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

August 10, 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.

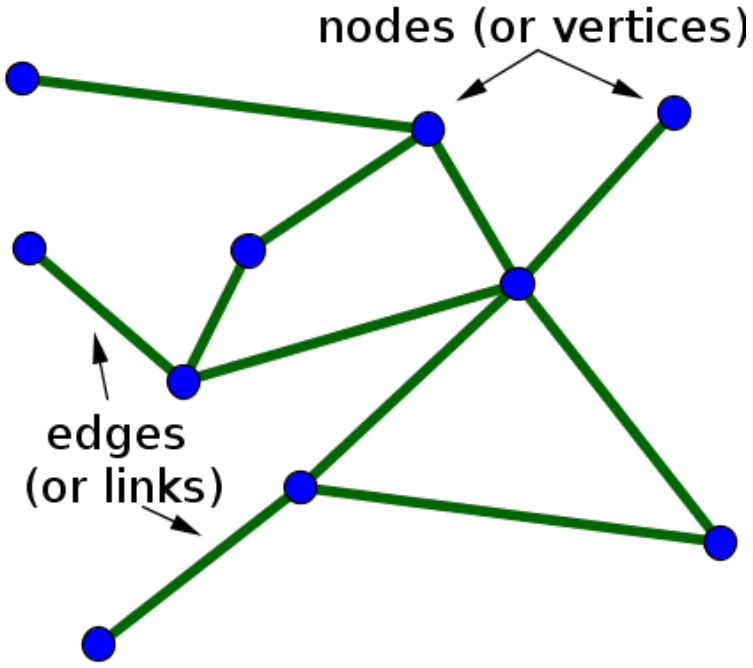
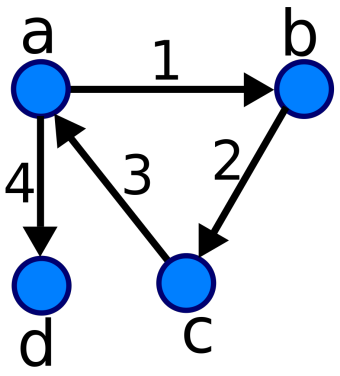


Figure 1: Representation of a graph

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. The figure below demonstrates a directed graph, commonly known as a digraph. This characteristic is demonstrated by the edge 1, that goes from node a to node b, and also by edges 3 that goes from node c to node a.



	1	2	3	4
a	1	0	-1	1
b	-1	1	0	0
c	0	-1	1	0
d	0	0	0	-1

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 goiing 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 Digraph

A Digraph in mathematics is defined as is a pair of vertices and edges (V,E) that are disjoint, and their repective mappings that comprise 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.

1.3 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.

1.3.1 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 A where there is an absence of an edge E are denoted by 0.

Figure 3 depicts the association between a graph and its adjacency matrix.

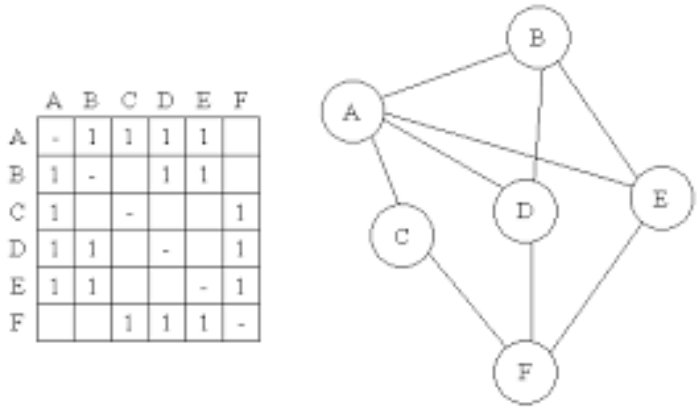


Figure 3: Representation of a graph and its associated adjacency matrix

1.3.2 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 connectect them to other vertices). Figure 4 depicts an example of an adjacency list.

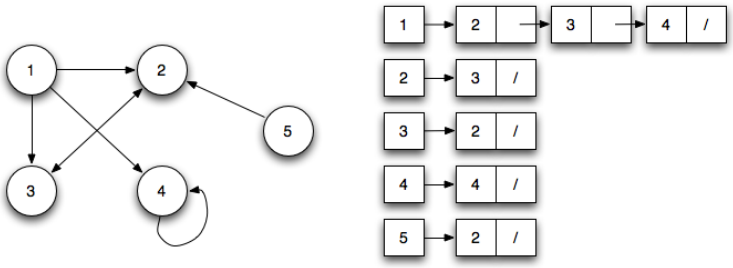


Figure 4: Representation of a graph and its associated adjacency list

1.4 Supergraphs and subgraphs

Let G_A be be graph define as follows G_A = (VA,EB) and let G_B be another graph that is defined as follows G_B = (VB,EB) where VA,VB are sets of vertices and EA,EB 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) VA ¡= VB, and 2) Every edge of G_A is also an edge in G_B.

The figure 5 below depicts this relation.

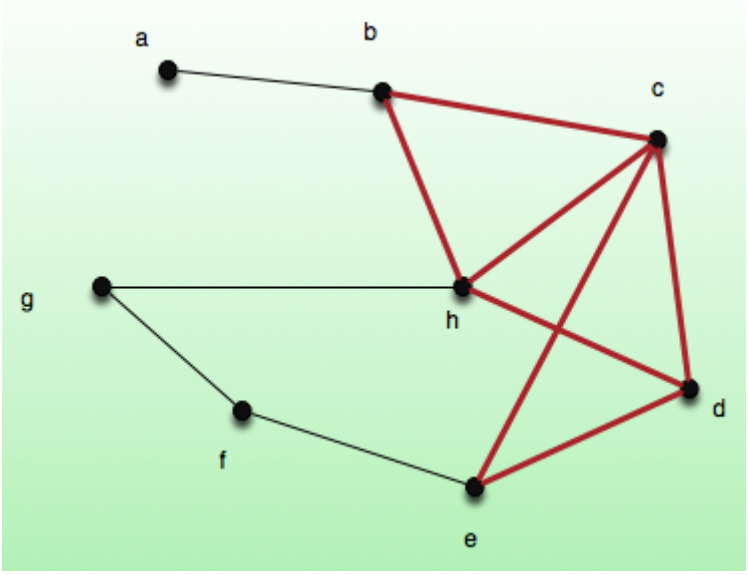


Figure 5: Representation of a super graph and its subgraph that is depicted in red

1.5 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 G1 correspond to exactly one edge of G2, and vice versa[12], i.e. the graphs are structurally the same to each other. This property is demonstrated in figure 6. The two graphs look very different, but when they are further inspected, it is evident that the two are a representation of the same data scheme or even the same graph that has been rearranged.

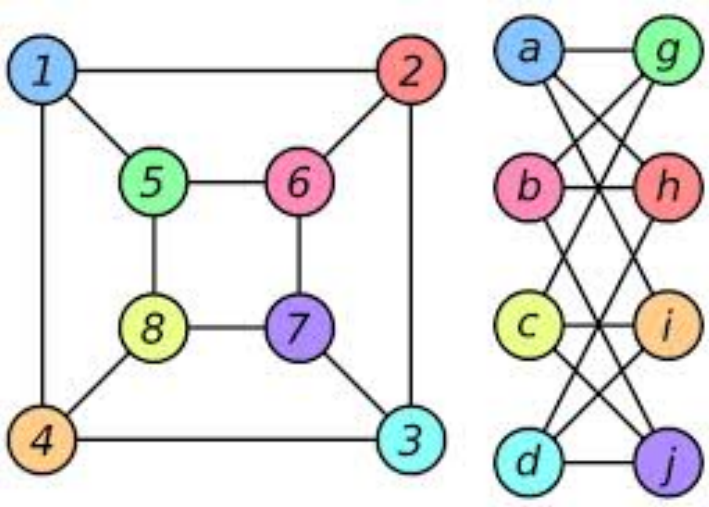


Figure 6: Representation of a super graph and its subgraph that is depicted in red

2 Ullman Algorithm

2.1 Background

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 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 Vn into the graph, thus the total number of subgraphs that are 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 and 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.

2.2 Algorithm

he Ullman algorithm was developed by J.R.Ullman and was published in his 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 nxm 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 M0ij 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 Md 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.

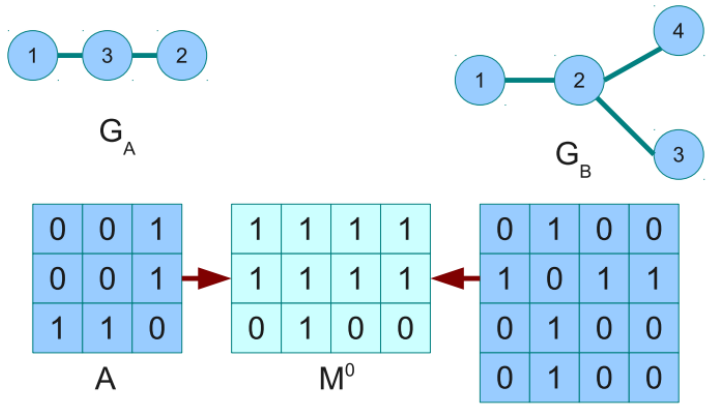


Figure 7: Demonstation of how a permutation matrix is generated from two graphs

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=Mn(Mn . G_A)T, where G_A = input graph A Mn = permutated matrix Mn in Md, obtained from the starting matrix M0 (Mn . G_A)T = the transpose of the dot product of the permutated matrix Mn and the graph G_A. If there is a single instance of the matrix C, that is calculated using some permutated matrix Mn 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_B 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.

3 VF2 Algorithm

3.1 Background

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.

3.2 Algorithm

3.2.1 Overview

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.

3.2.2 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¡=G_A and m¡=G_B. The isomorphism determining properties of the mapping are defined as follows, a mapping M ¡ NA X NB is isomorphic iff M is a bijection, that preserves the branching structure of G_A and G_B, where NA is a set of vertices from G_A and NB a set of vertices from G_B.

The mapping M ¡ NA X NB is subgraph isomorphic iff M is isomorphic to G_A and a subgraph of G_B.

3.2.3 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 asthe 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].

3.2.4 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(pow(k)) are consistant with the desired morphisms.

3.2.5 Condidate Pairs

The candidate pairs are obtained by considering the vertexs 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 $T_A in(s)$ 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 $T_B in(s)$ 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 $T_A out(s)$ 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 $T_B out(s)$ 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 $T_A out(s)$ and m from $T_B out(s)$. If the any of the sets is empty, then we consider the vertex n from $T_A in(s)$ and m from $T_B in(s)$. 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.

3.2.6 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 A more formal definition is as follows:

$$Fsyn(s, n, m) = Rpred \wedge Rsucc \wedge Rin \wedge Rout \wedge Rnew \tag{7}$$

A more formal definition of each rule is provided below.

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. This brings us to the conclusion that, in order for a state to be considered consistent it must pass a combination of all of the five rules, namely:

$$\begin{aligned} &Rpred(s,n,m) \Longleftrightarrow \\ &\quad (\forall n' \in M1(s) \cap Pred(GB,n) \exists m' \in Pred(GA,m) | (n',m') \exists M(s) \wedge \\ &\quad\quad \forall m' \in M2(s) \cap Pred(GA,m) \exists n' \in Pred(GB,n) | (n',m') \exists M(s)), \end{aligned} \tag{8}$$

$$\begin{aligned} &Rsucc(s,n,m) \Longleftrightarrow \\ &\quad (\forall n' \in M1(s) \cap Succ(GB,n) \exists m' \in Succ(GA,m) | (n',m') \exists M(s) \wedge \\ &\quad\quad \forall m' \in M2(s) \cap Succ(GA,m) \exists n' \in Succ(GB,n) | (n',m') \exists M(s)), \end{aligned} \tag{9}$$

$$\begin{aligned} &Rin(s,n,m) \Longleftrightarrow \\ &\quad (Card(Succ(GB,n) \cap TBin(s)) \geq Card(Succ(GA,m) \cap TAin(s))) \cap \\ &\quad\quad (Card(Pred(GB,n) \cap TBin(s)) \geq Card(Pred(GA,m) \cap TAin(s))) \end{aligned} \tag{10}$$

$$\begin{aligned} &Rout(s,n,m) \Longleftrightarrow \\ &\quad (Card(Succ(GB,n) \cap TBout(s)) \geq Card(Succ(GA,m) \cap TAout(s))) \cap \\ &\quad\quad (Card(Pred(GB,n) \cap TBout(s)) \geq Card(Pred(GA,m) \cap TAout(s))) \end{aligned} \tag{11}$$

$$\begin{aligned} &Rnew(s,n,m) \Longleftrightarrow \\ &\quad Card(\tilde{NA}(s) \cap Pred(GB,n)) \geq Card(\tilde{NB}(s) \cap Pred(GA,n)) \wedge \\ &\quad\quad Card(\tilde{NA}(s) \cap Succ(GB,n)) \geq Card(\tilde{NB}(s) \cap Succ(GA,n)) \end{aligned} \tag{12}$$

This, as described earlier is defined as the feasibility function.