Three New Approaches for the Maximum Common Edge Subgraph Problem*

James Trimble[1], Ciaran McCreesh[1], and Patrick Prosser[1]

University of Glasgow, Glasgow, Scotland
`j.trimble.1@research.gla.ac.uk`

Abstract. In the maximum common edge subgraph problem, the objective is to find a graph with as many edges as possible that is simultaneously a non-induced subgraph of each of two input graphs. We report our results from adapting a recent algorithm for maximum common *induced* subgraph, using a well-known reduction using line graphs. We also introduce three new direct constraint program encodings for the problem, and describe a new dedicated solver.

Keywords: Maximum common subgraph · Line graph · Constraint programming

1 Introduction

This paper considers the problem of finding a subgraph that appears in each of two input graphs and has as many edges as possible. This problem has numerous applications in biology and chemistry [12], and has also been applied in computer science to a problem of mapping tasks to processors [3]. All of the graphs we consider are undirected, unlabelled, and without loops.

Given two input graphs P and T (for "pattern" and "target"), the *maximum common edge subgraph problem (MCES)* is to find a graph with as many edges as possible that is isomorphic to a subgraph of P and to a subgraph of T simultaneously. These subgraphs are not required to be induced; that is, there can be edges present in either input graph that are not present in the subgraph.

MCES is NP-hard by a simple reduction from HAMILTONIAN CYCLE: the graph in the HAMILTONIAN CYCLE instance becomes the target graph in the MCES instance, and a cycle graph with the same number of vertices becomes the pattern graph. However, a variant of MCES in which the found subgraph is required to be connected can be solved in polynomial time on outerplanar graphs of bounded degree [1].

The *maximum common induced subgraph (MCIS)* problem—which we will take advantage of to solve MCES—is defined similarly, but the objective is to find a subgraph with as many vertices as possible, and the subgraphs must be induced. The two graphs on the left of fig. 1 show an example of an MCES instance; the two graphs on the right show an MCIS instance. In each case, an optimal solution is highlighted.

* Supported by EPSRC.

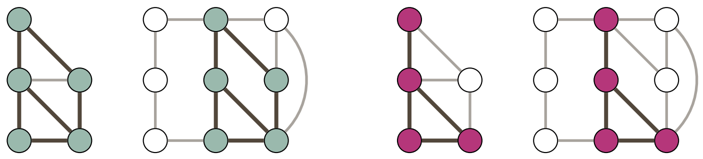

Fig. 1. The two graphs on the left are the pattern and target of a MCES instance, with an optimal solution (6 edges) highlighted. The two graphs on the right are the pattern and target of an MCIS instance, with an optimal solution (4 vertices) highlighted.

In this paper, we give preliminary results comparing three new approaches to solving MCES. The first of these approaches is to use constraint program encodings with an off-the-shelf CP solver. The second uses a modified version of a recent MCIS solver, McSplit, on the line graphs of the two input graphs. The third solves MCES directly, using a data structure and a propagation algorithm similar to those in McSplit. We find that the dedicated algorithms are orders of magnitude faster than our CP models.

Section 2 outlines related prior work. Section 3 describes our three new methods for solving the problem. Section 4 presents experimental results. Section 5 concludes and gives directions for further work.

2 Related work

McGregor [9] gives an early forward-checking algorithm in which decisions are made by mapping a vertex in P to a vertex in T. A matrix—which has a row for each edge in P and a column for each edge in T—keeps track of the remaining feasible edge assignments. The upper bound used is simply the number of rows in the matrix containing at least one non-zero value.

The *line graph* of a graph $G = (V, E)$ is a graph with a vertex for each element of E, such that two vertices are adjacent if and only if their corresponding edges in G share an endpoint. By a result due to Whitney [14], we can find the maximum common edge subgraph of two graphs by searching for a maximum common *induced* subgraph on their line graphs. Several algorithms for MCES, such as RASCAL [11], use this method. This approach has a single pitfall: the triangle graph K_3 and the claw graph $K_{1,3}$ (fig. 2) both have K_3 as their line graph, and it is possible that identical induced subgraphs of the line graphs correspond to subgraphs of the original graph with a triangle-graph connected component of one graph replaced by a claw-graph connected component of the other. Due to the shapes of the graphs, this is known as a ΔY exchange. It is straightforward to check for this occurrence during search, and to backtrack when it is detected. RASCAL does this by comparing the degree sequences of the subgraphs of the original graphs.

Marenco [7] and Bahense et al. [2] formulate MCES as an integer program, and solve it using techniques including branch-and-cut.

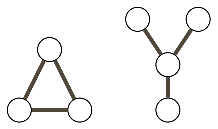

Fig. 2. The graphs K_3 and $K_{1,3}$.

3 Our Methods

In this section, we describe the methods we have used to solve maximum common edge subgraph: a set of three CP models implemented in MiniZinc, and two dedicated solvers inspired by CP techniques.

3.1 CP Models

We have implemented three constraint program models for MCES in the MiniZinc language. In each case, we assume without loss of generality that the pattern graph has no more vertices than the target graph. The first (and simplest) model (fig. 3) has one variable for each vertex in the pattern graph, and one value for each vertex in the target graph. An alldifferent constraint over the set of variables ensures that no target vertex is used twice. The objective function counts the number of adjacent pairs of pattern vertices that are mapped to adjacent target vertices.

```
int: np;                 % order of pattern graph
int: nt;                 % order of target graph

set of int: VP = 1..np;  % pattern vertices
set of int: VT = 1..nt;  % target vertices

array[VP, VP] of int: P; % adjacency matrix of pattern graph
array[VT, VT] of int: T; % adjacency matrix of target graph

array[VP] of var VT: m;  % pattern vertex -> target vertex mappings

var 1..np*(np-1): objval;

constraint objval = sum (v in VP, w in VP where w > v /\ P[v, w]==1)
        (T[m[v], m[w]]);

constraint alldifferent(m);

solve :: int_search(m, first_fail, indomain_split, complete)
        maximize objval;
```

Fig. 3. The first MiniZinc model

A second MiniZinc model could construct a mapping from edges to edges. However, doing so would require a large number of disjunctive constraints to enforce incidence. We solve this problem by instead mapping pattern edges to *oriented* target edges, having two values per target edge in each variable. We also have an additional value $\perp$ signifying that the pattern edge is not used. Constraints ensure that if two pattern edges are assigned to target edges, then they have a shared endpoint if and only if their assigned target edges share the corresponding endpoint. An alldifferent-except-$\perp$ constraint ensures that each edge value is used only once. Finally, a set of constraints ensures that a single target edge may not be used in both orientations. The objective value is the count of variables that take non-$\perp$ values. This model is intricate, and we do not list it here for space reasons.

Our third model (fig. 4) includes the vertex variables and alldifferent constraint of model 1, and the edge variables and objective function of model 2. A set of constraints ensures that an edge variable takes a particular edge value if and only if the vertex variables corresponding to the pattern edge's endpoints take the values corresponding to the target edge's endpoints.

```
int: np;                 % order of pattern graph
int: nt;                 % order of target graph

int: mp;                 % size of pattern graph
int: mt;                 % size of target graph * 2
                         %   (i.e. number of oriented edges)

set of int: VP = 1..np;  % pattern vertices
set of int: VT = 1..nt;  % target vertices

array[VP, VP] of int: P; % adjacency matrix of pattern graph
array[VT, VT] of int: T; % adjacency matrix of target graph

array[1..mp, 1..2] of int: PE; % edge list of pattern graph
array[1..mt, 1..2] of int: TE; % oriented edge list of target graph

array[VP] of var VT: m;  % pattern vertex -> target vertex mappings
array[1..mp] of var 0..mt: m_edge;  % pattern edge index -> target
                                    %   edge index mappings. 0 means _|_

var 1..mp: objval;

constraint forall (i in 1..mp, j in 1..mt)
        (m_edge[i]==j <-> (m[PE[i,1]]==TE[j,1] /\ m[PE[i,2]]==TE[j,2]));
constraint objval = sum (a in m_edge) (a != 0);
constraint alldifferent(m);

solve :: int_search(m_edge, first_fail, indomain_split, complete)
        maximize objval;
```

Fig. 4. The third MiniZinc model

3.2 McSplit

Our first dedicated solver is based on McSplit, a recent algorithm for the MCIS problem [8]. McSplit is a forward-checking algorithm, with a variable for each vertex in the pattern graph and a value for each vertex in the target graph, along with a dummy value $\bot$ representing that the pattern vertex is not used. During search, any two domains are either disjoint (if we ignore $\bot$) or identical; this special structure of the problem allows domains to be stored in a compact data structure in which variables with identical domains share a single representation of their domain in memory. McSplit implements the soft all-different bound [10] in linear time by exploiting this special structure to avoid having to perform a matching, and uses variable-ordering heuristics inspired by the dual viewpoint [5].

We do not call McSplit directly on the pattern and target graphs, but rather on their line graphs, using the method described in section 2. To detect ΔY exchanges, we maintain a counter for each vertex in the *original* pattern and target graphs, which records how many incident edges (represented by vertices in the line graphs) are currently being used. If the number of non-zero counters for the pattern graph differs from the number of non-zero counters for the target graph, a ΔY exchange has occurred and it is necessary to backtrack.

3.3 SplitP

Our second dedicated solver, SplitP, shares the compact data structures and the forward-checking of McSplit, and uses a similar partitioning algorithm to filter domains. However, SplitP does not use line graphs, but rather models the problem directly in style of our MiniZinc models 2 and 3, with variables for pattern edges and values for *oriented* target edges.

We use the example graphs in fig. 5 to illustrate SplitP's data structures.

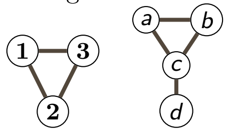

Fig. 5. Example graphs P and T

Initially, each pattern edge (which we denote by its endpoints; for example $(1,2)$) has all four target edges in its domain. Rather than storing the four domains separately, we store the same information using two arrays of edges. In addition, we store the integer 2 to signify that either orientation of a target edge may be chosen (for example, edge $(1,2)$ may be mapped to either (a,b) or (b,a)). Thus, before any tentative edge assignments are made, the domains are stored as:

$$[(1,2),(1,3),(2,3)] \qquad [(a,b),(a,c),(b,c),(c,d)] \qquad\qquad 2$$

Now, suppose that the algorithm has made the tentative assignment of pattern edge $(1,2)$ to target edge (a,c). Edge $(1,3)$ has (a,b) and $\bot$ in its domain, while edge $(2,3)$ has (c,b), (c,d) and $\perp$ in its domain. These domains are stored as follows (with the 1 at the end of each line signifying that only the shown orientation of each target edge is permitted).

$[(1,3)]$	$[(a,b)]$	1
$[(2,3)]$	$[(c,b),(c,d)]$	1

We can view the McSplit approach (where values correspond to vertices in the target line graph, and thus effectively correspond to unoriented edges in the original target graph) as delaying the decision of which way to orient the target edges until after finding an optimal solution. SplitP, by contrast, makes orientation decisions as early as possible.

3.4 "Down" variants

The branch-and-bound solvers McSplit and SplitP attempt to find increasingly large incumbent subgraphs. In addition, we have implemented variants of these two algorithms that take the opposite approach, solving a sequence of decision problems where we first ask whether we can find a subgraph that uses all of the edges in the smaller graph, then all edges but one, and so on. These are named McSplit↓ and SplitP↓. (A version of McSplit↓ for MCIS was introduced in [8]; this used the approach of Hoffmann et al. [6].)

4 Experiments

For our experiments, we used a database of randomly-generated pairs of graph [13, 4]. To keep the total run time manageable, we selected the first ten instances from each family with no more than 35 vertices, giving a total of 2750 instances.

We used a cluster of machines with Intel Xeon E5-2697A v4 CPUs. For the CP models, MiniZinc 2.1.7 and the Gecode solver were used. We used the `first_fail` and `indomain_split` MiniZinc search annotations for these models.

Our experiments cover only the new approaches introduced in this paper, and do not provide a comparison with existing state-of-the-art algorithms. We intend to provide a fuller comparison—using more solvers and more families of instances—in a future version of this paper.

Figure 6 shows a cumulative plot of run times. Each point on a curve indicates the number of instances that were, individually, solved in less than a time shown on the horizontal axis. For example, approximately 200 of the instances could be solved by the MiniZinc 1 model in 1000 ms or less per instance.

The dedicated solvers are more three orders of magnitude faster than the fastest MiniZinc model (model 3). The ↓ variants are slightly faster than the branch-and-bound variants, and each version of SplitP is around twice as fast as its corresponding McSplit program.

The first plot of Figure 7 is a scatter plot of run times for SplitP↓ and McSplit↓, with one point per instance. The second plot shows search node counts.

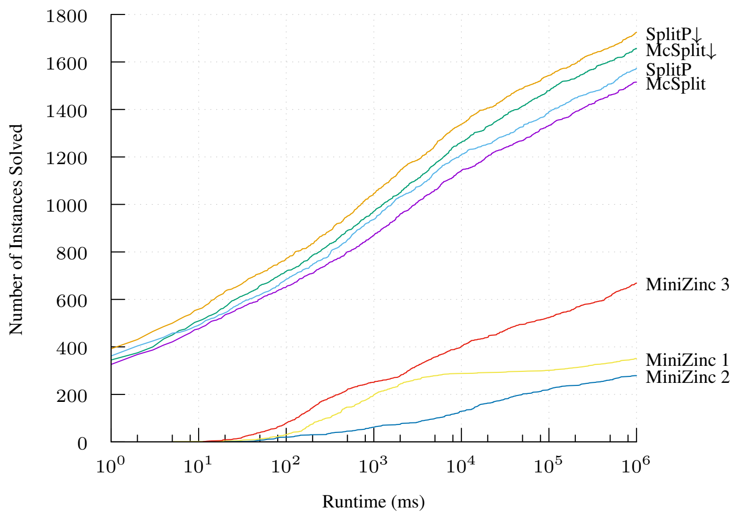

Fig. 6. Cumulative plot of run times

Instances where either algorithm timed out are not shown on the second plot. On most instances, McSplit↓ takes slightly more time and search nodes than SplitP↓. This difference is perhaps down to small differences in the choices made for variable and value ordering heuristics. While both McSplit↓ and SplitP↓ use a smallest-domain-first variable ordering, McSplit↓ breaks ties on degree in the linegraph, while SplitP↓ uses dynamic heuristics. We plan to carry out further investigations into tie-breaking rules in the future.

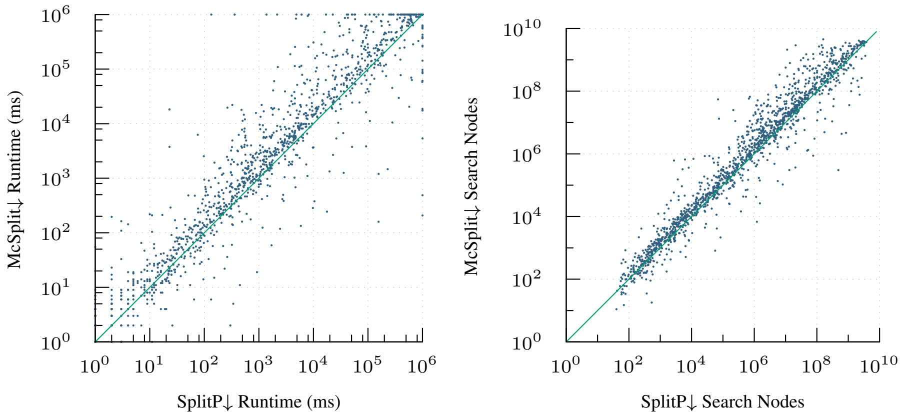

Fig. 7. Scatter plots of run times and search nodes, SplitP↓ versus McSplit↓. Each point represents one instance.

Figure 8 shows run times and search nodes for SplitP and SplitP↓. SplitP↓ is seldom more than three times slower than the branch-and-bound variant, and is often orders of magnitude faster.

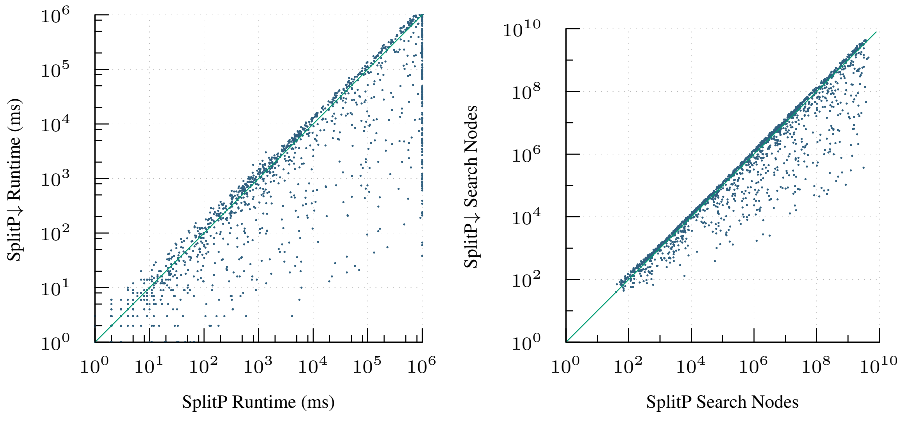

Fig. 8. Scatter plots of run times and search nodes, SplitP versus SplitP↓. Each point represents one instance.

5 Conclusion

We have implemented three constraint program models for maximum common edge subgraph, and two dedicated solvers inspired by constraint programming techniques. Of the CP models, we found the most effective to be a model containing variables for both vertices and edges. The dedicated solvers were orders of magnitude faster than the CP models; of these, the SplitP↓ and McSplit↓ variants were faster than their branch-and-bound counterparts. SplitP was somewhat faster than McSplit, but this may be simply due to the choices of variable and value ordering heuristics.

Although the CP models ran much more slowly than the dedicated algorithms, they have the advantage of flexibility: it would be straightforward to add additional constraints to the model if required. By contrast, it is unlikely that additional constraints could easily be used with the compact data structures of McSplit and SplitP.

In the future, we plan to extend our programs to handle labels on vertices and edges, and directed edges. We will also investigate a weighted version of the problem, where each mapping between an edge in the pattern graph and an edge in the target graph has an associated weight. We believe that a reduction to maximum weight clique using an association graph encoding could be used to solve this problem.

We also intend to run a fuller set of experiments, comparing against existing state-of-the-art algorithms and studying the effect of heuristics on solver performance.

References

1. Akutsu, T., Tamura, T.: A polynomial-time algorithm for computing the maximum common connected edge subgraph of outerplanar graphs of bounded degree. Algorithms **6**(1), 119–135 (2013). https://doi.org/10.3390/a6010119, https://doi.org/10.3390/a6010119
2. Bahiense, L., Manic, G., Piva, B., de Souza, C.C.: The maximum common edge subgraph problem: A polyhedral investigation. Discrete Applied Mathematics **160**(18), 2523–2541 (2012). https://doi.org/10.1016/j.dam.2012.01.026, https://doi.org/10.1016/j.dam.2012.01.026
3. Bokhari, S.H.: On the mapping problem. IEEE Trans. Computers **30**(3), 207–214 (1981). https://doi.org/10.1109/TC.1981.1675756, https://doi.org/10.1109/TC.1981.1675756
4. Conte, D., Foggia, P., Vento, M.: Challenging complexity of maximum common subgraph detection algorithms: A performance analysis of three algorithms on a wide database of graphs. J. Graph Algorithms Appl. **11**(1), 99–143 (2007), http://jgaa.info/accepted/2007/ConteFoggiaVento2007.11.1.pdf
5. Geelen, P.A.: Dual viewpoint heuristics for binary constraint satisfaction problems. In: ECAI. pp. 31–35 (1992)
6. Hoffmann, R., McCreesh, C., Reilly, C.: Between subgraph isomorphism and maximum common subgraph. In: Singh, S.P., Markovitch, S. (eds.) Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA. pp. 3907–3914. AAAI Press (2017), http://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14948
7. Marenco, J.: Un algoritmo branch-and-cut para el problema de mapping. Ph.D. thesis, Masters thesis, Universidad de Buenos Aires, 1999. (1999)
8. McCreesh, C., Prosser, P., Trimble, J.: A partitioning algorithm for maximum common subgraph problems. In: Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017. pp. 712–719 (2017). https://doi.org/10.24963/ijcai.2017/99, https://doi.org/10.24963/ijcai.2017/99
9. McGregor, J.J.: Backtrack search algorithms and the maximal common subgraph problem. Softw., Pract. Exper. **12**(1), 23–34 (1982). https://doi.org/10.1002/spe.4380120103, https://doi.org/10.1002/spe.4380120103
10. Petit, T., Régin, J., Bessière, C.: Specific filtering algorithms for over-constrained problems. In: Walsh, T. (ed.) Principles and Practice of Constraint Programming - CP 2001, 7th International Conference, CP 2001, Paphos, Cyprus, November 26 - December 1, 2001, Proceedings. Lecture Notes in Computer Science, vol. 2239, pp. 451–463. Springer (2001). https://doi.org/10.1007/3-540-45578-7_31, https://doi.org/10.1007/3-540-45578-7_31
11. Raymond, J.W., Gardiner, E.J., Willett, P.: RASCAL: calculation of graph similarity using maximum common edge subgraphs. Comput. J. **45**(6), 631–644 (2002). https://doi.org/10.1093/comjnl/45.6.631, https://doi.org/10.1093/comjnl/45.6.631
12. Raymond, J.W., Willett, P.: Maximum common subgraph isomorphism algorithms for the matching of chemical structures. Journal of Computer-Aided Molecular Design **16**(7), 521–533 (2002). https://doi.org/10.1023/A:1021271615909, http://dx.doi.org/10.1023/A:1021271615909
13. Santo, M.D., Foggia, P., Sansone, C., Vento, M.: A large database of graphs and its use for benchmarking graph isomorphism algorithms. Pattern Recognition Letters **24**(8), 1067–1079 (2003). https://doi.org/10.1016/S0167-8655(02)00253-2, http://dx.doi.org/10.1016/S0167-8655(02)00253-2

14. Whitney, H.: Congruent graphs and the connectivity of graphs. American Journal of Mathematics **54**(1), 150–168 (1932)