A Performance Analysis on Maximal Common Subgraph Algorithms

Ruud Welling
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
r.h.a.welling@student.utwente.nl

ABSTRACT

Graphs can be used as a tool to determine similarity between structured objects. The maximal common subgraph of two graphs G and H is the largest graph in terms of edges that is isomorphic to a subgraph of G and H. Finding the maximal common subgraph is an NP-complete problem. It is useful in many areas like (bio)chemistry, file versioning and artificial intelligence.
There are many papers that evaluate algorithms for finding maximal common induced subgraphs, but little research has been done on the maximal common subgraph that is not an induced subgraph. We have implemented and benchmarked two maximal common (not induced) subgraph algorithms: a backtrack search algorithm (McGregor), and an algorithm that transforms the maximal common subgraph problem to the largest clique problem (Koch). We created generators for randomly connected and mesh structured graphs, these generators have been used to create a database of graph pairs to benchmark the two algorithms.
The results of our benchmark have shown that in most cases Koch is more efficient, because after creating the edge product graph needed for the clique detection. The actual clique detection is a relatively simple search.

1. INTRODUCTION

Graphs are data structures that can represent structured objects, concepts or models. Determining similarity between two graphs is equivalent to determining the similarity between the structured objects, concept or models that the graphs represent [2]. When two graphs have subgraphs that are isomorphic, then these subgraphs are called common subgraphs. A maximal common subgraph is a common subgraph which has the maximal number of edges, in other words, if s is a common subgraph of graphs g and h, and there is no other common subgraph which has more edges than s, then s is a maximal common subgraph of g and h [12].

Finding the maximal common subgraphs of two graphs is important in many applications. For example comparing protein structures [6], chemistry [10, 11], file versioning [3, 8] and machine learning [2].

15th Twente Student Conference on IT June 20th, 2011, Enschede, The Netherlands.

The problem of finding the maximal common subgraph is NP-complete [2, 5]. This makes it an interesting computer science problem. We have benchmarked the performance of two maximal common subgraph algorithms: a backtrack search algorithm by McGregor [7] and a clique detection algorithm by Koch [5], which can be used to find the maximal common subgraph. We have created graph generators to generate a database of graph pairs to benchmark our algorithms on. Our results can be found in tables 1 and 2 and the tables in Appendix A.

An analysis similar to ours has been done already by Bunke et al. [1] and Conte et al. [2], however Conte et al. and Bunke et al. have benchmarked algorithms that search for the maximal common induced subgraph only. We have focussed on finding the maximal common subgraph which is the subgraph with the maximal number of edges. In section 5 we will compare our results with the results of Conte et al. and Bunke et al.

2. PRELIMINARIES

To introduce the maximal subgraph problem, we will first define what graphs, subgraphs and maximal common subgraphs are. We will then explain how the maximal common subgraph problem can be transformed to the maximal clique problem. In the last subsection we will describe several graph categories we have used to benchmark the algorithms.

2.1 Maximal Common Connected Subgraph

There are two different definitions of the maximal common subgraph. This section will explain explain the differences and which kind of maximal common subgraph we have chosen to focus on.

Definition 1. (based on [2]) A graph *is a 4-tuple* $G = (V, E, \alpha, L)$*, where*

- V *is the finite set of vertices*
- $E \subseteq V \times V \times L$ *is the set of edges*
- $\alpha : V \to L$ *is a function assigning labels to the vertices*
- L *is a finite nonempty set of labels for the edges and vertices*

An edge (u, v, l) is from vertex u to vertex v and has label l. Undirected graphs have an edge (v, u, l) for every edge $(u, v, l) \in E$ [2].

Two vertices u and v in graph G are *adjacent* if an edge (u, v, l) or (v, u, l) exists in G for at least one $l \in L$. There is a *path* from vertex u to v if there exists sequence of

vertices starting at u and ending at v, such that every vertex in the sequence is adjacent to the next vertex in the sequence.

A graph G is *connected* if there is a path (ignoring edge direction) between any two distinct vertices of G [4]. Our research will be limited to maximal common subgraphs that are connected. This is because non-connected common subgraphs could consist of many disconnected edges and their largest connected component could be much smaller than the maximal connected common subgraph. Because of this, only connected common subgraphs are relevant in many applications [5].

Definition 2. (based on [2]) Let $G = (V, E, \alpha, L)$ and $G' = (V', E', \alpha', L')$, be graphs; G' is a subgraph *of G, $G' \subseteq G$, if*

- $V' \subseteq V$
- $\alpha(v) = \alpha'(v)$ *for all* $v \in V'$
- $E' \subseteq E \cap (V' \times V' \times L')$
- $L' \subseteq L$

Special cases of the subgraph are the *induced subgraph* and the *clique*. An *induced subgraph* preserves all edges between between the vertices in the subgraph: $E' = E \cap (V' \times V' \times L')$ [2]. A *clique* in graph G is a subgraph of G that is a complete subgraph (a graph where every vertex is connected to every other vertex with an edge in both directions, these edges may have any label) [4].

Definition 3. (based on [2]) Let G and G' be graphs. A graph isomorphism *between G and G' is a bijective mapping $f : V \to V'$ such that*

- $\alpha(v) = \alpha'(f(v))$ *for all* $v \in V$
- *for any edge* $e = (u, v, l) \in E$ *there exists an edge* $e' = (f(u), f(v), l') \in E'$ *such that* $l = l'$

Graph isomorphism is a bijective mapping, this implies that for any edge $e' = (u', v', l')$ there exists an edge $e = (f^{-1}(u'), f^{-1}(v'), l) \in E$ such that $l = l'$.

If $f : V \to V'$ is a graph isomorphism between graphs G and G', and G' is a subgraph of another graph G'', i.e., $G' \subseteq G''$, then f is called a *subgraph isomorphism* from G to G''. [2]

Definition 4. (based on [2, 12]) Let $G_1 = (V_1, E_1, \alpha_1, L_1)$ and $G_2 = (V_2, E_2, \alpha_2, L_2)$ be graphs. A common subgraph *of G_1 and G_2 is a graph $G = (V, E, \alpha, L)$ such that there exist subgraph isomorphisms from G to G_1 and from G to G_2. We call G a* maximal common subgraph *of G_1 and G_2 if there exists no other common subgraph of G_1 and G_2 that has more edges than G.*

Definition 4 is the definition for the maximal common subgraph used in this paper. A different, very commonly used definition of the maximal common subgraph is what we call the *maximal common induced subgraph*. The latter is the common induced subgraph of two graphs that has the largest number of vertices. The difference between these two kinds of maximal common subgraphs is illustrated in figure 1.

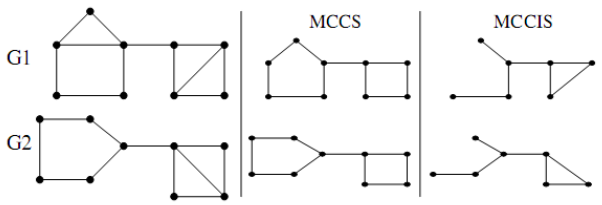

Figure 1. The difference between a maximal common connected subgraph (MCCS) and a maximal common connected induced subgraph (MCCIS) [12]. Labels and directions are not shown in this example. Every edge can be considered bidirectional, and all vertices and edges have the same label.

2.2 Maximal Clique

One of the algorithms we have benchmarked to solve the maximal subgraph problem attempts to solve the problem by transforming it to the maximal clique problem. This can be done by constructing an *edge product graph.*

Definition 5. (based on [5]) Let $G_1 = (V_1, E_1, \alpha_1, L_1)$ and $G_2 = (V_2, E_2, \alpha_2, L_2)$ be graphs. The edge product graph *$H_e = G_1 \circ_e G_2$ includes the vertex set $V_H = E_1 \times E_2$, in which all edge pairs (e_i, e_j) with $1 \leq i \leq |E_1|$ and $1 \leq j \leq |E_2|$ have to coincide in their edge labels and the corresponding end vertex labels. Let $e_i = (u_1, v_1, l_1)$ and $e_j = (u_2, v_2, l_2)$, the labels coincide if $l_1 = L_2$ and $\alpha_1(u_1) = \alpha_2(u_2)$ and $\alpha_1(v_1) = \alpha_2(v_2)$*

There is an edge between to vertices $e_H, f_H \in V_H$ with $e_H = (e_1, e_2)$ and $f_H = (f_1, f_2)$ if the two edge pairs are compatible, meaning that $e_1 \neq f_1$ and $e_2 \neq f_2$, and either

- *e_1, f_1 in G_1 are connected via a vertex of the same label as the vertex shared by e_2, f_2 in G_2 (labeled and called a* c-edge*), or*
- *e_1, f_1 and e_2, f_2 are not adjacent in G_1 and in G_2, respectively (labeled and called a* d-edge*).*

To get a common subgraph in G_1 and G_2 each edge pair in G_1 and G_2 (vertex pair in H_e) has to be compatible to all other edge pairs in G_1 and G_2 (edges in H_e), which are forming a common subgraph. Thus, a clique in H_e corresponds to a common subgraph in G_1 and G_2. [5]

If this clique is spanned by c-edges (there is a path from every vertex in H_e to every other H_e consisting of only c-edges), then the clique corresponds to a connected common subgraph. The maximal clique corresponds to the maximal common subgraph.

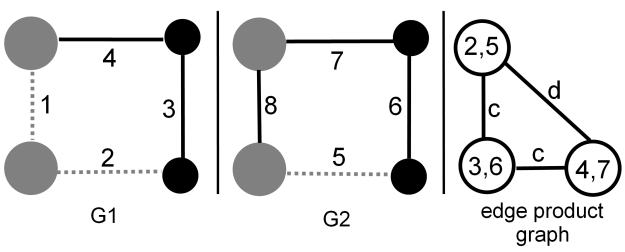

Figure 2. Two graphs $G1$ and $G2$ with their edge product graph. In this example, the size and colour of a vertex, and the style of an edge line (dotted or not) identify the labels.

2.3 Categories of graphs

We have benchmarked our algorithms on a randomly generated database of graphs. These graphs are generated to form some different structures. We have chosen for these structures to be able to compare the performance of the algorithms for graph that are more structured or less structured. These graph categories will be introduced in this section.

Randomly connected graphs are connected graphs where edges connect vertices without any structural regularity . In these graphs, the probability that there exists an edge connecting two distinct vertices is independent the vertices themselves. [2].

Regular Meshes are structured graphs. In a 2D mesh, every vertex (except for those at the border of the mesh) is connected with its 4 neighbour vertices. In 3D or 4D meshes, every vertex is connected with all 6 or 8 neighbour vertices. [2].

3. EXPERIMENTAL SETUP

The research will be a performance analysis of two commonly used graph algorithms. The algorithm by Koch [5], which uses clique detection, has proven to be efficient. The backtrack search algorithm by McGregor [7] is very widely used. Many other algorithms exist, but they all use an approach similar to these algorithms.

This section will give an overview of how the two algorithms work. We will then explain how we created our graph generators. In order to benchmark the algorithm, we created a database we will explain how this database had been created. Finally we will explain how we executed the benchmarks.

3.1 Algorithms

We have selected two algorithms. The algorithm by McGregor [7] is a backtrack search algorithm. The algorithm by Koch [5] is an algorithm that transform the maximal common subgraph problem to the maximal clique problem and searches for the maximal clique using a branch-and-bound algorithm.

3.1.1 Koch

In the Koch algorithm, several sets are remembered: The set C contains all vertices that belong to the clique which the algorithm is currently expanding. The set P contains vertices that can directly be added to C, because these vertices are adjacent to all vertices in C and adjacent to at least one of the vertices in C via a c-edge. The set D contains vertices that can not directly be added, they are adjacent to all vertices in C via d-edges.

Algorithm 2 is the recursive clique search algorithm. It checks if set C can be expanded, if so then the algorithm will expand try to expand the set for every vertex in P (algorithm 2, line 5), the algorithm recursively calls itself for every expansion of the set C.

In order to initialize all sets in the Koch algorithm there is another algorithm. Algorithm 1 initializes the sets C, P and D for every vertex in the edge product graph.

Algorithm 1 MAXIMAL_C_CLIQUE(), the initialization algorithm for algorithm 2 [5]

▷ returns the set R containing the vertices of the maximal clique in a graph G
T: set of vertices which have already been used for the initialization of EXPAND_C_CLIQUE V: set of all vertices in edge product graph G
C: set of vertices belonging to the current clique
P: set of vertices which can be added to C, because they are neighbours of all vertices $\in C$, and for every vertex $u \in P$ there exists at least one other vertex $v \in C$ such that u and v are connected via a c-edge
D: set of vertices which cannot directly be added to C, because they are neighbours of all vertices $\in C$ via d-edges
E: set of vertices resulting from a recursive call of EXPAND_C_CLIQUE
$largest$: the size of the largest clique found so far
$N[u] = \{v \in V | \{u, v\} \in E\}$ this is the set of neighbours of a vertex u in G

```
1: T ← ∅
2: R ← ∅
3: largest ← 0
4: for all u ∈ V do
5:    P ← ∅
6:    D ← ∅
7:    for all v ∈ N[u] do
8:       if v and u are adjacent via a c-edge then
9:          if v ∉ T then
10:            P ← P ∪ {v}
11:         end if
12:      else if v and u are adjacent via a d-edge then
13:         D ← D ∪ {v}
14:      end if
15:   end for
16:   E ← EXPAND_C_CLIQUE({u}, P, D, largest)
17:   if |E| > |R| then
18:      R ← E
19:      largest ← |E|
20:   end if
21:   T ← T ∪ {u}
22: end for
23: return R
```

Algorithm 2 EXPAND_C_CLIQUE($C, P, D, largest$) [5]

▷ returns the set R such that $C \subseteq R$ and R contains the vertices of the maximal clique (limited to the vertices in C) in a graph G

```
1: R ← C
2: if P = ∅ OR (|P| + |C| + |D| ≤ largest) then
3:    return R
4: else
5:    for all u ∈ P do
6:       P ← P \ {u}
7:       P' ← P ∩ N[u]
8:       D' ← D ∩ N[u]
9:       for all v ∈ D' do
10:         if v and u are adjacent via a c-edge then
11:            P' ← P' ∪ {v}
12:            D' ← D' \ {v}
13:         end if
14:      end for
15:      E ← EXPAND_C_CLIQUE(C ∪ {u}, P', D', largest)
16:      if |E| > |R| then
17:         R ← E
18:         largest ← |E|
19:      end if
20:   end for
21: end if
22: return R
```

The algorithm we use is slightly modified from the original algorithm by Koch. The original algorithm enumerates all cliques in the edge product graph, we only need to find the maximal clique. Therefore we have implemented a test: we remember the size of the maximal clique found so far in *largest*, this is supplied as an argument for the EXPAND_C_CLIQUE procedure. At the beginning of this procedure (algorithm 2, line 2) we check if the sizes of the sets C, P and D together are bigger than *largest*. If this total size is not bigger than *largest* then the current clique can never become the largest and the algorithm can stop expanding C. This check can significantly reduce the size of the recursion tree, allowing the algorithm to run much faster. In the optimal case, the maximal clique is the first clique that is found. When looking for other cliques the size of $|P| + |C| + |D|$ (which is the maximum size the current clique may become) will eventually be smaller than or equal to the size of the largest clique. The algorithm will immediately stop expanding the current clique and try another, leading to a significant reduction of the recursion tree.

3.1.2 McGregor

The McGregor algorithm [7] attempts to tentatively pair vertices from G_1 to vertices from G_2. A matrix *medges* keeps track of which edges of G_1 and G_2 might still correspond to each other. Every time a vertex G_1 and a vertex from G_2 are tentatively paired, *medges* is refined. For example when vertex i from G_1 is tentatively paired with vertex j in G_2, then any edge r connected to vertex i can correspond only to edges which are connected to vertex j in G_2.

The call allPossibleVerticesPaired() (line 16) returns true when there are no more unpaired vertices left with a similar label, this means that no more vertices can be tentatively paired. The state of *medges* at this point represents the edges of the common subgraph.

In order to reduce the size of the search tree, the algorithm checks if the number of edges left in medges is still higher then the number of edges in the best result so far. The call getEdgesLeft() (line 14) returns the number of edges that the current common subgraph may have at maximum, this is the number of rows in *medges* that contain at least one 1. To find only connected subgraphs the call getConnectedEdgesLeft() (line 17) returns the number of edges in the largest connected graph generated by the edges that are 1 in *medges*.

The algorithm keeps track of the number of vertices that have been tentatively paired for each label. This is because at some point, it may be possible that for a vertex i in G_1 there is no unpaired node j in G_2 with the same label. The noLabelMatch flag (line 10, 30) keeps track of this special case. When this flag is set, it means that there are still untried vertices for vertex i, but with a different label. This prevents the algorithm from backtracking when there are still vertices to be paired.

Backtracking is simple, whenever two vertices have been tentatively paired a copy of *medges* is stored in the workspace associated with the next vertex i, when the algorithm must backtrack, the current vertex value is decremented to vertex v, *medges* is restored from workspace associated with vertex v.

3.1.3 Implementation

The two algorithms have been implemented in Java using the JGraphT library [9]. The graphs in JGraphT use LinkedHashSets in which the order of elements of the set remains constant. Our implementations of the two algorithms use ArrayLists and LinkedHashSets to contain vertices and edges. Since the order of these sets is always the same, the algorithms are deterministic. For benchmarking this is important because this greatly reduces the variance of subsequent benchmark results.

Algorithm 3 MCGREGOR(G_1, G_2) [7]

▷ returns the maximal common subgraph of G_1 and G_2
V_1: the set of vertices of G_1
V_2: the set of vertices of G_2
E_1: the set of edges of G_1
E_2: the set of edges of G_2
medges: a boolean matrix, $medges[d][e]$ is true of edge d in G_1 is permitted to correspond to edge e in G_2
medgesCopies[i]: an array that stores copies of medges, these copies are restored when the algorithm backtracks
$T[i]$: the set of vertices from G_2 that have been tried for vertex i from G_1
noLabelMatch[i]: a boolean flag, corresponding to vertex i from G_1, initialized to false

```
1: let a = (v_a, u_a, l_a) and b = (v_b, u_b, l_b), set medges[a][b]
   to contain l_a = l_b for all a ∈ E_1 and all b ∈ E_2
2: i ← 0
3: bestEdgesLeft ← 0
4: T[i] ← ∅
5: while i ≥ 0 do
6:   if |T[i]| < |V_2| then
7:     xi ← getUntriedVertex(i)
8:     T[i] ← T[i] ∪ {xi}
9:     if α_1(i) ≠ α_2(xi) then
10:      noLabelMatch[i] ← true
11:    else
12:      medgesCopies[i] ← medges
13:      refineMedges(i, xi)
14:      edgesLeft ← getEdgesLeft()
15:      if edgesLeft > bestEdgesLeft then
16:        if allPossibleVerticesPaired() then
17:          if medges.getConnectedEdgesLeft()
             > bestEdgesLeft then
18:            bestMedges ← medges
19:            bestEdgesLeft ← edgesLeft
20:          end if
21:        else
22:          i ← i + 1
23:          medgesCopies[i] ← medges
24:          T[i] ← T[i] ∪ {xi}
25:        end if
26:      else
27:        medges ← medgesCopies[i]
28:      end if
29:    end if
30:  else if noLabelMatch[i] and i ≠ |V_1| − 1 then
31:    noLabelMatch[i] ← false
32:    i ← i + 1
33:    medgesCopies[i] ← medges
34:    T[i] ← T[i] ∪ {xi}
35:  else
36:    i ← i − 1
37:    medges ← medgesCopies[i]
38:  end if
39: end while
```

3.2 Graph Generators

In order to get a dataset to benchmark the algorithm on we have chosen to use graph generators to create a dataset for

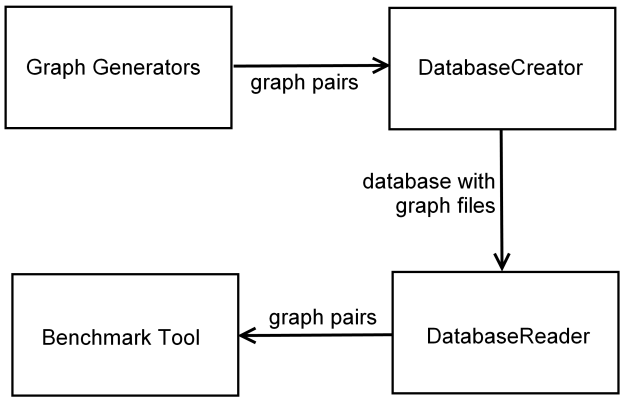

Figure 3. The tools we used, from graph generation to benchmark

our benchmarks. Another possibility would to use existing graphs from applications where finding the maximal common subgraph is relevant. However gathering this data is extremely time consuming, therefore we chose to use graph generators to create our dataset.

We have written two graph generators: one generates randomly connected graphs, the other generates regular mesh graphs. Both generators take the *supergraph size* N (in edges), *subgraph size* s (as a percentage of the supergraph size), and *label alphabet size* L as parameters. The graph generators have the possibility to to ensure that the pair of supergraphs generated have a maximal common subgraph which is exactly the specified size, by testing the maximal common subgraph size with one of the algorithms.

The randomly connected graph generator generates graphs with a fixed number of vertices and edges. The number of edges is defined in the total size and subgraph size arguments. The number of vertices is defined by the *edge density*. The edge density is the percentage of edges from the complete (undirected) subgraph that should be in the graph. The number of vertices $|V|$ for a graph with $|E|$ edges and an edge density of η is calculated as follows: $|V| = \sqrt{\frac{2|E|}{\eta}}$. If $|V| > |E| + 1$ then it is not possible to generate a graph for the provided values of $|E|$ and η. The graph generator ensures that the resulting graphs are connected.

The regular mesh generator has a dimension d as an extra parameter. Every vertex will have a maximum of $2 \cdot d$ neighbour vertices, in a 2d mesh, every vertex is connected to its north, east, south and west neighbour (if these exist), this way a structured graph is generated.

3.3 Database

The graph generators were used to generate a database. As values for the supergraph size we chose $\{10, 20, 30\}$, for the label alphabet the values $\{10, 20, 40, 60\}$ were chosen. The values $\{20\%, 40\%, 70\%, 90\%\}$ were chosen as maximal common subgraph sizes. For regular meshes, the dimensions $\{2, 3, 4\}$ were chosen. Randomly connected graphs have edge densities $\{0.1, 0.25, 0.5\}$. Some randomly connected graphs could not be generated, because for small graphs, it is not always possible to guarantee a low edge density. For every set of parameters, we have generated 10 graph pairs. The database contains a total of 2040 graph pairs.

It should be noted that the values for the supergraph sizes are not bigger because the running times of the algorithms could become very high in some cases. In our preliminary benchmarks there were some specific cases with graph with 40 edges and few labels where the running time of the algorithm would be in the order of hours. We could not include larger sizes because the total time it would take to complete a full benchmark would become too long.

3.4 Test Setup

A benchmark program has been created, which runs each algorithm 10 times on every graph pair. To prevent a high variance the test program calculates the relative standard deviation $\frac{\sigma}{average}$. If the relative standard deviation is higher then 0.20, the test is run again until the relative standard deviation is low enough. All results are stored in a comma separated value file.

While running the benchmark, we discovered that for some graph pairs, the algorithms could take extremely long to finish. Therefore we implemented a time-out. if a graph algorithm does not find any results before the time-out, then the algorithm is stopped and the next test is started.

4. BENCHMARK RESULTS

For every graph pair, both algorithms have been benchmarked ten times (unless the benchmark could not complete before the time-out). Since the variance is low for these results, we have calculated the mean value for the execution time for each algorithm for all graph pairs. For every set of parameters for the graph generator, there were ten graphs of this type in the database. Even for graphs generated with the same parameters, the variance of the algorithm execution times may be quite high in some cases. Therefore we have calculated the median value for both algorithms for each graph parameter set. The median value is a measure where 50% of the graph pairs of the same type would take longer to execute, and the other 50% would execute faster. In order to see which parameters have most effect on the algorithm performance we have calculated a ratio $r = \frac{\text{time of McGregor algorithm}}{\text{time of Koch algorithm}}$. This ratio makes comparing the performance easier. Tables 1 and 2 show the ratios for all generated graph pairs. In table 1 a ' - ' means that this graph pair was not in the database, because the subgraph was too small to be able to create a graph for the specified edge density. There were some cases where more than half of the tests could not be completed before the time-out for the McGregor algorithm, the table shows *>9999* for these cases.

In the two tables, the cells where the ratio is lower then 1 are coloured, these represent the graph pairs where McGregor performs faster than Koch. We notice that McGregor only performs better in situations where the subgraph percentage is 70% or 90%. For the randomly connected graphs, we see that a higher edge density, leads to better results for the McGregor algorithm. This is most likely because a higher edge density means fewer vertices (since the number of edges is constant). The McGregor algorithm attempts to pair vertices of the two supergraphs, the fewer vertices there are to pair, the less time it will take to try all possible pairs. So the McGregor algorithm will finish faster.

The label alphabet size also has some influence on the ratios: for a smaller label size, the Koch algorithm performs relatively better than for large label sizes. This is most likely because in the McGregor algorithm, only vertices with the same label are tentatively paired. When there are more labels, the chance that two vertices have the same label is smaller. This results in less pairs of vertices that may be paired, which leads to a faster search. For

Table 1. Randomly connected graphs. In this table N is the total size of the two supergraphs, s is the size of the subgraph in relation to the supergraph, L is the size of the label alphabet, η is the edge density. The results in this table are the ratio's between the time taken to calculate the result for the graph pair. Let t_K be the time needed for the Koch algorithm to find a result, and let t_M be the time needed for the McGregor algorithm to find a result, the ratio r will then be $r = \frac{t_M}{t_K}$. If $r > 1$ then the Koch algorithm is faster for graphs generated with the parameters of r.

	N	10		20			30			
η	$L \setminus s$	70%	90%	40%	70%	90%	20%	40%	70%	90%
0.1	10	-	-	-	-	-	-	-	-	171.9
	20	-	-	-	-	-	-	-	-	>9999
	40	-	-	-	-	-	-	-	-	14.34
	60	-	-	-	-	-	-	-	-	2.811
0.25	10	-	-	-	2.885	2.504	-	206.0	5.650	0.013
	20	-	-	-	1.921	1.488	-	8.258	2.572	0.019
	40	-	-	-	1.941	0.963	-	5.252	1.289	0.609
	60	-	-	-	1.752	1.043	-	4.644	1.202	0.652
0.5	10	2.011	1.752	2.696	1.006	0.099	9.543	1.873	0.055	0.001
	20	1.929	1.459	4.197	0.975	0.585	8.066	2.510	0.441	0.011
	40	1.921	1.313	2.828	0.901	0.532	7.945	2.395	0.584	0.346
	60	2.028	1.305	2.995	0.889	0.482	6.936	2.182	0.614	0.307

Table 2. Regular meshes. The results in this table are the ratio's between the time taken to calculate the result for the graph pair. The value d stands for the dimension of the mesh.

	N	10				20				30			
d	$L \setminus s$	20%	40%	70%	90%	20%	40%	70%	90%	20%	40%	70%	90%
2	10	13.93	9.135	4.369	2.807	148.9	12.29	3.228	0.740	>9999	11.46	1.915	0.044
	20	7.120	7.008	3.711	2.368	44.92	15.02	3.315	2.113	642.6	23.13	4.210	1.833
	40	7.130	6.329	3.100	2.279	24.77	8.919	3.383	1.591	77.62	14.00	3.509	1.381
	60	6.988	6.679	3.354	2.167	21.88	8.000	2.901	1.656	31.93	9.828	3.192	1.622
3	10	11.17	9.439	4.392	2.716	76.06	247.6	5.276	2.410	>9999	2869	0.574	0.003
	20	9.120	7.948	3.833	2.811	31.97	14.93	4.629	2.049	111.9	196.5	4.186	2.091
	40	8.913	6.420	3.996	2.549	30.60	9.731	3.440	1.901	51.78	13.85	3.769	1.493
	60	7.233	6.306	3.808	2.452	25.18	10.05	3.114	1.740	41.95	14.19	3.083	1.613
4	10	15.00	9.350	4.976	2.930	82.68	95.30	6.045	1.884	>9999	>9999	5.543	0.039
	20	10.56	8.068	3.438	2.403	39.01	16.03	4.097	2.183	102.7	60.31	4.550	2.339
	40	8.048	5.577	3.583	2.056	28.20	12.95	3.472	2.006	66.43	14.54	3.382	1.840
	60	7.362	6.584	3.631	2.352	28.14	10.74	3.543	2.218	45.81	13.70	3.585	1.809

the Koch algorithm all edge and vertex labels are checked during the construction of the product graph; more labels means a smaller product graph, which also results in a faster search. From our results in tables 1 and 2 we can see that the McGregor algorithm benefits more from larger label sizes in relation to the Koch algorithm.

The tables in appendix A show a selection of the exact running times of the algorithms. In tables 3-6 we can see that for the results with label alphabet sizes 40 and 60, there is little difference in the running times. It seems that when the label alphabet size becomes significantly larger than the graph size, a change in the label alphabet size has little to no effect on the running time.

We can see in tables 8-10 that a larger subgraph percentage has a negative influence on the performance of the Koch algorithm. This is most likely because the edge product graph that is created for these graph pairs is very connected, so many more potential cliques must be evaluated. The recursion tree for this algorithm becomes significantly bigger causing longer running times for larger subgraph percentages. For the McGregor algorithm a larger subgraph percentage does not have a clear effect on the running times of the algorithm.

We see that the algorithm by Koch performs better in most cases. This is because during the construction of the edge product graph, the edge and vertex labels are checked, and all incompatibilities are eliminated. The labels greatly reduce the size of the product graph, and they are only evaluated during the construction of the product graph. It appears that the time it takes to construct the product graph is regained by a faster search.

5. RELATED WORK

Several authors have compared and analysed maximal common induced subgraph algorithms. These authors have used algorithms similar to ours, but search for induced subgraphs only.

5.1 Bunke et al.

Bunke et al. [1] performed a benchmark of two algorithms that find maximal common induced subgraphs. One of these is a modified version of the McGregor algorithm for induced subgraphs. The other algorithm reduces the maximal common induced subgraph problem to the clique detection problem and uses clique detection to solve the maximal common induced subgraph problem (like the Koch algorithm). Bunke et al. have used a database containing generated randomly connected graphs.

In the paper Bunke et al. concluded that for graphs with a high edge density it is efficient to build a vertex product graph and use clique detection to solve the problem. This

is the opposite of what we have seen. For cases with a high edge density the McGregor algorithm performs relatively better.

Further research would be needed to find the exact cause of this difference. It should be noted though that the implementations of the algorithms are very different, because they search for induced subgraphs. Another major difference is that we have defined the size of a graph by the number of edges (because a maximum common subgraph is maximal when it has the maximal number of edges), and Bunke et al. have defined the size of a graph by the number of vertices (because an induced subgraph is maximal when the number of vertices is maximal). This difference changes the effect of modifying the edge density. If the number of vertices is kept constant, increasing the edge density means increasing the number of edges. In our study, we kept the number of edges constant, so increasing the edge density means reducing the number of vertices. This difference may be a cause of the difference in results.

5.2 Conte et al.

Conte et al. [2] have researched the performance of three maximal common induced subgraph algorithms. A modified version of the McGregor algorithm, a (relatively simple) clique detection algorithm and a clique detection algorithm with more advanced heuristics. Conte et al. also use a database with artificially generated graph pairs. Conte et al. conclude that the algorithm derived from McGregor is more suitable for regular graphs such as meshes. Conte states that for graphs with a non-regular structure, the efficient response time of the simple clique algorithm makes it repay the time spent to construct the product graph. The clique algorithm with complex heuristics is most efficient for the largest graphs according to Conte et al.

6. CONCLUSION

In this paper we have presented two algorithms which have been thoroughly benchmarked, on graph pairs that we generated using our own graph generators. We created a large database of graph pairs and executed a benchmark using these graph pairs. The database, graph generators and implementation of the algorithms have been made available publicly for future benchmarking activities at http://home.student.utwente.nl/r.h.a.welling/bref.zip

Our results show that the algorithm by Koch, which searches for the maximal clique, performs better in most cases. The time spent constructing an edge product graph is repaid because of the faster search algorithm. The McGregor algorithm performs more efficiently in cases with a high edge density and a high label alphabet size.

Results show that for most cases the Koch algorithm performs better. For that reason Koch is the preferred choice for as a maximal common subgraph algorithm.

For future work, we could extend the database with more graph categories and we could generate larger graphs and graphs with less labels. It would also be interesting to implement more algorithms to compare performance, for example a clique algorithm with more advanced heuristics. It would also be interesting to extend the database with real life graphs, such as protein molecules.

7. REFERENCES

[1] H. Bunke, P. Foggia, C. Guidobaldi, C. Sansone, and M. Vento. A comparison of algorithms for maximum common subgraph on randomly connected graphs. *Structural, Syntactic, and Statistical Pattern Recognition*, pages 85–106, 2002.

[2] D. Conte, P. Foggia, and M. Vento. Challenging complexity of maximum common subgraph detection algorithms: A performance analysis of three algorithms on a wide database of graphs. *Journal of Graph Algorithms and Applications*, 11(1):99–143, 2007.

[3] S. Förtsch and B. Westfechtel. Differencing and merging of software diagrams - state of the art and challenges. In J. Filipe, B. Shishkov, and M. Helfert, editors, *ICSOFT (SE)*, pages 90–99. INSTICC Press, 2007.

[4] R. P. Grimaldi. *Discrete and Combinatorial Mathematics: An Applied Introduction, Fifth Edition*. Addison Wesley, 2003.

[5] I. Koch. Enumerating all connected maximal common subgraphs in two graphs. *Theoretical Computer Science*, 250(1-2):1–30, 2001.

[6] I. Koch, T. Lengauer, and E. Wanke. An algorithm for finding maximal common subtopologies in a set of protein structures. *Journal of Computational Biology*, 3(2):289–306, 1996.

[7] J. McGregor. Backtrack search algorithms and the maximal common subgraph problem. *Software: Practice and Experience*, 12(1):23–34, 1982.

[8] A. Mehra, J. Grundy, and J. Hosking. A generic approach to supporting diagram differencing and merging for collaborative design. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 204–213. ACM, 2005.

[9] B. Naveh. JGraphT. *Internet: http://jgrapht.sourceforge.net*, 2010.

[10] V. Nicholson, C. Tsai, M. Johnson, and M. Naim. A subgraph isomorphism theorem for molecular graphs. *Graph Theory and Topology in Chemistry*, (51):226–230, 1987.

[11] J. Raymond and P. Willett. Maximum common subgraph isomorphism algorithms for the matching of chemical structures. *Journal of Computer-Aided Molecular Design*, 16(7):521–533, 2002.

[12] P. Vismara and B. Valery. Finding Maximum Common Connected Subgraphs Using Clique Detection or Constraint Satisfaction Algorithms. *Modelling, Computation and Optimization in Information Systems and Management Sciences*, pages 358–368, 2008.

APPENDIX

A. RUNNING TIMES

Here we provide a selection of our results in tables as running times in microseconds. In these tables L is the label alphabet size, N is the number of edges in the supergraphs and s is the percentage of edges from the supergraph that is in the maximal common subgraph. Cells are marked with a ' - ' if the parameter combination was not in the database.

Table 3. Koch algorithm results in microseconds for randomly connected graphs with edge density 0.5 and subgraph size 70%.

L \ N	10	20	30
10	521	5 635	707 722
20	388	2 162	11 534
40	355	2 202	7 674
60	352	2 133	7 347

Table 4. McGregor algorithm results in microseconds for randomly connected graphs with edge density 0.5 and subgraph size 70%.

L \ N	10	20	30
10	1 047	5 670	38 792
20	749	2 109	5 083
40	682	1 985	4 485
60	714	1 897	4 514

Table 5. Koch algorithm results in microseconds for three dimensional regular meshes with subgraph size 40%.

L \ N	10	20	30
10	138	620	2 389
20	138	502	1 505
40	137	505	1 447
60	138	490	1 461

Table 6. McGregor algorithm results in microseconds for three dimensional regular meshes with subgraph size 40%.

L \ N	10	20	30
10	1 305	153 526	6 855 645
20	1 096	7 495	295 849
40	884	4 918	20 042
60	871	4 928	20 731

Table 7. Koch algorithm results in microseconds for randomly connected graphs with edge density 0,5 and label alphabet size 10.

s \ N	10	20	30
20%	-	-	580
40%	-	796	6 080
70%	521	5 635	707 722
90%	614	31 528	6 291 900

Table 8. McGregor algorithm results in microseconds for randomly connected graphs with edge density 0,5 and label alphabet size 10.

s \ N	10	20	30
20%	-	-	5 543
40%	-	2 148	11 387
70%	1 047	5 670	38 792
90%	1 076	3 131	8 194

Table 9. Koch algorithm results in microseconds for two dimensional regular meshes with label alphabet size 20.

s \ N	10	20	30
20%	85	167	335
40%	138	503	1 428
70%	353	2 260	7 898
90%	633	4 790	18 420

Table 10. McGregor algorithm results in microseconds for two dimensional regular meshes with label alphabet size 20.

s \ N	10	20	30
20%	611	7 534	215 628
40%	968	7 564	33 037
70%	1 312	7 491	33 248
90%	1 500	10 121	33 758