feasibility rules.

The algorithm uses a function called a feasibility function to test that an additon of a pair (n, m) to a state will be consistant. If the addtion of the pair passes all the feasibility rules, the algorithm will return a true value, if not, a false value indicating that the procedure results in an inconsistant successor state s', and thus that state s' will not be explored by the algorithm.

A further filter can be applied in the consistent states to rule out those states whose successor states will be inconsistant, this apporoach is employed by adding a additional rules called $k-look-ahead$ rules [12]. They check to see if the current state s will have a consistant successor state after k steps, i.e. they check to see if the states from s to s^k are consistant with the desired morphisms.

2.3.1.4 Condidate Pairs

The candidate pairs are obtained by considering the vertices that are connect to $G_A(s)$ and $G_B(s)$, the sub-graphs of G_A and G_B in the state s. The vertexs are used to form the pairs (n, m) as defined above. In order explain how the pairs are formed, we must first introduce the following definitions: Let:

(1) $T_Ain(s)$ be the set of vertexs that are not yet in the partial mapping $M(s)$ and are the origin of the edges from graph G_A

(2) $T_Bin(s)$ be the set of vertexs that are not yet in the partial mapping $M(s)$ and are the origin of the edges from graph G_B

(3) $T_Aout(s)$ be the set of vertexs that are not yet in the partial mapping $M(s)$ and are the destination of the edges from graph G_A

(4) $T_Bout(s)$ be the set of vertexs that are not yet in the partial mapping $M(s)$ and are the destination of the edges from graph G_B

The pair (n, m) is made by vertex n from $T_Aout(s)$ and m from $T_Bout(s)$. If the any of the sets is empty, then we consider the vertex n from $T_Ain(s)$ and m from $T_Bin(s)$. In the case where that graphs are not connected, the pairs will be made by all the vertex not yet contained in either $G_A(s)$ and $G_B(s)$. These pairs form the entries in the set $P(s)$ for that respective state s.

2.3.1.5 The feasibility rules

The feasibility rules that are used to ensure that the states that are evaluated play a role in improving the performance, by preventing inconsistent states from being explored and thus optimizing the execution of the algorithm. There are five general feasibility rules defined as R_{pred},R_{succ},R_{in},R_{out} and R_{new} respectively.

The feasibility functions check for two main things, firstly they check the consistency of the partial solution in the successor state s', namely $M(s')$. Rules R_{pred} and R_{succ} are the rules used for this checking.
The remaining rules are used for pruning the search space for different levels of look ahead. The R_{in} and R_{out} are used to look ahead one level and determine which of those successor states are consistent, and R_{new} is used for the same purpose, but for a look ahead level of two. The definition for each rule is as follows:

$$Rpred(s,n,m) \Longleftrightarrow (\forall n' \in M1(s) \cap Pred(GB,n) \exists m' \in Pred(GA,m) | (n',m') \exists M(s) \wedge \forall m' \in M2(s) \cap Pred(GA,m) \exists n' \in Pred(GB,n) | (n',m') \exists M(s)), \quad (7)$$

$$Rsucc(s,n,m) \Longleftrightarrow (\forall n' \in M1(s) \cap Succ(GB,n) \exists m' \in Succ(GA,m) | (n',m') \exists M(s) \wedge \forall m' \in M2(s) \cap Succ(GA,m) \exists n' \in Succ(GB,n) | (n',m') \exists M(s)), \quad (8)$$

$$Rin(s,n,m) \Longleftrightarrow (Card(Succ(GB,n) \cap TBin(s)) \geq Card(Succ(GA,m) \cap TAin(s))) \cap (Card(Pred(GB,n) \cap TBin(s)) \geq Card(Pred(GA,m) \cap TAin(s))) \quad (9)$$

$$Rout(s,n,m) \Longleftrightarrow (Card(Succ(GB,n) \cap TBout(s)) \geq Card(Succ(GA,m) \cap TAout(s))) \cap (Card(Pred(GB,n) \cap TBout(s)) \geq Card(Pred(GA,m) \cap TAout(s))) \quad (10)$$

$$Rnew(s,n,m) \Longleftrightarrow Card(\tilde{NA}(s) \cap Pred(GB,n)) \geq Card(\tilde{NB}(s) \cap Pred(GA,n)) \wedge Card(\tilde{NA}(s) \cap Succ(GB,n)) \geq Card(\tilde{NB}(s) \cap Succ(GA,n)) \quad (11)$$

In order for a state to be considered consistent, it must pass a combination of all of the five rules, namely:

$$Fsyn(s, n, m) = Rpred \wedge Rsucc \wedge Rin \wedge Rout \wedge Rnew \tag{12}$$

where $F_{\text{syn}}(s, n, m)$ the feasibility function that is envoked upon the state s.

2.3.2 VF2 Pseudo code

```
PROCEDURE Match(s)
   INPUT:  an intermediate state s; the initial state s0 has M(s0)=∅
   OUTPUT: the mappings between the two graphs

   IF M(s) covers all the nodes of G2 THEN
     OUTPUT M(s)
   ELSE
     Compute the set P(s) of the pairs candidate for inclusion in M(s)
     FOREACH p in P(s)
       IF the feasibility rules succeed for the inclusion of p in M(s) THEN
         Compute the state s´ obtained by adding p to M(s)
         CALL Match(s')
       END IF
     END FOREACH
     Restore data structures
   END IF
END PROCEDURE Match
```

Figure 9: Pseudo code of the VF2 algorith [11]

2.4 Conclusion

The Ullman and VF2 algorithm that are discussed above both follow a similar approach in attempting to perform graph mathing. They both construct a tree from the adjacency representation of their input graphs and use the depth first tree traversal techniques to evaluate the graphs, this is done every effectively by both the algorithms.

Though both algorithms are effective in their own respective regards, they are optimized rather differently and thus differe in their degree of complexity. The VF2 algorithm optimizes its execution by performing a look-ahead operation of two states from its current states in an attempt to ignore paths that will result in inconsistant states.

The Ullman algorithm on the other hand optimizes its execution by not computing all possible sub-graphs of some graph G, but reduces the computated matrices by initially computing a matrice $M0$ from the input graphs and then ensuring that all the matrices that are computed from $M0$ are tested so as to prevent the algorithm from exploring a branch that will not result in a graph or sub-graph isomorphism

3 VFLibGraph Library

3.1 Introduction

The VFLibGraph library is a graph matching library that is written in C++, it was developed at the Intelligent Systems and Artificial Vision Lab (SIVALab) which is situated in the University of Naples [13]. The library was originally developed to test the VF algorithm [13], the predecessor of the VF2 algorithm, but the library has since evolved to include some of the latest graph matching algorithms such as the Schmidt and Druffel algorithm.

3.2 Using the library

The library provides interfaces to the implemented algorithms that it has in its employ, and it also has a comprehensive documentation of how each algorithms interface can be constructed, as well as the matching process between two input graphs.

We have constructed interfaces for the Ullman algorithm and the VF2 algorithm to test against our data set, which comprises on graphs of various number of vertices and edges.

4 Experimentation

4.1 Data Set

The graph data that is used in the experimentation was created by David S. Johnson, Cecilia R. Aragon,Lyle A. McGeoch and Catherine Schevon [14]. It was created in 1991 to study optimization by simulated annealing. One of the graphs that used was chosen to be experimented on to study the subject algorithms, namely the Ullman and VF2 algorithms. The graph used can be found here [14].

The graph is a large direct graph that comprises of 250 vertices and 3218 edges [14], and it is used as the super-graph in the experiments, and will be refered to as the subject-graph from here on out. Figure 10 depicts a diagram of how the graph looks like.

The algorithms both need two graph in order to determine if they are isomorphically related to each other or not, thus subgraphs from our object graph are required for this comparison. These subgraphs are then generated from the object graph and they all meet the condition that:

- They are either partially or completely isomorphic in relation to one or other digraph inside the set.

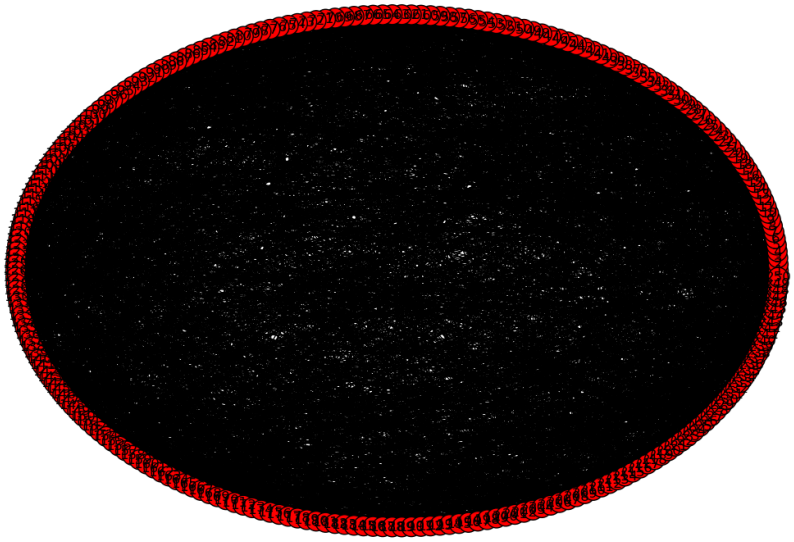
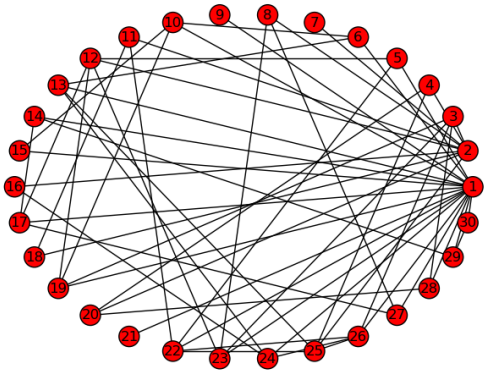


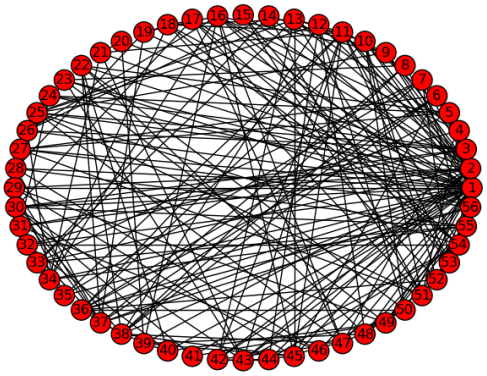
Figure 10: The super-graph used in the experiment comprising of 250 vertices

The subgraphs will be refered to as subject-graphs from here on out.

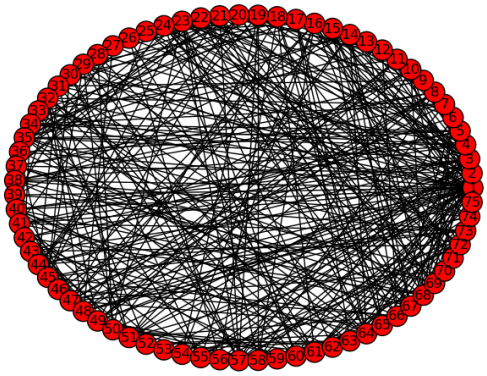
The generated subgraphs have the following number of vertices: 30,56,75,92,109,121,148,166,181,197,211 and 222. Each one of the graphs is sub-graph isomorphic to the object graph, and will compared with it to determine how long it takes to find the isomorphic relationship, and how much virtual memory [15] is used to find the relationship.
Figure 11 depics how some of the generated subgraphs lot relative to figure Figure 10.



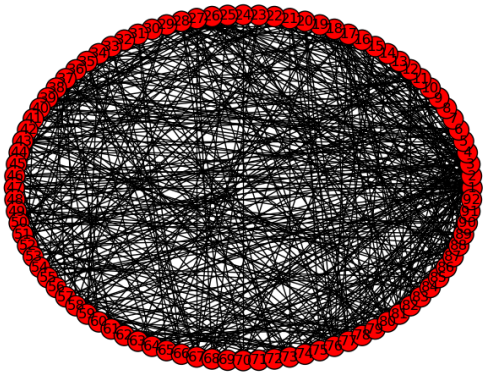
(a) Generated subgraph with 30 vertices



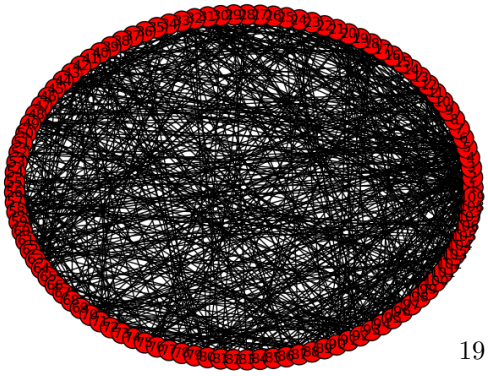
(b) Generated subgraph with 56 vertices



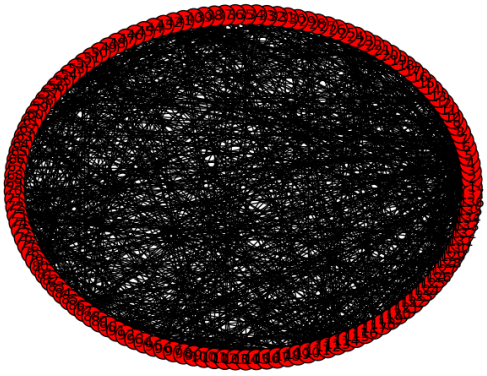
(c) Generated subgraph with 75 vertices



(d) Generated subgraph with 92 vertices



(e) Generated subgraph with 109 vertices



(f) Generated subgraph with 121 vertices

Figure 11: A demostration of two isomorphic graphs

4.2 Experiment

The experiment that is conducted on the Ullman [1] and the VF2 [11] algorithm uses a quantitative approach that evaluates the algorithms against each other. The search graph matching algorithms are implemented and experimented on, using two digraph or disjoint objects at a time. The experiment for that is employed, measured and records the algorithms ability and performance of the algorithms in cases where Graph G_A and Graph G_B are either partially or completely isomorphic.

Several experiments are done on the two algorithms so that we can understand their behaviour in different conditions, and these experiments that are conducted are done on both joint and disjoint graphs. The experiments that are performed are expained below.

4.2.1 The comparison criteria

The evaluation performed on the algorithm results is based on two criteria's. These are use to weigh the algorithms against against each other to determine which amongst them is better. The criteria's are listed below.

Space efficiency: This factor measures the amount of virtual memory (RAM) that is used by the algorithm in the graph matching process of its execution.The unit used to measure the amount of memory by the respective algorithms is measured in *bytes (b)*.

Time efficiency: This factor measures the time taken by the algorithm to start and *successfully* complete the graph matching process of its execution. The unit used to measure the amount of time taken by the respective algorithms is measured in *millisecond (ms)*.

4.2.2 Description Conducted Experiments

This section lists each of the experiments that where conducted for both algorithms as well as provides an explains of each experiment.

4.2.2.1 Performance evaluation for different vertice numbers

This experiment uses data that is mentioned in chapter 4.1. The experiment is conducted on joint and disjoint graphs. This experiment is intended on evaluating the behaviour of the two algorithms performance for graphs of different vertice numbers. The number of the vertices in the graphs range from 55 to 250 vertices in the graphs.

4.2.2.2 Performance evaluation for different edge numbers

This experiment uses generated graph data comprising of 1000 vertices and 499500 edges. The generated graph data is defined using the following rules.

(1) Define a set of vertices V for some graph G.

(2) Each vertice v in V is connect to every other vertice in V

(3) Non of the edges in the graph G are reflexive.

This experiment is intended on evaluating the behaviour of the algorithms for when they are searching for subgraphs of various sizes proprotional to some supergraph G_{SUP}.
The subgraphs are generated from the supergraph G_{SUP} for different percentages e.g. a generated subgraph that is 10% of the supergraph G_{SUP} graph.

4.2.3 Experiment implementation

The experimentation process is accomplished in phases and each phase is explained below. We will refer to Graphs G_{A} and G_{B} in this context.

4.2.3.1 Generate two graphs, G_{A} and G_{B}

In the first phase of the experimentation process, a supergraph G_{A} and a subgraph G_{B} are generated. The two graphs are built together with their associate vertices and edges.

4.2.3.2 Syntactic comparison

The comparison that is done, is dependent on the relationship between Graph G_{A} and G_{B}.

(1) G_{A} is compared with the whole of G_{B} for syntactical similarity.

   (I) A new subgraph of Graph B is generated.

   (II) The subgraph is compared with Graph G_{A} for syntactical similarity.

      (A) The time taken to perform the comparison is recorded.

      (B) The amount of memory used by the algorithm to perform the comparison is recorded.

(2) If Graph G_{A} and G_{B} are not completely syntactically similar, Graph G_{A} is compared with all the possible subgraphs of Graph B.

4.2.4 Result presevation

The amount of time and memory required by both alogrithm to complete the comparison procedure of each respective experiment is stored and graphically represented. The results from the algorithms are weighed against a comparison criteria, and it is this criteria that is used to evaluate the algorithms against each other.
Once the results have been obtained and throughly evaluated, then the algorithms with the best perform per criteria field are recorded.
And from that set, the best algorithm is chosen overall relative to the others based on the criteria.

5 Experiment Results

This section presents the results of the experiments described in Chapter 4.2.2.An evaluation of the results is performed before the algorithms can be compared against each other based on the criteria specified in chapter 4.2.1, namely the time and space efficiency of the algorithms.

5.1 Joined graph results for the evaluation of different vertice numbers experiment

This section presents the time and memory results for experiment described in section 4.2.2.1. The experiment is conducted on joined graphs.

5.1.1 Memory Results

This section compares the efficiency of the two algorithms in terms of their memory. The amount of virtual memory used by the two algorithms are graphically represented, analysed and a conclusion of as to which of the two algorithm is more efficient in terms of memory is reached based on figure 12. The red line depics the amount of memory used by the Ullman algorithm and the blue line depicts the amount of memory used by the VF2 algorithm.

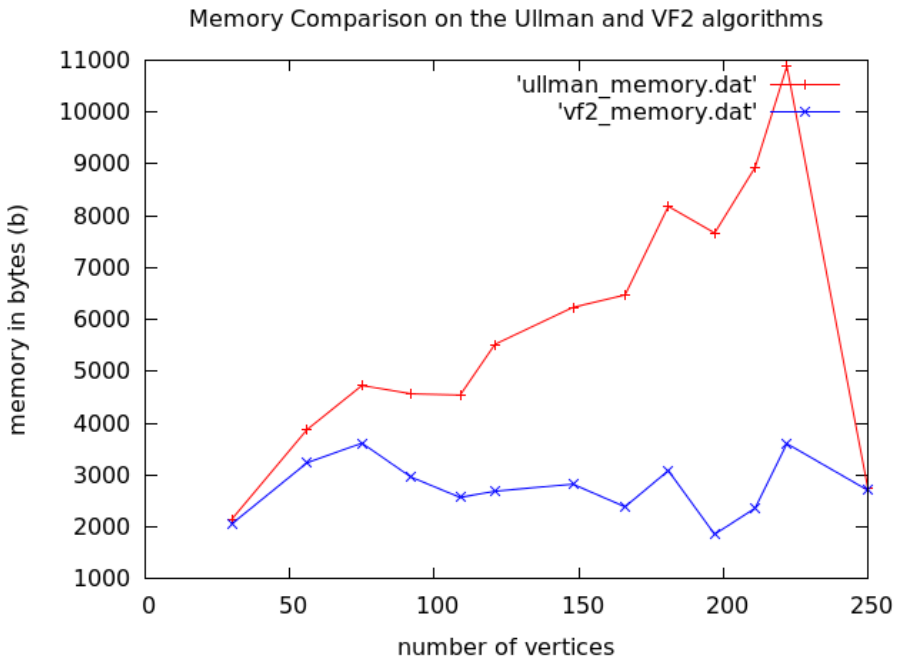


Figure 12: Graph depiting the results for the memory comparison between the algorithms for joined graphs

5.1.1.1 Analysis of results

Figure 12 depicts the comparison of the Ullman and the VF2 algorithms on the criteria of memory used during the execution of the graph matching processes of the respective algorithms.

The virtual memory used by the the Ullman algorithm is depicted by the *red* line in figure 12 and the *blue* line represents the virtual memory used by the VF2 algorithm.

Figure 12 depicts that the memory usage of the two algorithms is approximatly the same for smaller graphs of *30 - 40* vertices. But from approximatly *40* and above, it is clear that the performance of the VF2 algorithm is exceptionally superior to that of the Ullman algorithm. This deduction is derived from the observation on figure 12, we can see the more vertices there are in the graph, the more virtual memory resources is required by both graphs, but the Ullman algorithm requires a greater amount of resources than those used by the VF2 algorithm.

The superiority of the VF2 algorithm for this data set is limited, this is because when the number of vertices in the graph is approximatly *220*, the required memory resources for both algorithms start dropping rapidly, and they finally end up requiring the same amount of memory resources.

5.1.1.2 Conclusion

Figure 12 has provided with a graphical representation of the virtual memory required by both algorithms, and thus a way to make deductions about the efficiency of the two algorithms in terms of memory.

From 12, it is clear that the VF2 is algorithm is more memory efficient than the Ullman algorithm as it requires the least amount of virtual memory overall for its execution.

5.1.2 Time Results

This section compares the efficiency of the two algorithms in terms of their time. The amount of taken by the two algorithms are graphically represented, analysed and a conclusion of as to which of the two algorithm is more efficient in terms of taken is reached based on figure 13. The red line depics the amount of memory used by the Ullman algorithm and the blue line depicts the amount of memory used by the VF2 algorithm.

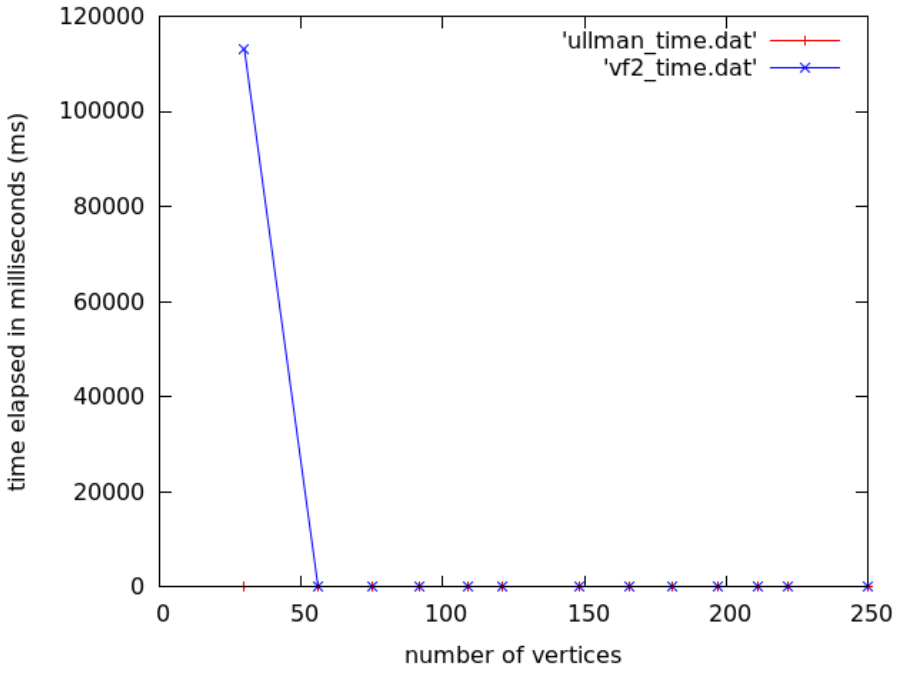


Figure 13: Graph depiting the results for the time comparison between the algorithms for a set of joined graphs

5.1.2.1 Analysis of results

Figure 13 depicts the comparison of the Ullman and the VF2 algorithms on the criteria of the time taken during the execution of the graph matching processes of the respective algorithms.

The amount of time taken by the Ullman algorithm is depicted by the *blue* line in figure 13, and the time taken by VF2 algorithm is depicted by VF2 algorithm is depicted by the *red* line in figure 13.

Figure 13 depicts that the amount of time taken by the Ullman algorithm to successfully complete its graph matching processes is very small and consistant for the data set used in the experimentation.

The time taken by the VF2 algorithm however is very high for small sets of vertices, approximately *40 - 60* vertices. But as the number of vertices increase, the amount of time declines rapidly and it equal to that of the Ullman algorithm.

5.1.2.2 Conclusion

Based of figure 13, it can be deduced that the time efficiency of the Ullman and VF2 algorithms are approximately equal to each other overall, with the exception that for small graphs, the Ullman

algorithm is more efficient than the VF2 algorithm in terms of the time taken to complete the graph matching procedures.

5.2 Disjoined graph results for the evaluation of different vertice numbers experiment

This section presents the time and memory results for experiment described in section 4.2.2.1. The experiment is conducted on disjoined graphs.

5.2.1 Memory Results

This section depicts the memory results for the two algorithms on a set disjoined graphs of various vertice numbers. The red line depics the amount of memory used by the Ullman algorithm and the blue line depicts the amount of memory used by the VF2 algorithm.

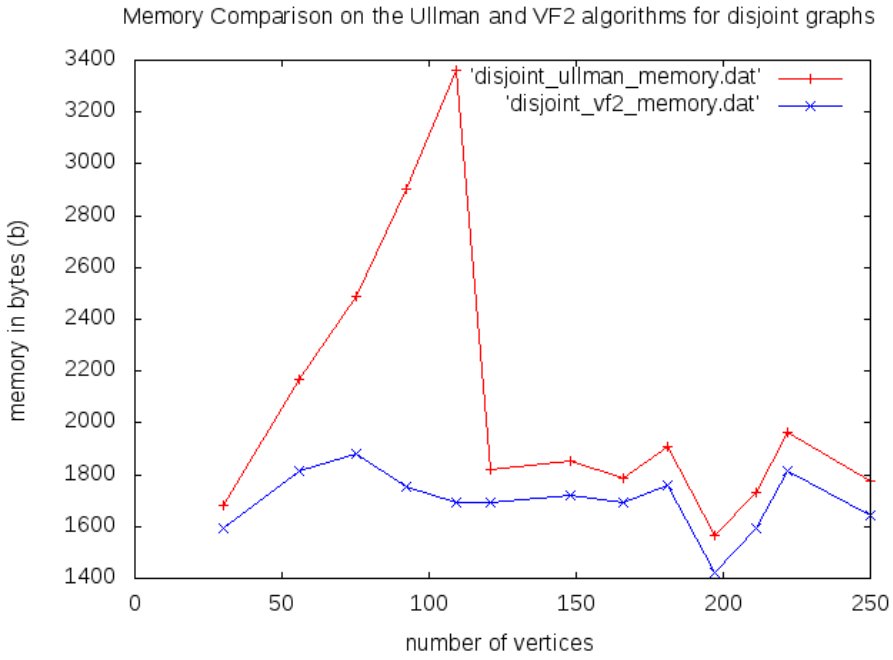


Figure 14: Graph depiting the results for the memory comparison between the algorithms for a set of disjoined graphs

5.2.1.1 Analysis of results

Figure 14 depicts the memory used by the two graphs by the two algorithms for a set of disjoint graphs with different vertice numbers.

The behavour of the two graphs appear to be very different for small graphs .i.e that is graphs

with a size less than 120 approximately. The Ullman algorithm's memory usage has a steep positive gradient from the beginning, and the gradient does not flactuate alot and stays consistant until it reaches its peeck. The VF2 algorithm however does not have very steep gradient, and it reaches its peek faster than the Ullman algorithm.

The behavour of the algorithms is approximately similar, they both increase and crease and decrease their gradient in a similar pattern, even though the Ullman still uses a greater amount of memory than the VF2 algorithm.

5.2.1.2 Conclusion

Based on figure 14, it is clear that the VF2 more memory efficient than the Ullman algorithm. The its difference in memory usage is more vast for small disjoint graphs, and the Ullman algorithms demand for memory is very high and consistant. The algorithms have a similar performance for larger graphs, but the Ullman algorithm still has a larger demand of memory than the VF2 algorithm.

5.2.2 Time Results

This section reports the time taken by the algorithms to complete the graph matching procedure for a set of disjoint graphs with different number of vertices.The red line depics the amount of memory used by the Ullman algorithm and the blue line depicts the amount of memory used by the VF2 algorithm.

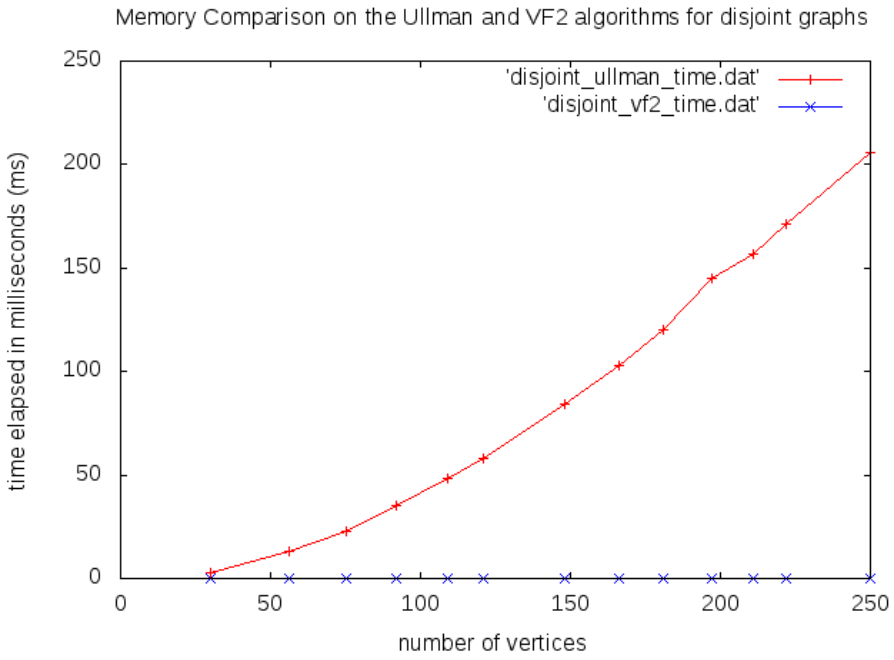


Figure 15: Graph depiting the results for the time comparison between the algorithms for a set of disjoined graphs

5.2.2.1 Analysis of results

Figure 15 depitcs the behavour of the two algorithms with regards to the amount of time taken by each algorithm to complete its respective graph matching procedures.

The Ullman algorithm has a gradually increasing slope that appears to be arbitrarily consistant throughout all the graph sizes. The VF2 algorithm has a very constant and consistent behavour with regards to the time that it requires. The amount of time required seems to be neglibable for this algorithm as it seems to be very close to zero for all graph sizes.

5.3 Boundary Cases

5.3.1 Introduction

This chapter evaluates the best, average and worst cases for the Ullman and the VF2 algorithms in terms of time and memory efficiency. The best, average and worst cases are by the graph traversal stratedy that is followed by both algorithms as well as the number of edges that the graphs have.

The best, average and worst cases for graph matching are define below as well as a graphical representation of 10 vertices depicting how the specific case looks like.

Best case: The best case for graph matching G A onto graph G B is defined as a graph G A having no edges defined in their graphs, i.e $E_{\text{A}} = 0$. Figure 16 depicts a graph of 10 vertices depicting the best case scenario.

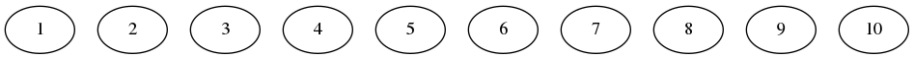


Figure 16: Graph of 10 vertices representing the best case scenario

Average case: The average case for graph matching G_{A} onto G_{B} is defined as a graph G_{A} with half of its vertices connected to each other. Thus the graph will have $(n(n1)/2)/2$ edges that are uniquily connected to each vertice in the graph. Figure 17 depicts a graph of 10 vertices depicting the average case scenario.

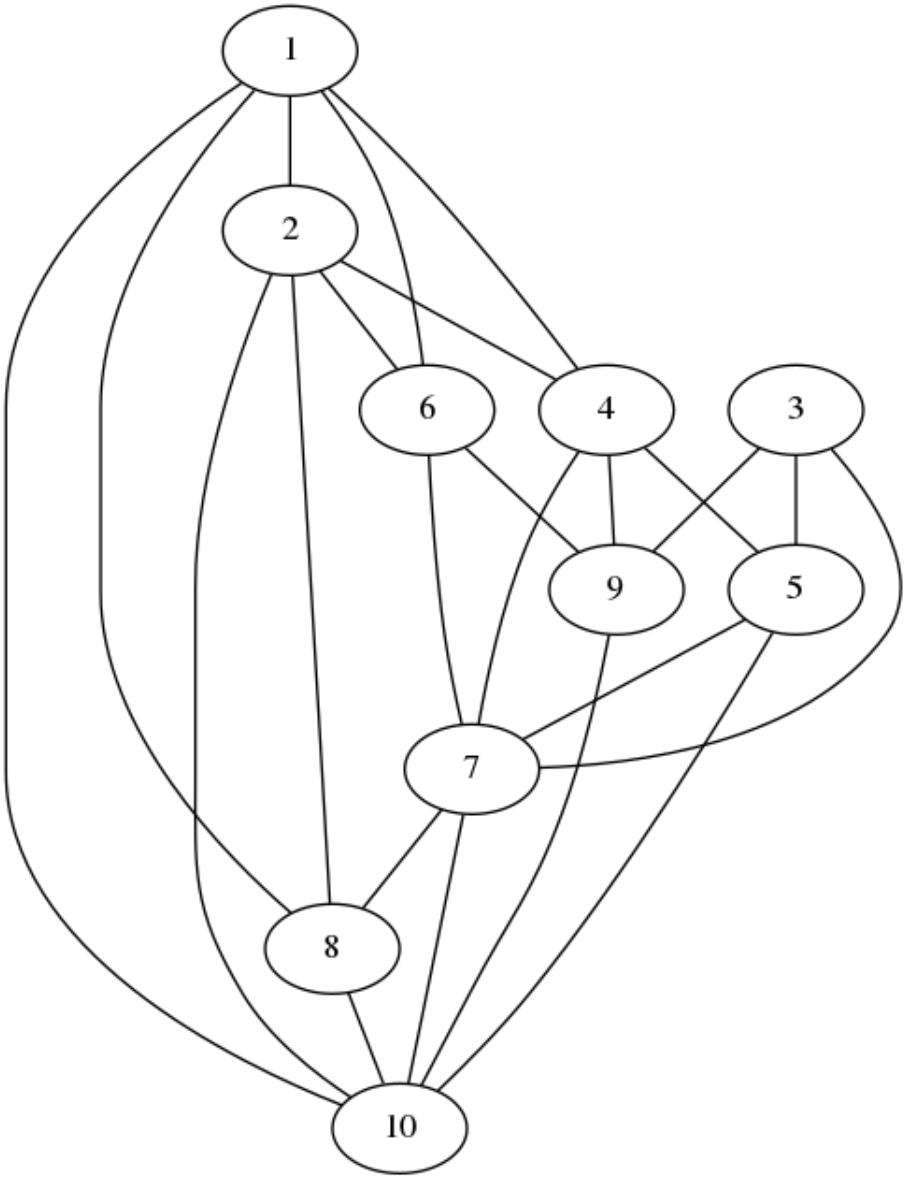


Figure 17: Graph of 10 vertices representing the average case scenario

Worst case: The worst case for matching graph G_A onto G_B is defined as a graph G_A with all its vertices connected to eact other. Thus the graph will have $n(n1)/2$ edges that are uniquily connected to each vertice in the graph. Figure 18 depicts a graph of 10 vertices depicting the worst case scenario.

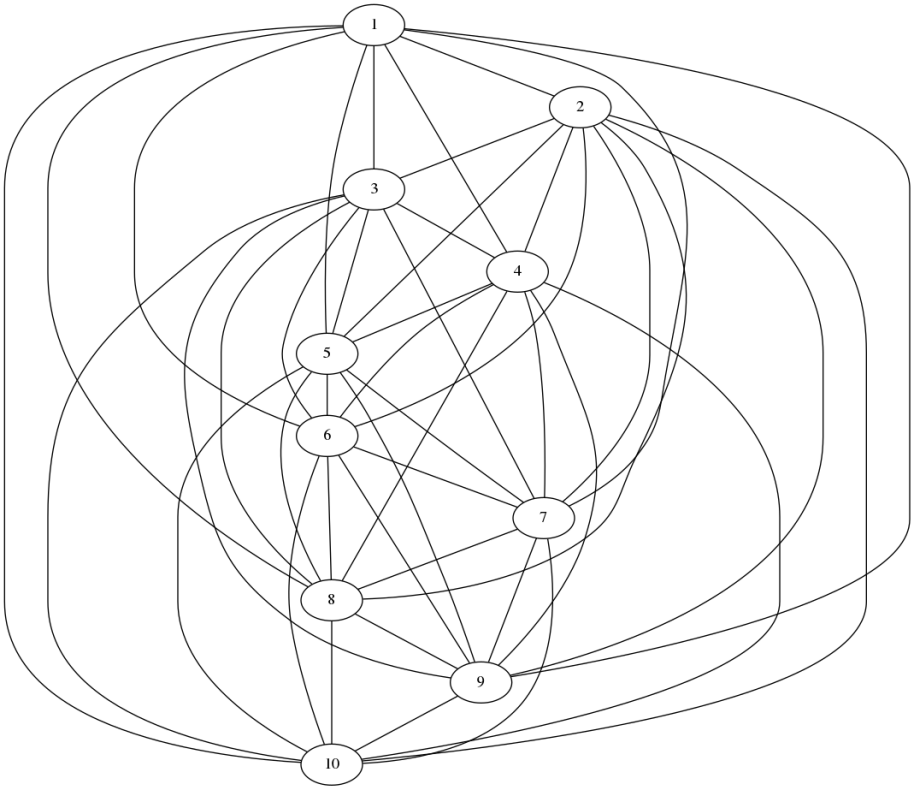


Figure 18: Graph of 10 vertices representing the worst case scenario

5.3.2 Memory Boundary Results

This chapter depicts the amount of memory used by the algorithms over time, especially on the boundary conditions of the experiment.

5.3.3 Ullman Algorithm

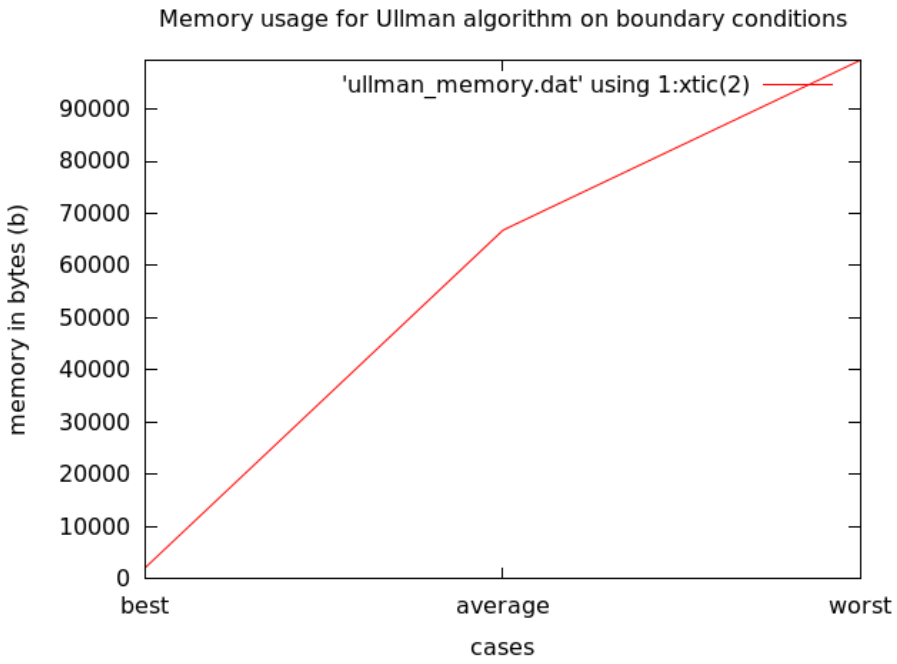


Figure 19: Representation of the memory usage by the Ullman algorithm on boundaries

5.3.3.1 Analysis of results Figure 19 depicts the amount of memory used by the Ullman algorithm over time. The figure indicated that the amount of memory used by the algorithm increases at a steady from the defined best case to the worst case for some graph G.

Based on the results from figure 19, it is clear that the algorithm consumes more of the systems RAM on the worst of the extreme case and the most little of RAM in the best of such cases.

5.3.4 VF2 Algorithm

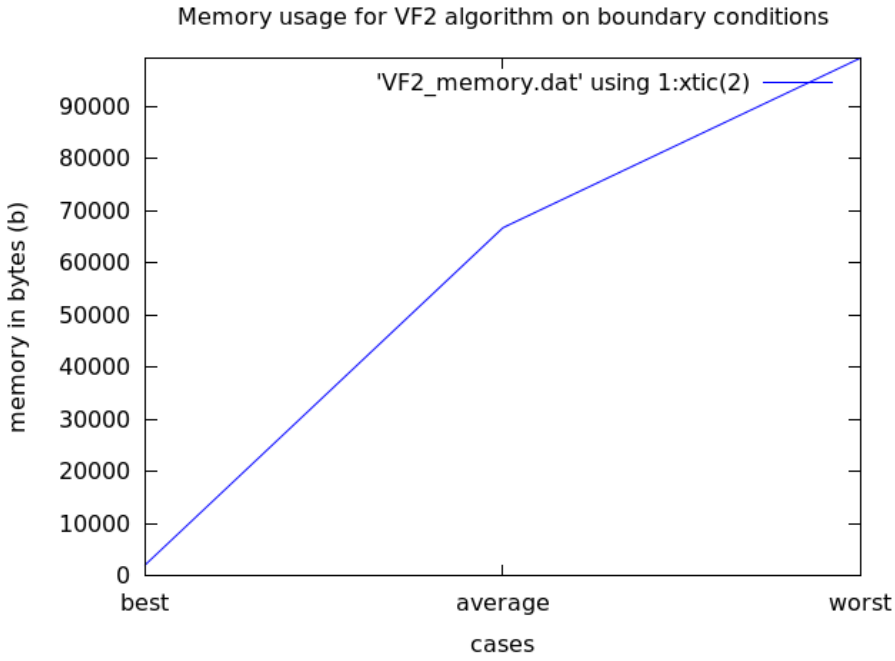


Figure 20: Representation of the memory usage by the VF2 algorithm on boundaries

5.3.4.1 Analysis of results Figure 20 depicts that amount of memory used by the VF2 algorithm during the execution of the graph matching procedure of the VF2 algorithm over time. The figure demonstrates the memory consumption characteristics of the algorithm on the defined boundary conditions.

Based on the results from 20, we can see that the amount of memory used by the VF2 algorithm, much like the Ullman algorithm also increases overtime.

5.3.5 Conclusion

The memory consumption characteristics demonstrated in figures 19 and 20 respectively have similary characteristics. The characteristic being that both algorithms seem to use very little amount of memory for the best case of the boundaries, but as the case of the boundaries becomes gradually worse, the more memory is required by both algorithms.

Apart from the forementioned characteristic, another observation that is made is that the memory used by the algorithms seems to be arbitrarily similary for each instance of their respective executions.

Best: condition, the Ullman algorithm uses 2136 bytes of memory and so does the VF2 algorithm.

Average: case, the Ullman algorithm uses 66780 bytes and the VF2 algorithm uses 66784 bytes.

Worst: case, Ullman uses 99344 and VF2 uses 99340 bytes. Thus it can be concluded from this observation that with regards to the memory used, the two algorthms are arbitrarily similar in that regards.

5.3.6 Time Boundary Results

This chapter depicts the amount of time taken by the algorithms in order to successfully complete the graph matching procedure, especially on the boundary conditions of the experiment.

5.3.7 Ullman Algorithm

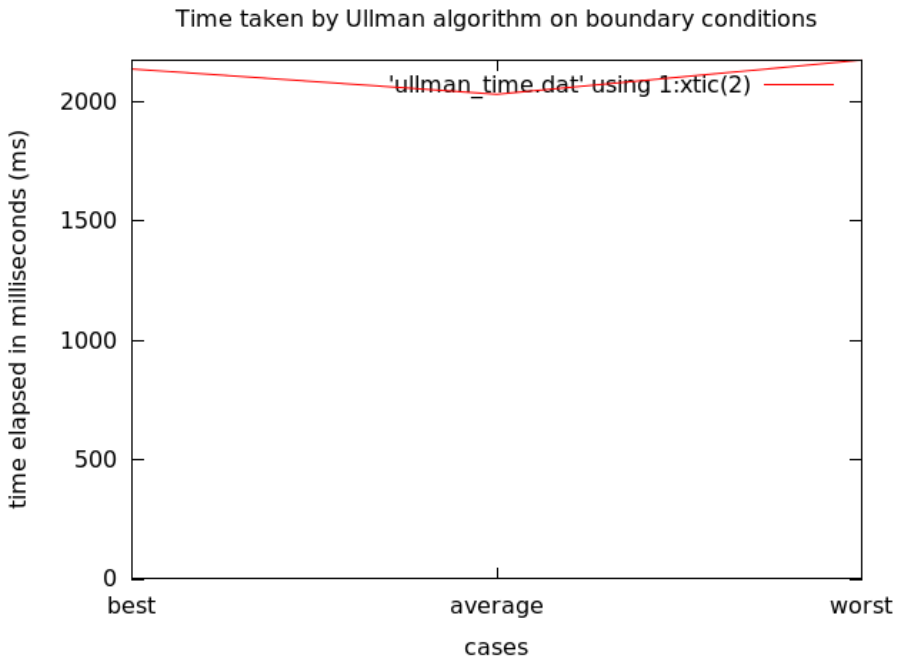


Figure 21: Representation of the time taken by the Ullman algorithm on boundaries

5.3.7.1 Analysis of results Figure 21 demonstrates the amount of time taken by the Ullman algorithm to successfully complete its graph matching procedure. From figure 21, we can see that the Ullman algorithm is extremely resource intensive will regards to the time it takes. Even in the **best**, the algorithm takes a relatively long time to complete its execution.

A secondary observation made from 21 is that, though the time values are high, they do appear to show very little flactuations, and are thus steady.

5.3.8 VF2 Algorithm

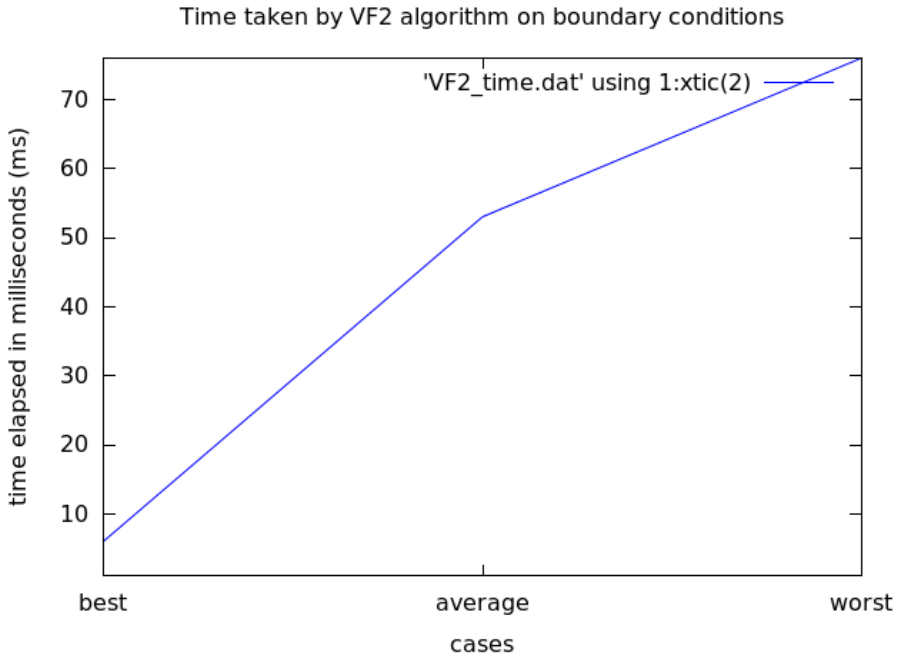


Figure 22: Representation of the time taken by the Ullman algorithm on boundaries

5.3.8.1 Analysis of results Figure 22 depicts the amount of time taken by the VF2 algorithm to successfully complete its graph matching procedure. From figure 22, it is clear that the amount of time taken to successfully complete the matching procedure increases over time, and the increase seems to be constant, from the **best** case to the worst.

5.3.9 Conclution

The Ullman and VF2 algorithms demonstrate very different behaviours with regards to the amount of time take to complete their respective matching procedures.

The Ullman algorithm requires a much larger amount of time for its completion, even in the **best** case, where the time required is 2136 ms.

The VF2 algorithm however, requires very little time relative to the Ullman algorithm, especially in the best case, as oppose to the Ullman algorithm. The time required by the algorithm does increase as the cases worsen, and based on figure 22,we can see that the increase is relatively constant.

By analysing the time requirements of both algorithms, we can safely deduce that the VF2 algorithm is castly more efficient than the Ullman algorithm in terms of the amount of time required

to complete the graph matching procedure. The values for each case indicating this fact are demonstrated in the table below.

Table 1: Time values for Ullman algorithm vs. VF2 algorithm

Case	Algorithm	
	Ullman	VF2
Best	2136	6
Average	2030	53
Worst	2173	76

6 Conclusion

This chapter introduces the result of the comparison of the algorithms based on chapter 5. The results in chapter 5 indicated that the two algorithms are arbitrarily similar with regards to the amount of memory used when performng the graph matching procedures of their respective executions.

The two algorithms perform very differently when it comes to their respective execution times. The Ullman algorithm requires a lot of time complete its exection, even in the best case scenrio where the conditions are favourable. The time required accross all different cases is relatively consistant.
The VF2 algorithm performs differently accross its test cases. The algorithm requires very little time to complete its graph matching procedures for the best cases, but the time required gradually increases as the cases become worse. Thus based on the presented evidence,the deduction that the VF2 algorithm is more efficient than the Ullman algorithm when considering the time required to complete its execution.

References

[1] J.R. Ullmann, *"An Algorithm for Subgraph Isomorphism, Journal of the Association for Computing Machinery"*, vol. 23, pp. 31-42, 1976.

[2] H. Ehrig, *"Introduction to Graph Grammars with Applications to Semantic Networks Computers"*, Math. Appl. 23, pp. 557-572, 1992.

[3] D.H. Rouvray, A.T. Balaban, *"Chemical Applications of Graph Theory in Applications of Graph Theory Academic Press"*, New York, pp.177-221, 1979.

[4] S. Melnik, H. Garcia-Molina and E. Rahm, *"Similarity flooding: a versatile graph matching algorithm and its application to schema matching"*, in Proceedings 18th ICDE, San Jose CA, Feb 2002.

[5] L. Cordella, P. Foggia, C. Sansone and M. Vento, *A (sub)graph isomorphism algorithm for matching large graphs IEEE Transactions on Pattern Analysis and Machine Intelligence"*, vol. 26, no. 10, pp. 13671372, Oct. 2004.

[6] *VF2 Algorithm(2015)*, Available at: `https://networkx.github.io/documentation/latest/reference/algorithms.isomorphism.vf2.html`, (Accessed: 10 April 2015).

[7] R. Diestel, *Graph Theory*, Graduate Texts in Mathematics, Vol. 173, Springer Verlag, Berlin, 1991.

[8] A. Drozdek, *Data Structures and Algorithms in Java*, Third Edition,Cengage Learning, pp. 386415, 2013.

[9] Reinhard Diestel, *Graph Theory*, Graduate Texts in Mathematics, Vol. 173, Springer Verlag, Berlin, 1991.

[10] *Graph Definitions*, Available at: `http://web.eecs.utk.edu/~leparker/Courses/CS302-Fall06/Notes/graph-defs-rev.html`, (Accessed: 10 June 2015).

[11] N.J. Nilsson, *Principles of Artificial Intelligence*, Springer-Verlag, 1982.

[12] M. Kubale and B. Jackowski, *A Generalization Implicit Enumeration Algorithm for Graph Coloring*,Communiations of the ACM, 28(4):412-418, 1985.

[13] *The Stony Brook Algorithm Repository*, Available at: `http://www3.cs.stonybrook.edu/~algorith/implement/vflib/implement.shtml`, (Accessed: 2 June 2015).

[14] *Random graph used in the paper "Optimization by Simulated Annealing: An Experimental Evaluation; Part II, Graph Coloring and Number Partitioning"*, Available at: `http://mat.gsia.cmu.edu/COLOR08/INSTANCES/DSJC250.1.col`, (Accessed: 30 June 2015).

[15] D. R. Cheriton , K. J. Duda *Logged virtual memory, Proceedings of the fifteenth ACM symposium on Operating systems principles*, p.26-38, December 03-06, 1995, Copper Mountain, Colorado, USA

[16] Weisstein,Eric W., *Empty Graph*,From MathWorldA Wolfram Web Resource. Available at: `http://mathworld.wolfram.com/EmptyGraph.html`,(Accessed: 10 April 2015)

[17] Linda Marshall, *A graph-based framework for comparing curricula.*, PhD thesis, University of Pretoria, 2014.