Merge Search: a Matheuristic Framework for Combinatorial Optimisation Problems

Angus Kenny[a], Dhananjay Thiruvady[b], Davaatseren Baatar[c], Andreas T. Ernst[c], Xiaodong Li[d], Mohan Krishnamoorthy[e], Gaurav Singh[f]

[a]*School of Engineering and Information Technology, University of New South Wales, Canberra ACT, Australia*
[b]*School of Information Technology, Deakin University, Geelong VIC, Australia*
[c]*School of Mathematics, Monash University, Clayton VIC, Australia*
[d]*School of Computing Technologies, RMIT University, Melbourne VIC, Australia*
[e]*School of Information Technology and Electrical Engineering, The University of Queensland, St Lucia QLD, Australia*
[f]*BHP, Perth WA, Australia*

Abstract

Exact methods are useful when solving combinatorial optimisation problems, although their time complexity is often high. Conversely, meta-heuristics are adept at finding feasible solutions in reasonable time, but their stochastic nature can make it hard to refine solutions effectively. To this end, a class of algorithms known as matheuristics was developed, which harness the strengths of both these methods.

This paper presents a matheuristic called merge search, an iterative decomposition algorithm which creates a reduced sub-problem by grouping variables that share common values across a population of solutions. Merge search can be thought of as a generalisation of crossover, from evolutionary algorithms; instead of combining two solutions to create a single offspring solution, the subspace that is spanned by the entire population is searched for the optimal way to combine all solutions.

Merge search is applied to two very different NP-Hard problems: the constrained pit limit (CPIT) problem, from the field of open-pit mining; and the well-studied Steiner tree problem in graphs (STPG). The CPIT results are used to show the effectiveness of merge search, improving the best-known bounds in two of the six instances tested; while the STPG results demonstrate the versatility of this method.

Keywords: `elsarticle.cls`, LaTeX, Elsevier, template
2010 MSC: 00-01, 99-00

1. Introduction

Combinatorial optimisation problems, at their very basis, involve searching a finite (but often *very* large) and discrete search space to find an optimal

object (Papadimitrou & Steiglitz, 1982). Optimisation problems modelling situations in the real-world can be very messy and complex, often involving many variables and constraints (Deb, 2012). Traditional mathematical solvers are not well-suited to these messier, large-scale problems; making finding ways to decompose complex problems into smaller, more manageable sub-problems an important topic of study (Blum et al., 2011).

There are two large factors that can contribute to how difficult a problem is to solve (Wolsey & Nemhauser, 1999). The first being the number of decision variables that need to be considered. The time complexity of many exact solvers is exponential (Boyd & Vandenberghe, 2004), meaning that a problem with many decision variables can quickly become intractable. The second factor (although there are many others) is the interaction between decision variables. If a problem is completely separable, it can be solved by completely decomposing it into its individual variables and optimising them separately. The other extreme to this is when it is completely inseparable, meaning the entire problem must be solved at once. Most problems tend to fall somewhere between these two ends of the spectrum, making a search for an effective way to ascertain these interactions an important area of research (Dantzig & Wolfe, 1960; Omidvar et al., 2013).

The method proposed in this paper is a *divide-and-conquer* (Levitin, 2012) algorithm that addresses both these factors. Using information from across a locally generated population of solutions, the set of decision variables is partitioned with each partition being treated as a single variable. This reduced sub-problem, can then be solved using some exact method (e.g., MIP solver), as the number of decision variables has been significantly decreased. By partitioning the variables in this way, relationships between decision variables are preserved, so the groupings are more meaningful than if they were generated arbitrarily. The granularity of solutions produced by merge search can be controlled by randomly splitting the partitions. Without this mechanism, it is not guaranteed that all possible solutions can be produced.

The effectiveness and versatility of merge search is demonstrated in this paper by applying it to two very different combinatorial optimisation problems from the literature. The first is the constrained pit limit (CPIT) problem from the field of open-pit mining; and the second is the well-studied Steiner tree problem in graphs (STPG). By applying it to the CPIT problem, merge search is shown to be competitive with current state-of-the-art techniques, improving the best-known bounds on two of the six problems conisdered from the *minelib* (Espinoza et al., 2012) dataset. The intensification aspects of the algorithm are also investigated in these experiments, by analysing the effect that splitting the partitions has on solution quality.

Being a less complicated problem, the STPG is used to explore the diversification aspects of merge search, along with some properties of its individual components. Experiments are carried out to examine the effect that different population sizes have on the operation of the algorithm, along with comparing the performance of the full merge search against each of its two main constituent parts (local search and a MIP solver). As the STPG is such a well-studied prob-

lem, improving any known bounds for the *steinlib* (Koch et al., 2001) dataset would require very sophisticated formulations and pre-processing techniques — neither of which are the focus of this research — therefore, the experiments are used primarily to investigate the properties of merge search, and demonstrate its versatility. **(XL: this paragraph on STPG can be much reduced to just 1 or 2 sentences, as the emphasis is on CPIT)**

As well as investigating the different properties of the algorithm, by applying it to these significantly different problems, it can be demonstrated that merge search is generalisable across many problems from different domains. The results from both sets of experiments are compared against state-of-the-art published results and also a baseline of a custom greedy randomised adaptive search procedure (GRASP) (Feo & Resende, 1995) algorithm, developed for each problem. **(XL: I think this paragraph can be shortened and merged into the previous one, as it still says about "generalization across different problems". Isn't it the same message already mentioned?)**

(XL: what you could add here perhaps a list of explicit stated novel contributions, and how this current work differs from our previous GECCO work and other similar work on merge search). This paper is organised as follows. Section 2 highlights some related work in the field of matheuristics and solution merging techniques; it also introduces the two problems, CPIT and STPG, and summarises some of the literature surrounding them. The merge search algorithm itself is discussed as a general framework for solving combinatorial optimisation problems in Section 3. A description of how merge search is applied to the specified problems is provided in Section 4. Section 5 details the experimental setup and the datasets used; with results of these experiments being discussed and analysed in Section 6. Finally, Section **??** concludes the paper and outlines some possible future research.

2. Background

This section gives some background information on mathueristics and solution merging techniques. It also introduces the two problems used in this paper to demonstrate the effectiveness and versatility of merge search and provides a summary of the literature surrounding them.

2.1. Matheuristics and solution merging techniques

One of the most active areas of research in the area of hybrid meta-heuristics is the combination of integer and linear programming (ILP) techniques with meta-heuristics, known as matheuristics (Boschetti et al., 2009).

Ultimately, the purpose of creating a hybrid meta-heuristic from two different algorithms is to harness the characteristics of both types of algorithms and use the strengths of one to ameliorate the weaknesses of the other. Meta-heuristics are useful for conducting a global search for decent quality, feasible solutions reasonably quickly, as they employ stochastic techniques to sample large areas of the search space. However, they are not as adept at using a fine-grained search

to improve on those decent quality solutions. If the fitness landscape is multi-modal (i.e., it has many peaks), some meta-heuristics are also good at identifying multiple areas of the search space that might be of interest; this information can then be used to decompose the problem into smaller sub-problems.

Conversely, exact methods such as integer programming are typically very bad at global search for very large problem sizes and, sometimes, even finding a feasible solution to a problem can take a lot of computing resources. One aspect that exact methods are very good at though, is fine-grained search and also providing a guarantee of optimality. If the exact method is being used to solve a restricted sub-problem that was produced by a meta-heuristic, it is important to note that this "guarantee of optimality" is only a guarantee of *local* optimality, as the stochastic methods used by the meta-heuristic cannot guarantee that this was the globally optimal restricted sub-problem to produce.

A relatively new direction of research in this space is in combining information from a large number of solutions by a process called *merging*, to produce a restricted sub-problem which can then be solved using an exact solver, or by some other method.

An early approach to this technique that still used sampling, but in a more systematic and intelligent way than GA crossover, is the *path-relinking* algorithm proposed by Glover et al. (2000), which generalises an earlier concept called *scatter search* (Glover et al., 2003). One of the first instances of searching a restricted sub-space produced by merging a population of generated solutions is the application of the technique to the travelling salesman problem (TSP) by Applegate et al. (1998). An example of an iterative solution merging algorithm that incorporates elements of problem decomposition by partitioning variables is *kernel search*, developed originally to solve the problem of security portfolio optimisation (Angelelli et al., 2012). Kernel search has been adapted to a number of other problems, such as the multi-dimensional knapsack problem (Angelelli et al., 2010), with some success; however, the nature of this system of buckets means that it is better suited to problems where there is not a high degree of dependence between variables.

More recently, the construct, merge solve and adapt (CMSA) algorithm by Blum et al. (2016) employs similar strategies in order to narrow-down large search spaces. Starting with an empty sub-instance $\mathcal{C}'$, solutions are probabilistically generated, from scratch, and their components added to $\mathcal{C}'$. This reduced sub-instance is then solved using an exact solver and an ageing mechanism is used to remove components from $\mathcal{C}'$ that have not been useful in the preceding iterations.

Although relatively nascent, the field of hybrid meta-heuristics research is broad, with many new techniques being developed all the time. For further information on general hybrid meta-heursticis, the reader is directed to the survey papers by Talbi (2002); Blum et al. (2011); Raidl (2015) or the books by Maniezzo et al. (2009); Blum et al. (2008); and for information specifically on matheuristics, see Boschetti et al. (2009); Fischetti & Fischetti (2018); Ribeiro et al. (2020).

2.2. Open-pit mining problems

Open-pit mining is a very important industry in Australia and around the world Singh et al. (2012). Two of the most critical tasks within the life-cycle of an open-pit mine is planning and production scheduling. These tasks allow the mine operator to estimate the total value of the mine over its life and also to identify areas for excavation that will yield the most value. Proper planning of a mine ensures maximum profit for the operator and, because this is typically talked about in the hundreds of millions of dollars, it is an excellent application for optimisation techniques as very small changes in efficiency can still translate to significant sums of money.

In order to model something so complex as a combinatorial optimisation problem, the earth to be mined (known as the *orebody*) is typically discretised into a three-dimensional array of *blocks* with each assigned a value based on the ore content and the cost required to excavate it. These values are calculated by taking core samples and using geological and statistical methods to estimate the value of each block. The aim is to maximise the net present value (NPV) of the mine by determining the set of blocks to extract and the order in which to extract them (Meagher et al., 2014).

Problems in mine planning and production scheduling are very large and tend to have few side-constraints (often well under a hundred), but many blocks and many, many more precedence constraints governing when blocks can be mined. This means traditional mathematical solvers are unable to solve these problems without first using some form of decomposition, making these problems perfect candidates for hybrid meta-heuristics, despite there being very little in the literature.

Due to the sensitive nature of information surrounding mining enterprises, obtaining problem data for academic research can be challenging, however *minelib* (Espinoza et al., 2012) provides a repository of problems and results that are freely available to the general public. These sets contain data for versions of the problem such as the ultimate pit limit (UPIT), constrained pit-limit and the precedence constrained production scheduling problem (PCPSP). It is the CPIT problem that will be the focus of these experiments. A mathematical formulation of this problem can be found in this paper and is included in Appendix Appendix A (Equation A.1).

Despite being formulated as long-ago as the 1960s, there is very little in the literature about using meta-heuristic techniques to solve open-pit mining problems. Chicoisne et al. (2012) solve the LP relaxation of the CPIT problem and then use a topological sorting heuristic to find a feasible, integral solution from the fractional LP solution, which is then improved by local search. In an attempt to reduce the computation required by the large number of variables and constraints present in the CPIT problem, Jélvez et al. (2016) use a heuristic technique to aggregate blocks into larger groups and then solve this simple model. The solution to this simple model is then used to determine groups of blocks that can be disaggregated to obtain finer-grained solutions. Bley et al. strengthen the MIP formulation for open-pit mining problems by adding extra

inequality constraints. These constraints are derived by combining precedence and production constraints. They did not test their algorithm on the *minelib* instances, likely because of their size, however they report that the extra constraints present in their model serve to effectively reduce the computation time needed to solve their custom problem instances using CPLEX. For a good survey of research in these related areas, see Hochbaum & Chen (2000); Meagher et al. (2014). The *minelib* results used for comparison in this paper are taken from the masters thesis of Muñoz Martínez (2012). Muñoz uses a modified version of the Bienstock and Zuckerberg algorithm (Bienstock & Zuckerberg, 2010) to solve the LP relaxation, then uses a topological sorting algorithm, similar to Chicoisine et al. to produce a feasible solution.

(AK: ADD RECENT PAPERS AND OUR GECCO PAPERS) (XL: what is missing here is the information on why such a large scale combinatorial problem is so challenging, and conventional techniques cannot be applied directly. We want to make clear problem reduction is crucial step, and this is especially true for this CPIT problem)

2.3. Steiner tree problem in graphs (STPG)

The Steiner tree problem in graphs (STPG) is a classic problem in the field of combinatorial optimisation, the decision version being one of the original 21 NP-Complete problems outlined by Karp (1972) in his seminal paper.

Given an undirected, weighted graph $G = (V, E, c)$ where V is the set of vertices in G, $T \subseteq V$ is a special subset of V called the *terminal vertices*, E is the set of edges in G and $c : E \to \mathbb{Z}^+$ is a function that maps each edge to some positive, integer weight. The STPG aims to find a set of vertices $V' \subseteq V$ such that $T \subseteq V'$ and V' induces a spanning tree over T in G of minimum total weight. Vertices that are in the set $V' \setminus T$ are called *Steiner vertices*.

Exact methods for solving the STPG have been developed using techniques such as integer linear programming (ILP), lagrangian relaxation and primal-dual strategies (Polzin & Vahdati, 2000); however these approaches suffer from exponential worst-case computation times which can make some large-scale instances intractable. The current state-of-the-art exact approaches to solving the STPG are hybrid (Polzin, 2003; Vahdati Daneshmand, 2004); several algorithmic, graph reduction, metaheuristic and mathematical programming techniques, working together to produce provably optimal solutions in a much faster time than traditional optimisation techniques alone.

The experiments performed in this study were evaluated by comparing merge search to the published results of two current metaheuristic algorithms for solving the multicast routing problem and its underlying mathematical structure, the STPG; the *JPSOMR* algorithm developed by Qu et al. (2013) and the more famous *GRASP* algorithm, adapted for multicast routing and the STPG by Skorin-Kapov & Kos (2006).

The *jumping particle swarm optimisation for multicast routing* (JPSOMR) algorithm is based on a variant of the more common particle swarm optimisation algorithm (PSO) called *jumping particle swarm optimisation* (JPSO). As PSO

was developed to solve continuous optimisation problems, a different technique was needed in order to solve discrete and combinatorial problems; one of those such methods is called JPSO. For further information on the PSO, JPSO or JPOSMR algorithms, see Kennedy (2010); Consoli et al. (2010) and Qu et al. (2013).

The GRASP algorithm was adapted for the multicast routing problem and the STPG by Skorin-Kapov & Kos (2006). It constructs good quality initial solutions by using Dijkstra's algorithm (Dijkstra, 1959). The local search phase incorporates tabu search, with some modifications, applied iteratively until no better solution can be found or a certain number of iterations are performed with no improvement. More information on the GRASP algorithm and its application to the STPG can be found in Feo & Resende (1995); Skorin-Kapov & Kos (2006) and Martins (1999).

The STPG has applications ranging widely from integrated circuit design (Cho, 2001), to distribution and logistics network design (Eisenbrand et al., 2010), to computing phylogenetic trees in biology (Foulds & Graham, 1982) — and it has been of interest to mathematicians since before the advent of digital computers. For more information on the history of the STPG and methods of solving it, the reader is directed to Brazil et al. (2014); Hwang & Richards (1992); Prömel & Steger (2012); Du et al. (2013).

(XL: this section on STPG can be substantially reduced)

3. Merge Search

This section provides a description merge search as a general matheuristic framework for solving combinatorial optimisation problems with binary decision variables. It gives the pseudocode and details of the steps involved, before discussing some of its properties and giving a comparison with the CMSA heuristic mentioned in Section 2.

3.1. Description

The pseudocode for merge search is outlined in Algorithm 1 and comprises the following aspects.

Finding an initial solution

In order to produce a population of solutions, an initial, feasible solution x^0 must first be found. Some problems are so large and complex that even producing a feasible solution is quite computationally expensive — let alone one that is of guaranteed good quality. Although there are cases where finding a feasible solution is reasonably easy; in general, finding a feasible solution to a combinatorial problem is as hard as finding an optimal one (Papadimitrou & Steiglitz, 1982).

When considering merge search as a general framework for solving constrained optimisation problems, the ideal circumstance would be when there already exists a custom heuristic for constructing a solution to the problem.

Algorithm 1 Merge Search Matheuristic

Require: A combinatorial optimisation problem with variables x specified as $\max f(x) : x \in \mathcal{F}$ (where $\mathcal{F}$ is the feasible set)
Require: Initial solution $x^0 \in \mathcal{F}$
1: **for** $k = 1, 2, \ldots$ **do**
2: **for** $j = 1, 2, \ldots, m$ **do**
3: Let s^j be a neighbouring solution to x^{k-1}
4: **end for**
5: Let $S = \{s^1, \ldots, s^m\} \cup \{x^{k-1}\}$ be all such solutions
6: Let $\mathcal{P} = \{P_1, \ldots, P_p\}$ be a partition of the variables into sets for which all solutions are constant:

$$\bigcup_{P \in \mathcal{P}} P = \{1, \ldots, n\}, \quad P \cap Q = \emptyset \; \forall \; P \neq Q \in \mathcal{P},$$
$$\text{and } s_i = s_j, \quad \forall \; s \in S, \; P \in \mathcal{P}, \; i, j \in P$$

7: **if** $|\mathcal{P}| < K$ **then** split subsets until $|\mathcal{P}| = K$
8: Solve $x^k = \arg\max f(x) : x \in \mathcal{F}, \; x_i = x_j, \; \forall i, j \in P \in \mathcal{P}$
9: **end for**

However, if no such heuristic exists, then because it can be shown (Lemma 2) that the random splitting aspect of merge search makes it theoretically capable of producing any solution in the search space, it is possible to start with a completely random (feasible) solution and still find the optimal solution — if given enough time. Of course, in practice, "enough time" can be infeasibly long for large search spaces, so a more intelligent method of producing an initial solution is preferable.

Neighbourhood search (Step 3)

Many solution merging meta-heuristics such as the construct, merge, solve and adapt (CMSA) heuristic (Blum et al., 2016) require a method that constructs solutions from scratch in order to produce a population to merge. However, for some large and complex problems, producing a feasible solution from scratch can be very computationally expensive. Because it is generally easier to produce a new solution from an existing one, than it is to start from scratch, merge search generates its population by defining a local-search operator for the problem and sampling the neighbourhood of a given solution.

The local-search neighbourhood can be as simple, or as sophisticated, as is required. It can be a custom-built, problem specific, heuristic that always produces feasible solutions; or it can be a heuristic that simply generates random bit-strings. So long as there is at least one feasible solution in the population, it can be shown (Lemma 1) that the merge operation will always produce a feasible solution and the random splitting heuristic ensures that any possible solution in the search space can be produced.

When considering merge search as a general meta-heuristic framework, it

does not matter how the population is produced; some methods will produce populations that lead to more efficient searches, and some methods will produce populations that require a very long time to find the optimal solution. As with most hybrid meta-heuristic search techniques, it becomes about finding a good balance between how much computational effort is spent on exploration through, ensuring diversity of the population, and how much is spent on exploitation, through focusing on one particular area of the search space.

Once a population has been generated, it can be used to define a partition on the decision variables.

Defining the partition (Step 6)

The simplest way to partition the decision variables for a problem with a population of solutions $S = \{s_1, \ldots, s_m\} \cup \{x^{k-1}\}$ is to divide them into three groups: variables that *always* take the value 0 across all solutions; variables that *always* take the value 1 across all solutions; and variables that take *either* 0 or 1 across all solutions. This partition can be used to produce a reduced sub-problem with the variables that are in the first group fixed to 0; the variables that are in the second group fixed to 1; and the variables that are in the third group allowed to take either.

In this sense, merge search can be thought of as a generalisation of an optimised, multi-parent uniform crossover operator similar to that used in the genetic algorithm (GA). In the uniform crossover operator for GA (Luke, 2009), each decision variable is considered in turn and its value selected from one of two parents with some probability. Because the value for every decision variable must be taken from either parent, no matter how many offspring are produced from these two parents the set of decision variables that are 1 for both parents will *always* take the value 1, and the set of variables that are 0 for both parents will *always* take the value 0. Effectively, these variables have been fixed and the only variables that are free to take either 0 or 1 are those that are not in either of these sets. This is exactly equivalent to performing a simple merge operation, without grouping, on a population consisting of two solutions; but where uniform crossover merely randomly samples the sub-space of solutions produced by the two parents, the merge operation searches that sub-space for the (locally) optimal offspring. Uniform crossover is also *restricted* to a population of size two, whereas merge search generalises this idea to any arbitrary population size.

Using this naïve method of partitioning, with a very diverse population, when considering each variable in the 0/1 region individually can result in very little reduction in the size of the sub-problem produced. For this reason, a more sophisticated way of partitioning the decision variables is defined.

Definition 1. *Let x be the set of decision variables for a given problem, then the set $s_i \subseteq x$ is the set of decision variables that take the value 1 in a given solution i and $S = \{s_1, \ldots, s_m\}$ is a population of m solutions. Now, a* **merge partition** *is the set $\mathcal{P} = \{P \mid \bigcap_{i=1}^{m} s_i^{b_i},\ b \in \mathbb{Z}_2^m\}$, where $s_i^1 = s_i$ and $s_i^0 = s \setminus s_i$.*

Figure 1 illustrates all possible partitions that can be induced by a population of three solutions. The shaded circles indicate all of the variable assignments

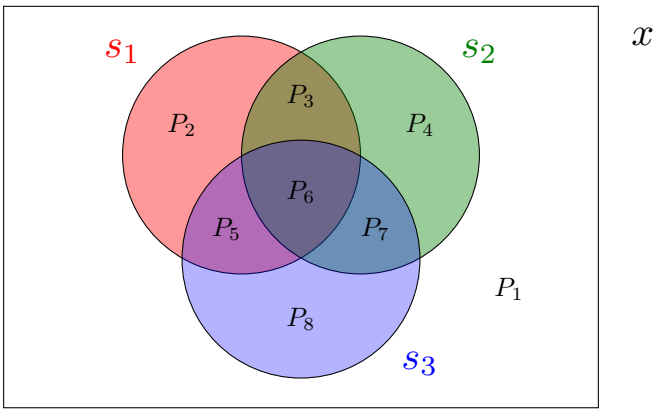


Figure 1: Partitions formed by intersecting solutions.

that are included in a particular solution. Here, partition $P_1 = s_1^0 \cap s_2^0 \cap s_3^0$, which corresponds to the set of decision variables that take the value 0 across all solutions; partition $P_6 = s_1^1 \cap s_2^1 \cap s_3^1$, which corresponds to the set of decision variables that take the value 1 across all solutions; and the other partitions correspond to sets of decision variables that, while they do not take the same value across all solutions, *agree amongst themselves* across all solutions. For example, all decision variables in partition P_5 take a 0 in s_2 and a 1 in s_1 and s_3.

By partitioning the decision variables in this manner, it is reasonable to assume that — given a big enough population size — were a new solution to be generated, the majority of the decision variables in the new solution would agree with the other variables in their respective partitions. Therefore, the decision variables can be aggregated into groups in the reduced sub-problem, with each partition being considered as an individual variable.

The theoretical maximum number of partitions (and therefore, decision variables in the reduced sub-problem) possible in a merge population of m solutions is 2^m. This theoretical maximum is only achieved when,

$$\forall (s_i, s_j) \in S^2,\ s_i^b \cap s_j^c \neq \emptyset, \forall b, c \in \mathbb{Z}_2;$$

however, typically not all solutions in a population will interact with all other solutions (i.e., there exist some pairs of solutions $(s_i, s_j) \in S^2$ such that $s_i^b \cap s_j^c \neq \emptyset, \forall b, c \in \mathbb{Z}_2$), so $|\mathcal{P}| << 2^{|S|}$, in practice.

Random splitting (Step 7)

When generating a population by sampling the neighbourhood around an initial solution, the size of the partition induced by this set of solutions is typically quite small. As the size of this partition directly affects the size of the reduced sub-problem, it is useful to be able to control the size of partition to increase the size of the merge neighbourhood that is searched. One way to do this is through a process called *random splitting*.

Definition 2. *Given a set $\mathcal{S}$, a* **random split** *is some heuristic process that produces two sets, $\mathcal{S}_1$ and $\mathcal{S}_2$, such that $\mathcal{S}_1 \cup \mathcal{S}_2 = \mathcal{S}$ and $\mathcal{S}_1 \cap \mathcal{S}_2 = \emptyset$.*

This method of arbitrary splitting can be used to further partition the decision variables that have been aggregated, to allow the optimal solution to be produced. A proof of this is given in Lemma 2.

Simply splitting the partitions arbitrarily is unlikely to produce a useful partitioning, let alone the optimal one; therefore, it is beneficial to use some heuristic strategy to do it — this is especially true of large scale and highly constrained problems. For example, if x_a, x_b and x_c are decision variables in some partition P_i and there are constraints in the problem model that say $x_a \leq x_b \leq x_c$, it does not make any sense to split P_i such that $x_a, x_c \in P_i{}'$ and $x_b \in P_i{}''$, as the values of the merge variables representing $P_i{}'$ and $P_i{}''$ in a merged solution would have to be equal in order to remain feasible. In this case, a random splitting heuristic should be designed that takes these precedence relationships into account.

There are many ways that a random splitting heuristic can be designed. The simplest is to generate a random bit string of length n and add it to the population before the partition is defined (Figure 2).

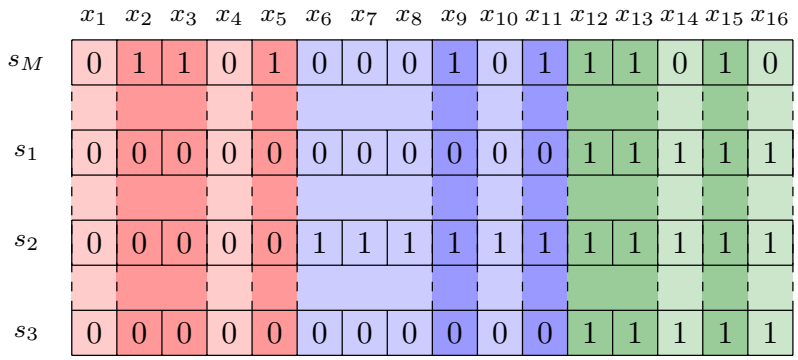


Figure 2: By adding the random bit string s_M to the population before merging, each partition is arbitrarily split into two.

In this figure, the "natural" partitions for the population $\{s_1, s_2, s_3\}$ are indicated by the blocks with shades of red, shades of blue and shades of green. By adding the random bit string s_M to the population, each natural partition has been arbitrarily split into two, indicated by the light and dark colour shades.

While this method is the simplest, it does not take into account any of the constraints, or the implicit structure of the problem. This means that, for highly constrained problems, the partitioning produced by this method is likely to be no more effective than the natural partitioning induced by the population itself. Therefore, in practice, it is wise to design a heuristic splitting method with these considerations in mind; however, if no such method is possible, it is theoretically possible to produce the optimal solution by simply using random splitting alone — when allowed enough time.

If defining a partition on a set of solutions can be said to be analgous to the crossover operator for GA, then splitting the partition is analogous to its mutation operator. As has been already established, crossover only allows solutions to be sampled from the sub-space induced by the properties of the two parents, not the entire search space; the same is true for defining a partition on the decision variables as described in the previous section — if $x_i = x_j$ across all solutions, any solution produced by merging in this way will also have $x_i = x_j$. In order to allow GA to produce any solution in the entire search space, a mutation operator is needed; the simplest version of which selects a gene at random and flips its corresponding bit. This can be seen as a special case of merge search, where a population consisting of a single solution and a single bit string with only one arbitrary 1 in it is merged. Here, the decision variables are partitioned into three groups: the group of variables that took the value 0 in the original solution; the group of variables that took the value 1 in the original solution; and the single variable to be "mutated". Again, as with the crossover analogy, whereas the mutation operator for GA merely samples the sub-space, merge search searches it to find the (locally) optimal choice.

By extending this idea further, it can also be shown that large neighbourhood search (LNS) (Pisinger & Ropke, 2010), or indeed any *destroy-and-repair* heuristic, is a special case of merge search. To demonstrate this, a population is constructed that consists of a single solution s and a set of unique bit string masks $\{M_1, M_2, \ldots, M_m\}$, each with a single arbitrary bit flipped to 1, the rest of the bits are all zeroes.

	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	x_{11}	x_{12}	x_{13}	x_{14}	x_{15}	x_{16}
M_1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
M_2	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
M_3	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
M_4	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
M_5	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
M_6	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
s	1	0	0	1	1	0	0	0	1	0	0	1	0	0	1	1

Figure 3: Each bit string mask separates its associated decision variable into a partition containing only that variable.

Figure 3 shows that when this population is merged, the decision variables are partitioned into $m+2$ groups: variables that took the value 0 in the original solution (red); variables that took the value 1 in the original solution (green); and m partitions containing a single variable, associated with each of the bit string masks (shades of blue). If the ratio of m to n is sufficient, such that enough of the original solution structure is maintained, then all those variables that took 0 or 1 in the original solution will be effectively fixed to their respective

values in the reduced sub-problem, and all those variables in the m singleton partitions are free to take either 0 or 1 in the solution to the reduced sub-problem. This is equivalent to LNS, where a subset of the variables in a given solution is selected for "destruction" (i.e., removed from the solution) and the solution is "repaired" by solving the partial solution to (local) optimality.

Solution merging (Step 8)

All of the steps leading up to this point have been working to construct a reduced sub-problem, which can now be solved, using an exact solver or some other method, to produce a locally optimal solution for use in the next iteration of the process. In very general terms, the reduced sub-problem takes the following form:

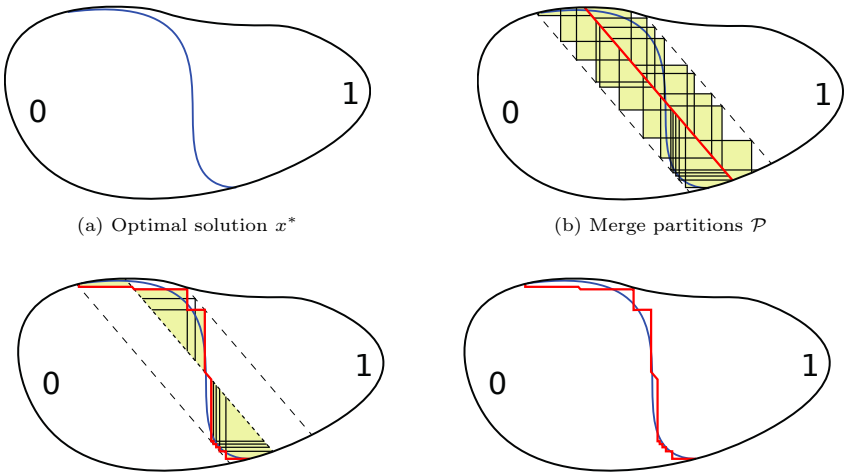
$$\begin{aligned} \text{minimise} \quad & f(\boldsymbol{x}) \\ \text{subject to} \quad & \boldsymbol{x} \in \mathcal{F} \subseteq \mathbb{Z}_2^{|\boldsymbol{x}|}, \\ & x_i = x_j \quad \forall i, j \in P, P \in \mathcal{P}. \end{aligned} \tag{1}$$

Recall, S is the generated population of solutions and $\mathcal{P} = \{P_1, P_2, \ldots, P_p\}$ is the set of merge partitions produced by the population and the random splitting heuristic.

For problems with large numbers of decision variables, it can be practical to replace the set of problem variables x with a vector of partition variables z. This will often create a certain amount of overhead in constructing the model for the reduced sub-problem as constraints need to be transformed to be in terms of partition variables instead of decision variables — and then again, when the produced solution must be re-expressed in terms of the problem variables. However, this usually results in the model taking up much less space in memory.

Finding the globally optimal solution to the reduced sub-problem will give a locally optimal solution to the master problem Figure 4 gives an illustration of this process.

The partition on the decision variables representing the optimal solution $x^* \in \mathcal{F}$ is shown in blue in Figure 4a. Figure 4b shows the merge partitions produced by merging the population S and applying the random splitting heuristic. Here, the initial solution x^{k-1} is the straight red partition boundary and each neighbouring solution in the population is represented by two triangular regions protruding from either side of this boundary. These regions represent the decision variables that take different values in the neighbouring solution, with respect to the initial — regions on the left of the boundary indicate variables that have changed from 0 to 1 in the new solution, regions on the right indicate those that have changed from 1 to 0. The population is merged and split to produce the reduced sub-problem which is expressed in terms of a set of partition variables z. This reduced sub-problem is then solved using an exact solver or some other method. Figure 4c shows that the optimal solution to the sub-problem z^* is a partition constructed from a subset of the boundaries of the merge partitions. Finally, z^* is mapped back to a solution to the master



(a) Optimal solution x^* (b) Merge partitions $\mathcal{P}$

(c) Sub-problem solution z^* (d) Merged $f(x^k) \leq f(x^*)$

Figure 4: Solving the reduced sub-problem induced by the partitions $\mathcal{P}$, gives an approximate solution to the master problem.

problem x^k (Figure 4c), an approximation of x^*. If the stopping criteria is not yet met, then the merged solution x^k is used as the initial solution for the next iteration.

Provided there is at least one feasible solution in the population at the time of merging, the merged solution x^k will always be feasible. As the merge partitions are formed by the intersections of all the members in the population S (and then randomly split), any solution $s_i \in S$ can be reconstructed by simply including all partitions that are associated with that solution. This is illustrated in Figure 1 above, where s_1 can be constructed by the union of partitions $P_2 \cup P_3 \cup P_5 \cup P_6$. If s_i is feasible, then the final merge operation is able to produce s_i as x^k; meaning that, so long as the initial solution x^{k-1} is feasible, x^{k-1} will be a lower bound on the solution produced by merging and $f(x^{k-1}) \leq f(x^k) \leq f(x^*)$, even if the entire rest of the population consists of randomly generated bit strings, representing infeasible solutions.

3.2. Properties of Merge Search

There are some simple but important properties that follow directly from the way this matheuristic has been defined.

Lemma 1. *The Merge Step 8 produces a solution that is at least as good as any of the neighbours in S: $f(x^k) \geq f(s)\ \forall s \in S$*

Proof. By construction any solution $s \in S$ satisfies $s \in \mathcal{F}$ and $s_i = s_j\ \forall i, j \in P \in \mathcal{P}$ and so is feasible for the optimisation problem in Step 8. Hence, $f(x^k) \geq f(s)$. □

Corollary 1. *Merge Search is a hill-climbing method that produces a non-decreasing sequence of solution values:* $f(x^0) \leq f(x^1) \leq f(x^2) \leq \ldots$

Hence, for diversification, the method relies entirely on the neighbourhood search in Step 3 of the algorithm and the randomised splitting in Step 7. This is not a problem for large instances such as this, where simply finding a very good local minimum is already a very challenging task. Furthermore, the following property holds:

Lemma 2. *For pure binary problems where all variables are* $0-1$*, if random splitting of subsets is used in Step 7 (allowing any possible split with some non-zero probability) with* $K \geq 2^{m+2}$ *then Algorithm 1 is guaranteed to converge to the optimal solution as the number of iterations approaches infinity.*

Proof. By assumption there exists an optimal solution in $\mathcal{F}$, due to the existence of $x^0 \in \mathcal{F}$ and boundedness of $\mathcal{F}$ when all variables are binary. Let s^* be any optimal solution. The first thing to note is that if a partition $\mathcal{P}^*$ satisfies $s_i^* = s_j^* \; \forall P \in \mathcal{P}^*, \; i, j \in P$ then the Merge problem in Step 8 yields and optimal solution. Such a partition could be generated, from any arbitrary partition $\mathcal{P}$ by splitting each $P \in \mathcal{P}$ into $P_0 = P \cap \{i \mid s_i^* = 0\}$ and $P_1 = P \cap \{i \mid s_i^* = 1\}$. This splitting at most doubles the number of elements in the partition.
Now the same argument can be used to show that $|\mathcal{P}|$ as produced in Step 6 has at most 2^{m+1} elements (corresponding to splitting based on solutions $s^1, \ldots, s^m$ and x^{k-1}). Hence, we only need K to be at least 2^{m+2} to allow some chance of generating the required partition in any step. Hence, as the number of iterations of Algorithm 1 goes to infinity the chance of *not* producing an optimal solution goes to zero. And of course, based on Corollary 1 once an optimal solution has been found, the algorithm will not depart from this. □

While convergence to the optimal solution is of course extremely unlikely for practical sized instances, the lemma shows that it is at least theoretically possible. Furthermore, while 2^{m+2} appears quite large, the only real requirement is that K is sufficiently large to allow each element of $\mathcal{P}$ to be split once, with $|\mathcal{P}|$ typically much smaller than 2^{m+1} in practice. Hence, while Merge Search is a hill-climbing method, it is at least in principle possible for the method to reach a global optimum from any starting point.

3.3. Comparison with CMSA

As mentioned previously, it may be noted here that this meta-heuristic has some similarity to the recently published CMSA heuristic by Blum et al. (2016). Although there are some similarities between merge search and CMSA, there are two areas where merge search diverges significantly from it. These are:

- the generation of candidate solutions to be merged; and
- the aggregation of decision variables in the reduced sub-problem.

The CMSA sub-problems are still defined based on a population of solutions, however these solutions are constructed probabilistically, with each solution being generated from scratch. This has the two-fold effect of reducing the capacity of CMSA to learn from the best individual solution found so far, and also makes it impractical when trying to solve large-scale, or very complicated, problems for which constructing a feasible solution is extremely time consuming, such as the CPIT problem that is considered here. Merge search avoids this issue by heuristically constructing an initial solution and then sampling its neighbourhood in order to generate a population, which is then used to define its sub-problems. The effect of this is to make the time taken to generate the population of solutions dependent on the local search algorithm used to sample the neighbourhood of a given feasible solution which is often much faster than the algorithm used to find feasible solutions from scratch. Of course, this leaves merge search potentially susceptible to being overly sensitive to the quality of the initial solution; however, this can be mitigated by increasing the diversity of the population, or further splitting the merge partitions.

The second area where there is a significant divergence between the two methods is in the aggregation of decision variables. CMSA uses its population of constructed solutions to determine the variables that are to be included in its reduced sub-problem. If a variable is represented by an element of one of the candidate solutions in the population, it is automatically included in the reduced sub-problem. These variables are added to the sub-problem individually, and as such this aspect functions similarly to large neighbourhood search (LNS). One consequence of this is that the sub-problems can become very large, especially for problems where there are naturally many non-zero values in a solution. To combat this, a so-called "ageing" mechanism is introduced in CMSA to ensure that elements that have not been useful in producing good quality solutions recently are removed from the pool of solution elements. In contrast, merge search uses information from across the entire population to determine which variables are added into the reduced sub-problem as well as to aggregate variables that share common values across the population, and so likely share some kind of dependency. This grouping allows for more compact sub-problems and a much larger region of the search space can be covered for the same computational power. The trade-off for this is much coarser-grained solutions, however this can be alleviated by the introduction of random splitting to these groups to help escape local optima.

For a more thorough comparison between merge search and CMSA, see Thiruvady et al. (2020).

4. Merge Search in Practice

Section 2 introduced the two problems that merge search was applied to in order to demonstrate its effectiveness and versatility: the constrained pit-limit (CPIT) problem; and the Steiner tree problem in graphs (STPG). This section gives the details of how merge search was applied to solve them.

4.1. The CPIT problem

Kenny et al. (2017) describe a novel representation for the precedence constrained production scheduling problem (PCPSP) — a related problem to CPIT — and a greedy randomised adaptive search procedure (GRASP) algorithm for solving it. They build on this with local search operator for the CPIT problem, and use it in a simple merge search framework that operates without variable partitioning or random splitting (Kenny et al., 2018). By incorporating a variable partitioning mechanism, they show that a much larger region of the search space is able to be explored for the same computational budget (Kenny et al., 2019). They exploit the structure of the problem, and treat many decision variables as a single group. This allows the size of the mixed-integer programming sub-problem to be greatly reduced and hence, the time required to solve it. A modified version of the CPIT formulation is given in Appendix Appendix A (Equation A.2). **(AK: put formulations in body)**

The algorithm presented in this paper is an exentsion of this work, with two additions: a random splitting heuristic, to allow greater granularity in the solutions produced; and a "solution polishing" technique, based on the local improvement heuristic from the GRASP algorithm.

The random splitting heuristic is based on the local search neighbourhood in Kenny et al. (2018). An arbitrary variable in a partition is selected and all other variables that are in the predecessor cone for that variable, in the time-expanded problem graph, are computed and split from the partition. This preserves the precedence relationships between the relative position of blocks in time and space. As the number of variables in the reduced sub-problem increases with each split that is made, the size of the reduced sub-problem can be controlled; and this splitting process performed until the desired size is reached.

A solution polishing step is used to locally improve the best solution produced by merge search. A "sliding window" mechanism is employed which solves a reduced sub-problem that has all decision variables fixed except for those in time period t and $t+1$, effectively allowing blocks to swap between time periods. The window of free variables is then moved to periods $t-1$ and t, and the process continues until the stopping conditions are met. This produces higher quality solutions, however the results in Section 6 show that the combination of merge search and this polishing technique produces better results than by polishing alone.

4.2. The Steiner tree problem in graphs

First described by Dowsland (1991), but subsequently used widely by many researchers, is the so-called key path neighbourhood. A key path is defined as a path within a Steiner tree where the two end vertices are either terminal vertices or vertices of degree at least 3; all intermediate vertices (if any) are of degree 2 and are not terminal. The useful property of such structures is that their removal from a solution to the STPG will always result in two disconnected trees which can then be subsequently reconnected. This local search neighbourhood was

extended by Kenny et al. (2016), by adding a so-called "jump" operator which aids in escaping local minima, and is used as the basis of the merge search algorithm presented in this paper.

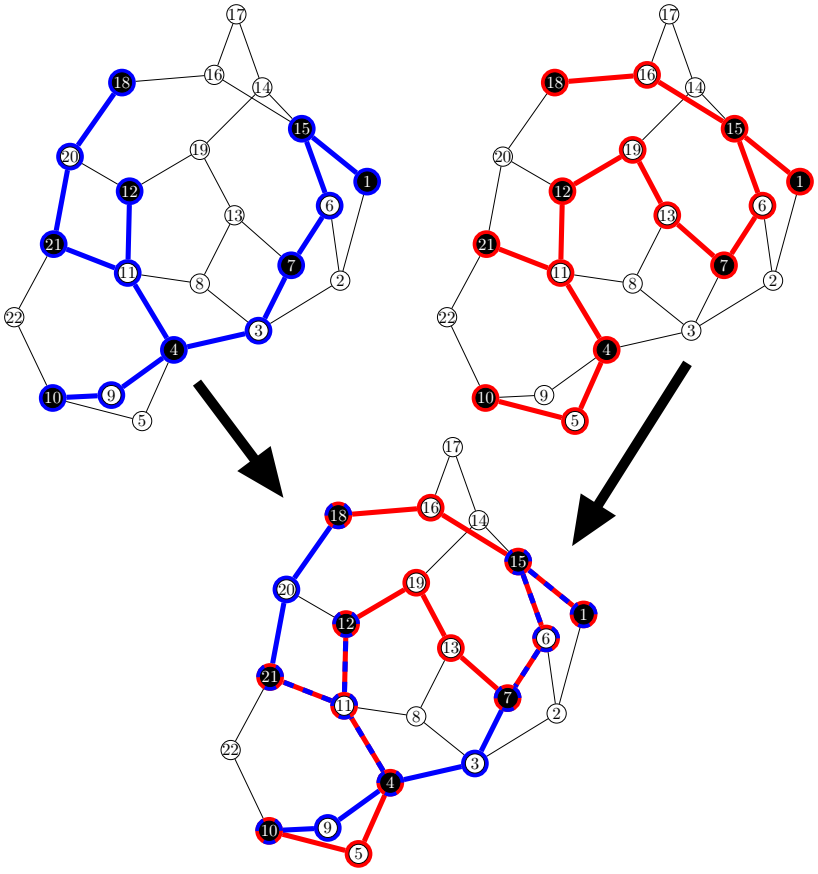


Figure 5: By merging two solutions (top left and right), the decision variables are partitioned into four disjoint sets, indicated here (bottom) by: red; blue; red and blue; and no colour.

The problem instance is first pre-processed using some of the techniques described in Duin (2000); Uchoa et al. (2002); Kingston & Sheppard (2003) to reduce the number of decision variables. An initial solution is constructed and a neighbouring population produced, using the methods described in Kenny et al. (2016). This population of solutions is used to partition the decision variables in the manner described in Section 3, illustrated in Figure 5.

These partitions can be split by selecting an arbitrary decision variable, and separating all variables representing vertices and edges that are connected to it and in the same partition. This ensures that any partition is split such that all variables removed from the original partition comprise a connected sub-graph, and as such should be able to be included in the resultant merged solution on its

own, without the original partition. If a partition is split such that all variables are separated, leaving the original partition empty, then the last variable added to the split is kept in the original partition. As the connected variables are added using DFS, this is always a variable on the boundary, and so this method will not suffer from the problems present with truely random splitting.

Having partitioned and split the decision variables, the resulting reduced sub-problem can be solved using a version of the Goemans & Myung (1993) MIP formulation that has been modified be expressed in terms of a single set of partition variables, instead of edge and vertex variables. The original Goemans & Myung formulation and its merge search variation are included in Appendix Appendix B (Equations B.1 and B.2). **(AK: put formulations in body)**

5. Experiments

This section details the experimental set up, the experiments performed and finally presents the results and discusses the implications those results suggest.

5.1. Datasets

The CPIT problem

Table 1 details the properties of the problem instances used to test the algorithm in this paper. The instance name is given in the first column; followed by the number of blocks in the orebody model; the number of precedence arcs for each instance is provided in the third column; the fourth column gives the total number of time periods available; the number of decision variables is shown in the second last column; with the last column giving the total number of constraints in the problem model.

Table 1: Characteristics of *minelib* (Espinoza et al., 2012) datasets.

Instance	Blocks	Precedences	Periods	Variables	Constraints
`newman1`	1,060	3,922	6	6,360	29,904
`zuck_small`	9,400	145,640	20	188,000	3,100,840
`kd`	14,153	219,778	12	169,836	2,807,196
`zuck_medium`	29,277	1,271,207	15	439,155	19,507,290
`marvin`	53,271	650,631	20	1,065,420	14,078,080
`zuck_large`	96,821	1,053,105	30	2,904,630	34,497,840

A few instances from the full *minelib* dataset were omitted due to missing files, referencing more than two resources or being too large for the algorithm in its current incarnation.

The Steiner tree problem in graphs

The experiments were were carried out on categories B, C, D and E of the STPG problems from the *steinlib* library (Koch et al., 2001), a standard

dataset used by many researchers. The smaller-scale, category B instances are 18 randomised networks with 50 to 100 vertices and 63 to 200 edges. The C and D datasets consists of 20 larger randomised networks, with 500 and 1,000 vertices, respectively, and between 625 to 25,000 edges. Finally, the E dataset contains the largest instances, each with 2,500 vertices and between 3,125 and 62,500 edges. Details of the individual instances can be found in Table C.6 of Appendix Appendix C.

The optimal solutions for these instances are reported in the library and have been proven by exact methods such as branch-and-bound with graph reduction techniques.

5.2. Pre-processing

Being graph-based problems, both the CPIT problem and STPG are amenable to some amount of pre-processing. Detail about the pre-procesing methods used for the CPIT problem experiments can be found in Kenny et al. (2017). For information about pre-processing techniques for the STPG, see Duin (2000); Uchoa et al. (2002); Kingston & Sheppard (2003).

5.3. Experimental setup

The experiments were carried out on an *Intel® Core™ i5-2320* processor (3.0GHz) with 24GB RAM running Linux. All code was implemented in C++ with GCC-4.8.0. CPLEX Studio 12.7, operating with a single thread due to the need for callbacks, was employed as the mixed integer programming (MIP) solver. For the CPIT problem experiments, the boost library implementation of the Boykov-Kolmogorov algorithm was used to solve the UPIT sub-problem during the pre-processing stage.

The CPIT problem

The merge search algorithm was compared against the baselines of the original results published on the *minelib* (Espinoza et al., 2012) website, the most recent state-of-the-art results published in Jélvez et al. (2019) and the results from using a greedy randomised adaptive search procedure (GRASP) heuristic, adapted from Kenny et al. (2017).

Also provided are the results for a variant of the merge search algorithm that uses variable grouping but no random splitting, previously published in Kenny et al. (2018); and the results of combining merge search with a solution polishing phase. These are included in order to illustrate the effect that random splitting has on solution quality, and to demonstrate the effectiveness of combining an explorative technique with an exploitative one.Each algorithm was run 20 times on each instance, recording the mean objective value and standard deviation of the resulting solution produced by each run.

The Steiner tree problem in graphs

The merge search algorithm was compared against three separate baseline algorithms, the greedy randomised adaptive search (GRASP) heuristic, pure local search and pure MIP. The method of constructing initial solutions was different between the GRASP and merge search algorithms, so pure local search and pure MIP were included to ensure any improvement was not solely based on this factor. Aside from population merging, the merge search algorithm comprises two main components, local search and MIP search; so by isolating these two factors, it can be shown that merge search is greater than the sum of its parts.

Each algorithm was run 30 times on each (pre-processed) instance from the datasets, recording the mean objective value and standard deviation across all of the runs. All instances in the datasets used are supplied with their optimal objective values and this is used for the main stopping criteria. Otherwise, the stopping criteria of the merge search algorithm is generally dictated by the number of seconds spent in the MIP search so the time taken for each search is not provided, as it does not give much information about the performance of the algorithm. However, special mention is made when the algorithm terminated early for all runs.

Additional experiments were performed to investigate the difference between using the random and deterministic solution construction heuristics and the effect of population size on the size of the reduced sub-problem. In order to preserve space, a representative subset of the problem instances is used to illustrate the outcome of these experiments; however, these results are not "cherry-picked" and the full tables are available in Appendix Appendix D.

6. Results and discussion

6.1. The CPIT problem

Table 2 provides the results of the three algorithms tested on the six problem instances from the *minelib* dataset, along with the linear programming (LP) upper bound, the best solution as published on the *minelib* website and the current best state-of-the-art solution as published in the survey paper from Jélvez et al. (2019) — ordered in ascending problem size.

In this table, it can be seen that all three merge search variants consistently produce better results than the *minelib* and the GRASP heuristic results, except for `newman1` and `zuck_large`. The results for `newman1` are similar because the problem instance is so small and it can be assumed that the LP gap of 1.26% is quite close to the optimal solution.

Comparing the results of the two variants of merge search (without polishing) illustrates the effect that random splitting has on the quality of the solution produced. For smaller problem instances, the two algorithms produce very similar quality solutions, indicating that the random splitting has little effect. However, as the size of the problem increases, the effect of the random splitting

Table 2: Results on *minelib* dataset instances. Given are the LP upper bound, the percentage gap between the LP bound and the best solution from the *minelib* website; the best known solutions as published in Jélvez et al. (2019) (with references to the original publications containing these results); the mean result from the GRASP heuristic; and the mean results from the merge search heuristic with no splitting, splitting and solution polishing, respectively.

					merge search with:		
Instance	LP UB	*minelib*	state-of-the-art	GRASP	no splitting	splitting	polishing
`newman1`	2.449E+07	4.12%	**1.26%**[a]	**1.26%**	**1.26%**	**1.26%**	**1.26%**
`zuck_small`	8.542E+08	7.67%	**0.71%**[b]	2.44%	1.28%	1.29%	0.83%
`kd`	4.095E+08	3.08%	0.14%[b]	0.29%	0.17%	0.22%	**0.10%**
`zuck_medium`	7.106E+08	13.40%	**5.24%**[b]	7.85%	7.35%	6.98%	5.88%
`marvin`	8.639E+08	5.00%	**0.64%**[b]	3.13%	1.16%	1.14%	0.87%
`zuck_large`	5.739E+07	1.05%	0.24%[c]	8.19%	5.18%	3.75%	**0.17%**

See: [a]Samavati et al. (2017), [b]Samavati et al. (2018), [c]Jélvez et al. (2016)

becomes more pronounced, with the biggest effect being on the two largest instances `zuck_medium` and `zuck_large`[1].

If there are no overlaps at all between variables across solutions, then the partitions in the reduced sub-problem are simply the set differences of the variables in each solution and the seed solution. In this extreme case, the merge operation is unable to produce a solution that does not already exist in the population, unless some random splitting of the partitions is performed. As the size of the problem instance decreases, the likelihood that there will be overlaps between variables across solutions increases. These overlaps in the variable sets produce splits in the partitions when producing the reduced sub-problem, reducing the need for additional random splitting to produce a different solution to those already existing in the generated population. This is not to suggest that random splitting would not be beneficial; but the problem with random splitting is that it is *random*, and there is no way of guaranteeing that a particular split will improve the quality of a solution after merging, any more than a split produced by overlapping variable sets will.

The GRASP heuristic produces consistently worse results than merge search. This is expected as the sliding window heuristic used in the local improvement phase of the algorithm is better suited to incrementally improving a good solution than turning a mediocre solution into a good one, as it only operates on a small subset of the variables at one time. This means that it is good for "tweaking" a solution by shifting the time that a block is mined forward or backward one or two periods; but it is no good if the time that the block is mined must be moved by many periods, as this would take a lot of passes to achieve.

[1] Although it is technically a larger problem than `zuck_medium`, solving the UPIT problem on the `marvin` instance, as part of the pre-processing stage, eliminates many blocks and makes its effective size much smaller than `zuck_medium`.

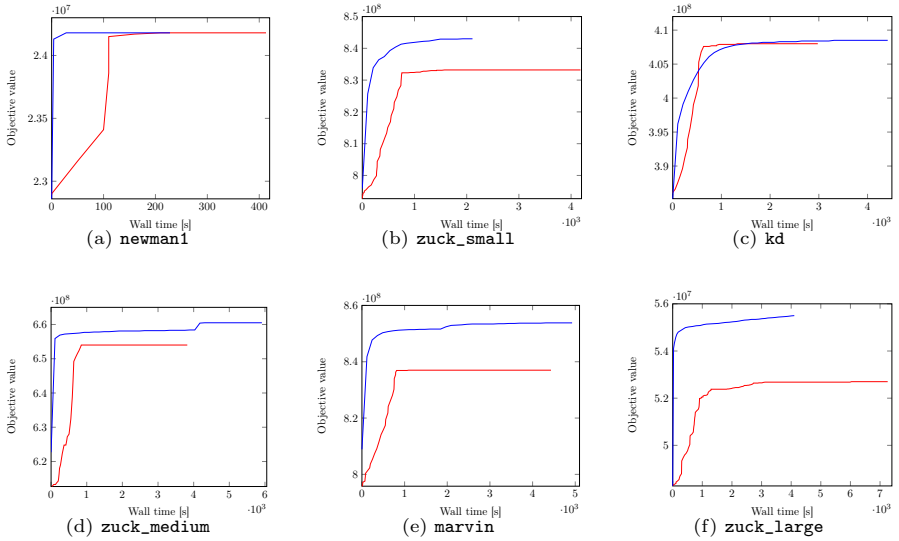


Figure 6: Convergence plots for CPIT instances. Merge search data is shown in blue and GRASP is shown in red. **(AK: should i change these so that they are showing the LP gap instead of absolute value?)**

Convergence behaviour

This idea of GRASP being better suited to incrementally improving an already good solution is demonstrated in Figure 6. This figure gives the plots of the convergence behaviour of both the GRASP algorithm and merge search on all six of the CPIT instances. It can be seen in these plots that merge search converges quicker in nearly all instances, except for `kd`.

The difference here, is that the initial solution is already quite close in value to the resultant solution, so there are not a lot of improvements that can be made. This means that the more exhaustive search for small improvements of the GRASP algorithm is more likely to be effective, early on, than the more global search of the merge search algorithm, which relies on the stochastic natures of the local search operator and arbitrary splitting heuristic to find its improvements. Merge search does get there in the end — and manages to find a slightly better solution in this case — but it takes longer and with a more gentle curve. For all the other instances, where the gap between the initial and resultant solutions is much larger, merge search is demonstrably more suited to the task.

It can also be seen from these plots that once the algorithm has seemingly converged, it can sometimes find a way out of the local optima and locate a much better solution. This is evidenced in the plots for `zuck_medium` (Figure 6d), `marvin` (Figure 6e) and also Figure 8a below. This behaviour is expected in such large problem instances, as once it has started to converge there are many ways to make the solution worse, but only a few to make it better; but once it has found a way out of the local optima, often several other improving

moves will become apparent as well.

Runtime information

Table 3 gives the runtime information (wall time and CPU time) for both GRASP and merge search. As they use a MIP solver to solve their restricted sub-problems, the runtime of both algorithms is reasonably easily configured by controlling how long the solver is allowed to run for each iteration. The parameters of the GRASP algorithm were chosen so that the search would take roughly the same amount of time as that of the merge search — and for the most part, they are pretty similar.

Table 3: Runtime information for experiments on CPIT. Given are the mean (μ) and standard deviations (σ) of the wall and CPU time for the GRASP and merge search algorithms, in seconds.

	GRASP				merge search			
	Wall time [s]		CPU time [s]		Wall time [s]		CPU time [s]	
Instance	μ	σ	μ	σ	μ	σ	μ	σ
`newman1`	453.2	33.8	1,430.6	88.0	227.5	17.6	710.9	41.7
`zuck_small`	4,215.3	152.5	12,128.7	563.3	4,214.2	188.6	9,938.5	636.8
`kd`	3,023.2	307.5	8,459.5	844.9	4,378.3	175.3	11,525.9	845.0
`zuck_medium`	4,021.7	252.3	9,123.6	478.4	5,487.0	248.8	16,843.3	745.1
`marvin`	4,388.8	252.6	14,653.2	753.4	4,867.8	214.7	15,072.5	396.4
`zuck_large`	7,081.3	453.8	18,066.7	1,399.9	3,978.3*	152.1	9,745.8	786.9

*population size for `zuck_large` set at 500 for merge search due to insufficient memory.

The two instances that are significantly different in runtime between GRASP and merge search are `kd` and `zuck_large`. The reason that `zuck_large` is so different is that it was impossible to run the merge search algorithm with a population of 1,000 on such a large problem instance, due to memory issues (even with 23 GB of RAM!); so a smaller population size of 500 was used, which took much less time to produce and to compute the merge partitions.

The reason for the differences in runtime on `kd` is illustrated in Figure 7. This figure shows plots of the amount of wall time in seconds consumed per iteration of the respective algorithms. The GRASP algorithm is measured by "window movements", which counts the number of times the window has incremented forwards or backwards along the solution. The merge search algorithm is measured by iterations; each the time for each iteration is measured at the point when a new candidate solution is generated for the population, so it is clear to see that, with a population size of one thousand, every thousandth iteration will include an extra amount of time to include the merge operation itself.

It is important to remember here that the `kd` instance only has a maximum of 12 periods allowed, while `zuck_small` has 20. So, while the number operations for GRASP is dependent on the number of periods (and therefore window movements), merge search is dependent on population size. This is demonstrated very clearly in Figure 7b, where there is very little difference between

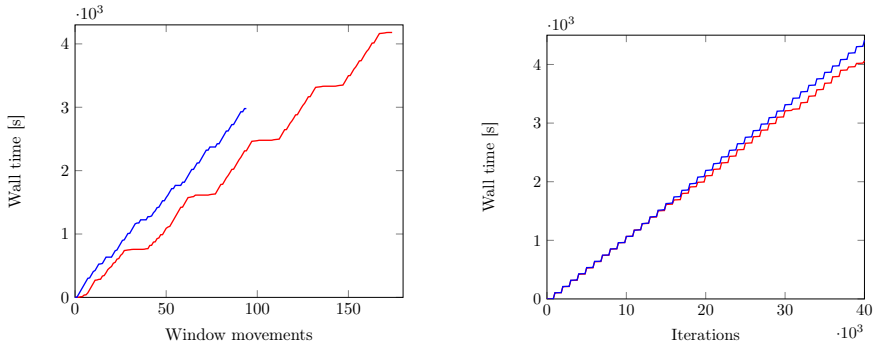


Figure 7: Plot of wall time [s] for `zuck_small` (red) and `kd` (blue) instances. Sub-figure (a) shows GRASP algorithm data and (b) shows merge search data.

the two instances. It can be seen that the population is generated reasonably quickly (indicated by the flat parts of the curve) and then each point where there is a sudden jump in the amount of time taken indicates that a merge operation has taken place.

But this is not the end of the story. In these two examples, GRASP took 2,979 seconds to complete the search on `kd` and it took 4,180 to complete the search on `zuck_small`. If there was a direct linear relationship between time taken to complete the search and number of periods, the amount of time GRASP should take to complete its search on `zuck_small` should be $\frac{2{,}979 \times 20}{12} = 4,965$. So where did the other 13 minutes go?

Looking at Figure 7a, the main thing that can be noticed here is that the shape of the plot for `zuck_small` is significantly different to the shape of the plot for `kd`. The regular flat spots in the plot for `zuck_small` indicate that there is a group of periods for that problem that are not used by the solution, or do not contain any blocks that can be moved around, and therefore the MIP solver will not take the full amount of allowed time to solve the reduced sub-problem. It so happens that the solution to `zuck_small` does not use any period after period 16, so the first and last few window moves of each full pass take very little time at all. In contrast the `kd` instance uses periods right up until period 10, so there are fewer window moves that will be skipped over, as evidenced by the straighter line on the plot.

Solution polishing

The results in the last column of Table 2 show the effect of combining merge search with the local improvement heuristic from the GRASP algorithm. These results clearly demonstrate the effectiveness of this combination, even improving the best known bounds for `kd` and `zuck_large`.

For these experiments, the best solution from the merge search runs is taken and then three passes of the sliding window heuristic is applied. Although doing this effectively increases the allowed computational budget, the quality of the

solutions produced by doing this is better than if the computational budget is increased for either of the two algorithms on their own. In the case of the GRASP algorithm, the quality of the solution at the end of its run is still too low to be able to improve the objective value by very much in only three passes. In the case of the merge search algorithm, because both the population generation and merge partition splitting occurs stochastically, as the quality of the solution increases the chances of finding an improving solution, or partition split, by chance decreases. This means that the search is likely to have converged by the end of its run, so increasing the computational budget at this point is unlikely to produce much of an improvement in the quality of the solution, if any at all.

Applying the sliding window heuristic at the end of the merge search process solves both of these problems. The sliding window heuristic performs a much more methodical and exhaustive search to find small improvements that can be made to the solution which might be difficult to find by random sampling, in such a large search space. Additionally, by starting the sliding window heuristic with a much higher quality initial solution, computational effort does not need to be wasted on the "low hanging fruit" that can be found easily by sampling the search space. As the heuristic only considers a small subset of the variables at a time, it cannot make large changes to the solution in a single iteration.

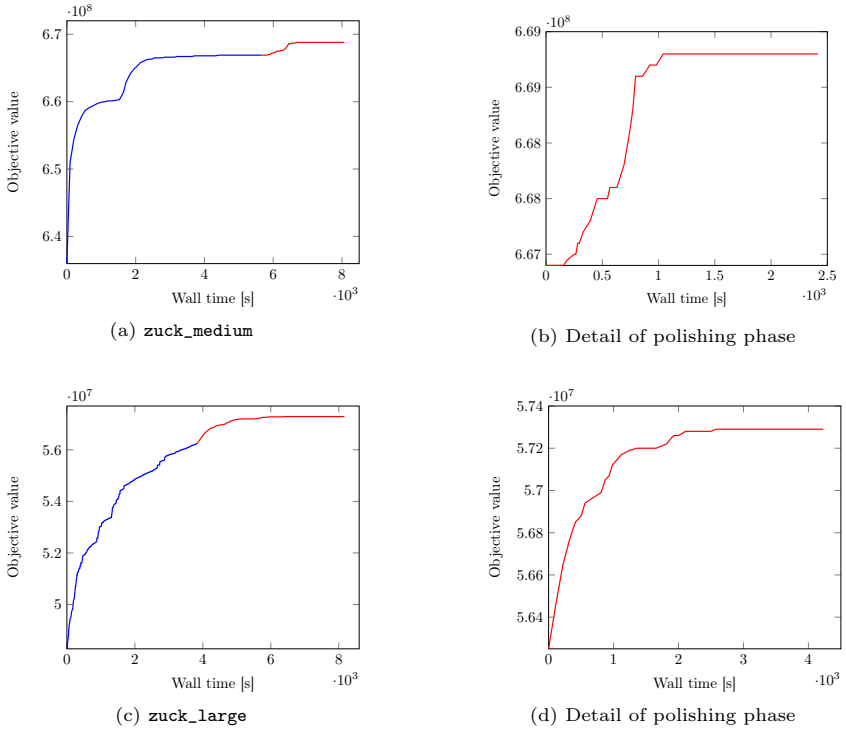
Figure 8 contains plots that illustrate this concept on two instances of the CPIT problem, `zuck_medium` and `zuck_large`. The left-hand sub-figures show the full process with both merge phase (shown in blue) and polishing phase (shown in red). Figure 8a demonstrates the power of intensification of the search with the sliding window heuristic. Here, the blue merge plot can seen to have effectively converged as there has been no significant improvement in objective value for over half an hour of wall time. Applying the sliding window heuristic to this "converged" solution results in an almost immediate improvement, followed by subsequent improvements, before this too converges to its local optimum.

Not as emphatic in demonstrating this point are the results presented in Figure 8c. It can be seen in this figure that merge search was still a ways off from converging, although it had started to slow its progress. It could be argued that, were merge search allowed to continue, it would have found the same solution eventually. Although, it can be seen in the plot that applying the polishing heuristic to this unconverged solution did still result in an initial rapid increase in solution quality; and as merge search was beginning to tail off, it probably would have taken longer to reach its goal.

The results from these experiments suggest that, while merge search is adept at finding a good quality global solution, it benefits from the addition of a more exhaustive local method to intensify its search.

6.2. The Steiner tree problem in graphs

Table 4 provides a representative sample of the results for the experiments performed to compare the merge search algorithm to the three base-line algorithms of GRASP, pure local search and pure MIP. Given is the known optimal solution, and the percentage gap between the optimal and the mean objective



(a) `zuck_medium`

(b) Detail of polishing phase

(c) `zuck_large`

(d) Detail of polishing phase

Figure 8: Plot of merge search with solution polishing on `zuck_medium` and `zuck_large` instances. Figures (a) and (c) show the merge search (blue) and polishing (red) phases; Figures (b) and (d) show details of the polishing phases.

value from all of the runs. The best results for each problem instance are indicated in bold; and, as the optimal solution is known for each instance, if the algorithm managed to find the optimal solution in any of its runs, this is indicated by an asterisk — clearly, if a result has a gap of 0.00%, this indicates the optimal solution was found in every run. The full results are available in Appendix Appendix D.

The results for the B dataset show that merge search demonstrably outperformed all of the algorithms it was tested against, as do the rest of the datasets. In this first dataset comprised of all the smallest sized instances (around 60 to 290 decision variables, pre-processed), merge search performed the best; managing to find the optimal solution in every run for every problem instance, with many terminating in the first couple of iterations. The next most successful algorithm was the pure MIP algorithm[2], which just edges out pure local search and

[2]It should be pointed out here that the pure MIP algorithm failed to produce even a single integer solution for most of the problem instances in the time allotted to it, if it was not given a heuristically constructed solution as a warm start. Therefore the success of the pure MIP algorithm over the GRASP algorithm for this dataset should be attributed to the fact that the

finally followed by GRASP at the end. The fact that both pure local search and pure MIP outperformed GRASP suggests that the quality of the initial solution is very important to the quality of the resultant solution — especially in these smaller problem instances — so, as pure local search and pure MIP both use the same, deterministic, solution construction as merge search, they are likely to perform better than GRASP, which uses the random solution construction heuristic.

Table 4: Representative sample of results across *steinlib* dataset instances, giving the optimality gap for the mean objective value of pure local search, pure MIP, GRASP and merge search, respectively. An asterisk indicates that the optimal solution was found at least once in all the runs.

Instance	Opt.	pure LS	pure MIP	GRASP	merge search
b01	82	**0.00%***	**0.00%***	1.44%*	**0.00%***
b07	111	**0.00%***	**0.00%***	**0.00%***	**0.00%***
b14	235	0.42%*	0.21%*	1.84%	**0.00%***
b18	218	**0.00%***	0.93%*	0.64%*	**0.00%***
c01	85	**0.00%***	2.19%*	**0.00%***	**0.00%***
c07	102	0.41%*	10.6%	3.23%*	**0.00%***
c14	323	1.34%	4.15%	3.98%	**0.48%**
c20	267	0.37%	0.37%	10.46%	**0.33%***
d01	106	**0.00%***	0.89%*	0.19%*	**0.00%***
d07	103	**0.00%***	1.15%*	**0.00%***	**0.00%***
d14	667	1.59%	5.12%	5.98%	**0.67%**
d20	537	1.14%	1.47%	16.98%	**0.87%**
e01	111	**0.00%***	10.99%	**0.00%***	**0.00%***
e07	145	2.29%*	12.47%	2.42%*	**0.62%***
e14	1,732	2.42%	5.25%	15.98%	**0.86%**
e20	1,342	1.05%	1.18%	31.85%	**0.84%**

The picture begins to change slightly with the results for the C dataset. Merge search is still the clear winner of the four here; although it has not managed to achieve the optimal solution in every single one of its runs and has actually failed to reach the optimal solution in a few of the instances. The big change here is the fact that pure MIP has fallen to last place for almost every single problem instance, even with the warm start seeding; and by the D dataset it is last in every problem instance. This suggests that, by the time the problem instances reach even this moderate size (between 400 and 5,280 decision variables), the problem is too large to be solved by a MIP solver alone, using comparable computing resources. This is evidenced by the fact that the standard deviation is zero (see Appendix Appendix D) for many of

MIP search was initially seeded with a higher quality solution than the GRASP algorithm.

the instances, suggesting that it had a hard time finding a better solution than it was seeded with.

It is worth mentioning that some of the state-of-the-art results in the *steinlib* library have been produced by a combination of pre-processing and integral LP formulation, so this does not mean that these results suggest the problems are too large for MIP solvers in general, merely that they are too large for the simple formulation used for this research. Better, more complicated MIP formulations for the STPG are available (Goemans & Myung, 1993; Byrka et al., 2010; Chakrabarty et al., 2010); however, these experiments are not intended to advance the state-of-the-art for STPG solvers, but to demonstrate the application of merge search to the STPG and that there is a non-trivial benefit that can be gained from doing so. If both pure MIP and merge search used better formulations, and more sophisticated pre-processing techniques, they would indeed be able to achieve better quality solutions; but one could expect a similar gulf in quality between the pure MIP and merge search to emerge — albeit, on much larger problem instances.

The trend of pure local search outperforming the GRASP heuristic continues throughout the rest of the datasets, with the exception of problem instances `c11`, `d06`, `d16` and `e02`. Differences between the mean objective values for these instances are small enough to suggest that these are anomalous and probably the result of the random construction heuristic finding a better initial solution to the deterministic heuristic, or simple luck during the local search.

The fact that pure local search outperformed GRASP in nearly every problem instance suggests that the choice of initial solution is very important to the quality of the solution produced, as otherwise, these two algorithms are nearly identical; so the use of the deterministic construction heuristic is preferable to the random one. Added to this, the fact that merge search equals — or outperforms — pure local search in every problem instance suggests that there is indeed a benefit to creating a hybrid meta-heuristic using a MIP solver that solves a reduced sub-problem induced by merging a population of solutions produced by the local search neighbourhood. Finally, the fact that the merge search algorithm equals — or outperforms — the pure MIP algorithm in every problem instance suggests that these benefits that are gained are not simply from the addition of a MIP solver itself; and therefore, must be the result of the hybridisation and population merging processes.

Additional experiments

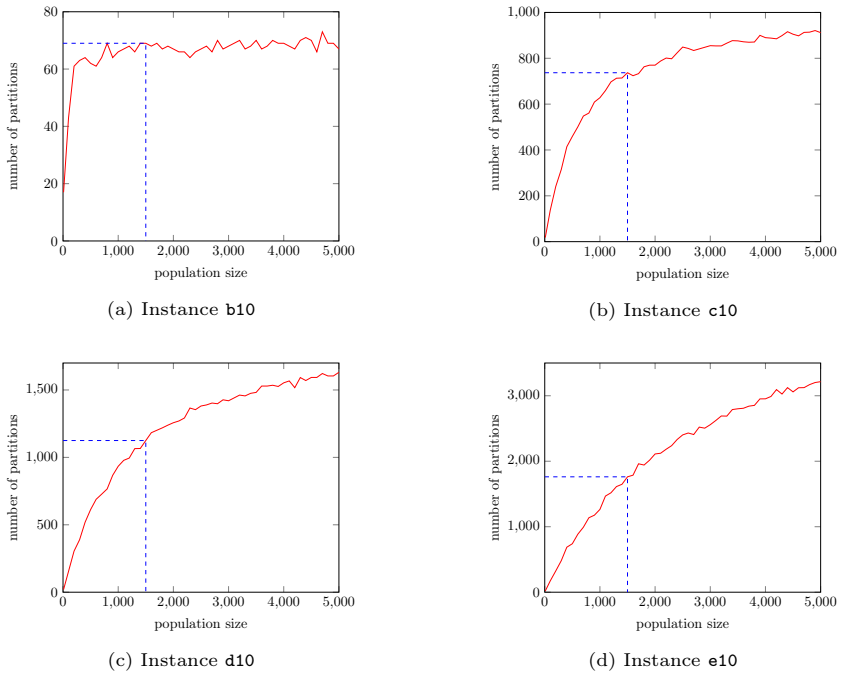
The results in Table 5 show the effect that changing the population size has on the number of partitions in the resultant reduced sub-problem, before random splitting is applied. It can be seen from the data that as the size of the population increases, the number of partitions also increases. This is to be expected, as two variables are included in the same partition only if they agree on values across the entire population of solutions; therefore, if more solutions are considered, the less likely variables are to agree across them all and the more partitions in the reduced sub-problem. Again, these are just a representative set of the problem instances, the full results are given in Table D.8 of the appendix.

Table 5: Representative sample of results across *steinlib* dataset instances, investigating the effect that population size has on the number of partitions in the reduced sub-problem. Given are the size of the instance ($|V|+|E|$) and average number of partitions and standard deviation over 30 runs for population sizes 50, 100, 1,500 and 10,000.

		50		100		1,500		10,000	
Instance	size	μ	σ	μ	σ	μ	σ	μ	σ
b01	57	12.40	2.06	15.60	0.80	19.60	0.49	20.00	0.00
b07	93	13.60	1.74	18.60	2.94	29.40	0.49	30.00	0.00
b14	148	43.80	2.79	56.40	3.50	76.20	0.40	77.00	0.00
b18	282	54.60	7.20	82.20	5.98	155.60	2.42	163.60	1.20
c01	421	19.80	2.23	19.20	1.72	28.00	0.63	28.80	0.40
c07	1,259	39.80	3.49	50.20	5.42	70.40	2.15	68.40	1.85
c14	2,695	96.60	7.68	176.20	8.33	912.80	14.76	1,377.00	7.92
c20	5,270	97.60	2.42	188.40	6.15	1,545.80	24.11	2,860.80	27.35
d01	799	20.20	4.96	29.60	0.80	30.60	0.80	30.20	0.40
d07	2,504	25.75	6.61	43.25	7.79	72.20	3.31	78.25	4.82
d14	5,658	93.75	3.63	183.25	4.60	1,426.80	17.59	2,696.75	22.00
d20	11,471	98.00	1.58	201.50	2.96	2,241.00	10.60	5,009.00	27.50
e01	1,972	27.75	7.46	32.75	2.05	37.00	3.41	35.75	2.49
e07	6,277	56.50	7.09	86.50	3.84	231.00	25.79	245.75	15.07
e14	14,617	96.00	2.55	191.25	7.50	2,281.80	22.48	5,835.25	45.78
e20	31,593	97.75	6.02	190.00	5.10	2,951.40	18.10	9,532.50	19.93

Figure 9 shows some of the plots that were used in the sensitivity analysis done to determine an appropriate population size for these experiments. The number of partitions in the reduced sub-problem was plotted against population size from 0 to 5,000 and a population size of 1,500 was decided upon for two main reasons. The first reason is that it can be seen in the plots that although the number of partitions is proportional to the population size, it is not directly proportional, and after a certain inflection point there are significant diminishing returns in the number of partitions for population size. It can be seen from the plots that the position of this inflection point moves further to the right with the size of the problem and it was decided that 1,500 was a decent value that could be used across the whole dataset. It is slightly after the inflection point for most of the C dataset instances and slightly before the inflection point for most of the D dataset instances. The B dataset instances are small enough that it doesnt matter that 1,500 is probably too many and, although it falls very short of the E dataset instances inflection point, larger population sizes produce too many partitions for the MIP solver to handle.

The second reason for choosing this population size is that in order for the MIP solver to be effective, the number of decision variables must be kept at a reasonable amount. It was decided that the number of partitions in the reduced sub-problem, after random splitting, should a maximum of around 2,000, on average. The number of random splits was set at 500, therefore the number of partitions before splitting should be around a maximum of 1,500 — which



(a) Instance `b10`

(b) Instance `c10`

(c) Instance `d10`

(d) Instance `e10`

Figure 9: Effect of population size on number of partitions in the reduced sub-problem. Dotted line shows population size of 1,500, which was used for these experiments.

occurs at a population size of around 1,500 for many of the problem instances. For larger problem instances which need a larger reduced sub-problem, an extra parameter was added, which ensures that the reduced sub-problem is some fraction of the total instance size.

7. Conclusion and future work

Constrained optimisation problems involve the search for an optimal object in (often *very* large) search spaces — in the case of combinatorial optimisation, these spaces are finite and discrete. As well as this, problems that are abstractions of real-world processes can be extremely complex; having many additional side-constraints, separate to the main problem constraints themselves, which must be considered when modelling the problem mathematically.

Karp (1972) showed that NP-Complete problems are all theoretically reducible to one another; however, in practice, they often require specific domain knowledge that requires the development of custom algorithms for each case. The discovery of a "one size fits all" solution for this class of problems would be a vital achievement in the field of operations research, and the purpose of this research is to attempt to provide a material step in the direction towards this goal.

Merge search, the algorithm proposed here, is a generalised hybrid meta-heuristic framework for solving large-scale and complex constrained optimisation problems — with, or without specific domain knowledge. The operation of this framework can be broken down into several phases:

- *Initial solution construction:* a feasible solution to the problem is constructed, either heuristically or by solving a generic mixed integer program (MIP) model of the problem that penalises non-feasible solutions;
- *Population generation:* a population of neighbouring solutions is generated, using some local search heuristic or by randomly perturbing an initial solution;
- *Partitioning variables:* the population of solutions is used to determine how the set of decision variables are partitioned. Additional splits to these partitions can be made to increase the granularity of the solutions produced; and,
- *Solving reduced sub-problem:* new local optima can be found by solving a reduced version of the problem which uses each partition as a meta-variable, either with a MIP solver or by some other method. By grouping the decision variables, this sub-problem requires much less computational effort to solve — the trade-off being that the solution produced becomes an approximation of the optimal one.

A theoretical description of merge search was given and it was also shown that merge search can be considered a generalisation of many optimisation techniques such as crossover and mutation operators in genetic algorithms, or search techniques like large neighbourhood search.

Practically, two problems were used as testing-grounds to demonstrate the effectiveness of merge search and explore some of its features and facets. The first was a complex, real-world problem from the field of open-pit mining called the constrained pit-limit (CPIT) problem and the second was a well-known problem from Karp's original 21 NP-Complete problems, the Steiner tree problem in graphs (STPG). These two problems were chosen because they are very dissimilar and require very different approaches to solve them, which allowed the versatility and flexibility of this hybrid meta-heuristic framework to be showcased.

As it is a much simpler problem, the experiments on the STPG were used to investigate some of the main properties and design choices of merge search. In contrast, the experiments on the CPIT problem highlighted the effectiveness of merge search in solving large scale, highly constrained, problems through its automatic decomposition aspects. They also investigated the effect that additional arbitrary splitting of the merge partitions has on the quality of the solutions produced and also the inclusion of a solution polishing phase afterwards.

While the datasets used for the STPG experiments already had been solved to optimality, and therefore were only being used to demonstrate the versatility of merge search and investigate its various aspects; the experiments on the CPIT

problem produced solutions which improved on the upper-bounds of all six of the problem instances, as published on the *minelib* website, and two of the most recent published upper-bounds as reported in Jélvez et al. (2019).

Although demonstrating that merge search can be applied to two very different problems shows that the algorithm is very versatile, making the claim that it is a general framework that can be applied to *any* combinatorial optimisation problem would require significantly more evidence than only two problems. Therefore, the most obvious future research that can be performed is in applying the technique to many more (and varying types of) problems. Aside from this, some further work could be done into developing a truly generic algorithm which operates entirely on the decision variables, and is completely problem agnostic; the interaction of different heuristics and other techniques, within the framework; the effectiveness of merge search with other exact solvers (such as constraint programming); the limits at which merge search is most effective; and, applying merge search to different classes (possibly non-discrete) of problems.

References

Angelelli, E., Mansini, R., & Speranza, M. G. (2010). Kernel search: A general heuristic for the multi-dimensional knapsack problem. *Computers & Operations Research*, *37*, 2017–2026.

Angelelli, E., Mansini, R., & Speranza, M. G. (2012). Kernel search: A new heuristic framework for portfolio selection. *Computational Optimization and Applications*, *51*, 345–361.

Applegate, D., Bixby, R., Cook, W., & Chvátal, V. (1998). On the solution of traveling salesman problems, .

Bienstock, D., & Zuckerberg, M. (2010). Solving LP pelaxations of large-scale precedence constrained problems. In F. Eisenbrand, & F. B. Shepherd (Eds.), *Integer Programming and Combinatorial Optimization* number 6080 in Lecture Notes in Computer Science. Springer Berlin Heidelberg.

Blum, C., Pinacho, P., López-Ibáñez, M., & Lozano, J. A. (2016). Construct, merge, solve & adapt a new general algorithm for combinatorial optimization. *Computers & Operations Research*, *68*, 75 – 88.

Blum, C., Puchinger, J., Raidl, G. R., & Roli, A. (2011). Hybrid metaheuristics in combinatorial optimization: A survey. *Applied Soft Computing*, *11*, 4135–4151.

Blum, C., Roli, A., & Sampels, M. (2008). *Hybrid Metaheuristics: An Emerging Approach to Optimization*. Studies in Computational Intelligence. Springer Berlin Heidelberg. URL: `https://books.google.com.au/books?id=4s9sCQAAQBAJ`.

Boschetti, M. A., Maniezzo, V., Roffilli, M., & Röhler, A. B. (2009). Matheuristics: Optimization, simulation and control. In *International Workshop on Hybrid Metaheuristics* (pp. 171–177). Springer.

Boyd, S., & Vandenberghe, L. (2004). *Convex optimization*. Cambridge university press.

Brazil, M., Graham, R. L., Thomas, D. A., & Zachariasen, M. (2014). On the history of the euclidean steiner tree problem. *Archive for history of exact sciences*, *68*, 327–354.

Byrka, J., Grandoni, F., Rothvoß, T., & Sanità, L. (2010). An improved lp-based approximation for steiner tree. In *Proceedings of the forty-second ACM symposium on Theory of computing* (pp. 583–592). ACM.

Chakrabarty, D., Könemann, J., & Pritchard, D. (2010). Hypergraphic lp relaxations for steiner trees. In *International Conference on Integer Programming and Combinatorial Optimization* (pp. 383–396). Springer.

Chicoisne, R., Espinoza, D., Goycoolea, M., Moreno, E., & Rubio, E. (2012). A new algorithm for the open-pit mine production scheduling problem. *Operations Research*, *60*, 517–528.

Cho, J.-D. (2001). Steiner tree problems in vlsi layout designs. In *Steiner Trees in Industry* (pp. 101–173). Springer.

Consoli, S., Moreno-Pérez, J. A., Darby-Dowman, K., & Mladenović, N. (2010). Discrete particle swarm optimization for the minimum labelling steiner tree problem. *Natural Computing*, *9*, 29–46.

Dantzig, G. B., & Wolfe, P. (1960). Decomposition principle for linear programs. *Operations research*, *8*, 101–111.

Deb, K. (2012). *Optimization for engineering design: Algorithms and examples*. PHI Learning Pvt. Ltd.

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische mathematik*, *1*, 269–271.

Dowsland, K. A. (1991). Hill-climbing, simulated annealing and the steiner problem in graphs. *Engineering Optimization*, *17*, 91–107.

Du, D.-Z., Smith, J., & Rubinstein, J. H. (2013). *Advances in Steiner trees* volume 6. Springer Science & Business Media.

Duin, C. (2000). Preprocessing the steiner problem in graphs. In *Advances in Steiner Trees* (pp. 175–233). Springer.

Eisenbrand, F., Grandoni, F., Rothvoß, T., & Schäfer, G. (2010). Connected facility location via random facility sampling and core detouring. *Journal of Computer and System Sciences*, *76*, 709–726.

Espinoza, D., Goycoolea, M., Moreno, E., & Newman, A. N. (2012). Minelib: A library of open-pit mining problems. *Annals of Operations Research*, *206(1)*, 91–114. URL: `http://mansci-web.uai.cl/minelib/`.

Feo, T. A., & Resende, M. G. (1995). Greedy randomized adaptive search procedures. *Journal of global optimization*, *6*, 109–133.

Fischetti, M., & Fischetti, M. (2018). Matheuristics. In *Handbook of Heuristics* (pp. 121–153). Springer.

Foulds, L. R., & Graham, R. L. (1982). The steiner problem in phylogeny is np-complete. *Advances in Applied mathematics*, *3*, 43–49.

Glover, F., Laguna, M., & Martí, R. (2000). Fundamentals of scatter search and path relinking. *Control and cybernetics*, *29*, 653–684.

Glover, F., Laguna, M., & Martí, R. (2003). Scatter search. In *Advances in evolutionary computing* (pp. 519–537). Springer.

Goemans, M. X., & Myung, Y.-S. (1993). A catalog of steiner tree formulations. *Networks*, *23*, 19–28.

Hochbaum, D. S., & Chen, A. (2000). Performance analysis and best implementations of old and new algorithms for the open-pit mining problem. *Operations Research*, *48*, 894–914.

Hwang, F. K., & Richards, D. S. (1992). Steiner tree problems. *Networks*, *22*, 55–89.

Jélvez, E., Morales, N., & Nancel-Penard, P. (2019). Open-pit mine production scheduling: Improvements to minelib library problems. In *Proceedings of the 27th International Symposium on Mine Planning and Equipment Selection-MPES 2018* (pp. 223–232). Springer.

Jélvez, E., Morales, N., Nancel-Penard, P., Peypouquet, J., & Reyes, P. (2016). Aggregation heuristic for the open-pit block scheduling problem. *European Journal of Operational Research*, *249*, 1169–1177.

Karp, R. M. (1972). Reducibility among combinatorial problems. In *Complexity of computer computations* (pp. 85–103). Springer.

Kennedy, J. (2010). Particle swarm optimization. *Encyclopedia of machine learning*, (pp. 760–766).

Kenny, A., Li, X., & Ernst, A. T. (2018). A merge search algorithm and its application to the constrained pit problem in mining. In *Proceedings of the Genetic and Evolutionary Computation Conference* GECCO '18. ACM.

Kenny, A., Li, X., Ernst, A. T., & Sun, Y. (2019). An improved merge search algorithm for the constrained pit problem in open-pit mining. In *Proceedings of the Genetic and Evolutionary Computation Conference* GECCO '19. ACM.

Kenny, A., Li, X., Ernst, A. T., & Thiruvady, D. (2017). Towards solving large-scale precedence constrained production scheduling problems in mining. In *Proceedings of the Genetic and Evolutionary Computation Conference* GECCO '17 (pp. 1137–1144). ACM.

Kenny, A., Li, X., Qin, A. K., & Ernst, A. T. (2016). A population-based local search technique with random descent and jump for the steiner tree problem in graphs. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016* (pp. 333–340).

Kingston, J. H., & Sheppard, N. P. (2003). On reductions for the steiner problem in graphs. *Journal of Discrete Algorithms*, *1*, 77–88.

Koch, T., Martin, A., & Voß, S. (2001). Steinlib: An updated library on steiner tree problems in graphs. In *Steiner trees in industry* (pp. 285–325). Springer.

Levitin, A. (2012). *Introduction to the Design and Analysis of Algorithms (3rd Ed.)*. Pearson Education.

Luke, S. (2009). Essentials of metaheuristics, .

Maniezzo, V., Stützle, T., & Voß, S. (2009). *Matheuristics: Hybridizing Metaheuristics and Mathematical Programming*. Annals of Information Systems. Springer US. URL: `https://books.google.com.au/books?id=3uN95LwxRlAC`.

Martins, S. (1999). Greedy randomized adaptive search procedures for the steiner problem in graphs sl martins, pm pardalos, mgc resende. and cc riheiro. In *Randomization Methods in Algorithm Design: DIMACS Workshop, December 12-14, 1997* (p. 133). American Mathematical Soc. volume 43.

Meagher, C., Dimitrakopoulos, R., & Avis, D. (2014). Optimized open pit mine design, pushbacks and the gap problem–a review. *Journal of Mining Science*, *50*, 508–526.

Muñoz Martínez, G. I. (2012). Modelos de optimización lineal entera y aplicaciones a la minería, .

Omidvar, M. N., Li, X., Mei, Y., & Yao, X. (2013). Cooperative co-evolution with differential grouping for large scale optimization. *IEEE Transactions on evolutionary computation*, *18*, 378–393.

Papadimitrou, C. H., & Steiglitz, K. (1982). Combinatorial optimization: algorithms and complexity, .

Pisinger, D., & Ropke, S. (2010). Large neighborhood search. (pp. 399–419). doi:`10.1007/978-1-4419-1665-5_13`.

Polzin, T. (2003). Algorithms for the steiner problem in networks, .

Polzin, T., & Vahdati, S. (2000). Primal-dual approaches to the steiner problem. In *International Workshop on Approximation Algorithms for Combinatorial Optimization* (pp. 214–225). Springer.

Prömel, H. J., & Steger, A. (2012). *The Steiner tree problem: a tour through graphs, algorithms, and complexity*. Springer Science & Business Media.

Qu, R., Xu, Y., Castro, J. P., & Landa-Silva, D. (2013). Particle swarm optimization for the steiner tree in graph and delay-constrained multicast routing problems. *Journal of Heuristics*, *19*, 317–342.

Raidl, G. R. (2015). Decomposition based hybrid metaheuristics. *European journal of operational research*, *244*, 66–76.

Ribeiro, C. C., Maniezzo, V., Stützle, T., Blum, C., Juan, A. A., Ramalhinho, H., Mladenovic, N., Sifaleras, A., Sörensen, K., & Souza, M. (2020). Preface to the special issue on matheuristics and metaheuristics. *International Transactions in Operational Research*, *27*, 5–8.

Samavati, M., Essam, D., Nehring, M., & Sarker, R. (2017). A methodology for the large-scale multi-period precedence-constrained knapsack problem: an application in the mining industry. *International Journal of Production Economics*, *193*, 12–20.

Samavati, M., Essam, D., Nehring, M., & Sarker, R. (2018). A new methodology for the open-pit mine production scheduling problem. *Omega*, *81*, 169–182.

Singh, G., Sier, D., Ernst, A. T., Gavriliouk, O., Oyston, R., Giles, T., & Welgama, P. (2012). A mixed integer programming model for long term capacity expansion planning: A case study from the hunter valley coal chain. *European Journal of Operational Research*, *220*, 210–224.

Skorin-Kapov, N., & Kos, M. (2006). A grasp heuristic for the delay-constrained multicast routing problem. *Telecommunication Systems*, *32*, 55–69.

Talbi, E.-G. (2002). A taxonomy of hybrid metaheuristics. *Journal of heuristics*, *8*, 541–564.

Thiruvady, D., Blum, C., & Ernst, A. T. (2020). Solution merging in matheuristics for resource constrained job scheduling. *Algorithms*, *13*. URL: `https://www.mdpi.com/1999-4893/13/10/256`. doi:`10.3390/a13100256`.

Uchoa, E., Poggi de Aragão, M., & Ribeiro, C. C. (2002). Preprocessing steiner problems from vlsi layout. *Networks: An International Journal*, *40*, 38–50.

Vahdati Daneshmand, S. (2004). *Algorithmic approaches to the Steiner problem in networks*. Ph.D. thesis Universität Mannheim.

Wolsey, L. A., & Nemhauser, G. L. (1999). *Integer and combinatorial optimization* volume 55. John Wiley & Sons.

Appendix A. CPIT Formulations

Espinoza formulation

Espinoza et al. describe a mathematical programming formulation for the CPIT problem Espinoza et al. (2012) which maximises the discounted profit summed over all blocks $b \in \mathcal{B}$ and time periods $t \in \mathcal{T}$, multiplied by the value of the decision variables x_{bt}. In their formulation, the variable x_{bt} is equal to 1 if block b is excavated at time t and 0 otherwise.

The data for the CPIT problem consists of the following sets and parameters:

- $\mathcal{B}$ the set of blocks;
- $\mathcal{T}$ the set of time periods;
- $\mathcal{R}$ the set of resources. Typically, $|\mathcal{R}| = 2$;
- $\mathcal{P}$ the set of precedences. $a \to b$ if $(a, b) \in \mathcal{P}$ means block a must be mined before block b;
- $\hat{p}_{bt}$ the discounted profit for mining block b at time t (can be negative if it is a cost only). This is simply $\frac{p_b}{(1+\alpha)^t}$ for some base cost p_b and a discount value typically 0.1;
- q_{br} the amount of resources required to extract block b;
- $\overline{\mathcal{R}}_{rt}$ the maximum amount of resource r available in time period t; and
- $\underline{\mathcal{R}}_{rt}$ the minimum amount of resource r that must be consumed in time period t (typically 0).

The variables used to solve this problem are:

- x_{bt} is a matrix of binary variables in $\mathbb{Z}_2^{|\mathcal{B}||\mathcal{T}|}$ that are 1 if block b is mined in period t and 0 otherwise.

Using this notation, the problem is written as:

$$
\begin{aligned}
\text{maximise} \quad & \sum_{b\in\mathcal{B}} \sum_{t\in\mathcal{T}} \hat{p}_{bt} x_{bt}, && && \text{(A.1)}\\
\text{subject to} \quad & \sum_{\tau\le t} x_{b\tau} \le \sum_{\tau\le t} x_{a\tau} && \forall\ (a,b) \in \mathcal{P}, t \in \mathcal{T},\\
& \sum_{t\in\mathcal{T}} x_{bt} \le 1 && \forall\ b \in \mathcal{B},\\
& \underline{\mathcal{R}}_{rt} \le \sum_{b\in\mathcal{B}} q_{br} x_{bt} \le \overline{\mathcal{R}}_{rt} && \forall\ r \in \mathcal{R}, t \in \mathcal{T},\\
& x_{bt} \in \mathbb{Z}_2 && \forall\ b \in \mathcal{B}, t \in \mathcal{T}.
\end{aligned}
$$

Merge search formulation

By aggregating the block-time decision variables into groups, the problem can be reduced significantly by modifying the Espinoza formulation given above

such that each group $G_i \in \mathcal{G}^p$ is represented by a new variable z_i:

$$\begin{aligned} \text{maximise} \quad & \sum_{G_i \in \mathcal{G}^p} p_i z_i, && && \text{(A.2)} \\ \text{subject to} \quad & z_j \leq z_i && \forall\ (i,j) \in \mathcal{P}^G, \\ & \sum_{G_i \in \mathcal{G}^p} q_{ir} z_i \leq \sum_{t \in \mathcal{T}} \overline{R}_r && \forall\ r \in \mathcal{R}, \\ & z_i \in \mathbb{Z}_2 && \forall\ G_i \in \mathcal{G}^p. \end{aligned}$$

Here, the objective value now becomes the sum of the cumulative profits for all groups that are included in a solution. Because the x_{bt} variables were cumulative, but the z_i variables are not, computing the total value of a given group or its resource usage becomes a bit complicated.

Definition 3. *Let $G_i \in \mathcal{G}^p$ be a group of cumulative x_{bt} binary decision variables where $x_{bt} = 1$ means that block b is extracted at time t or earlier. Then, $\alpha_{bi} = \min\{t \mid x_{bt} \in G_i \wedge x_{bt} = 1\}$ is the earliest time that block b appears in group G_i and $\omega_{bi} = \min\{t \mid x_{bt} \in G_i \wedge x_{bt} = 1\}$ is the latest.*

Proposition 1. *The total value that G_i contributes to the NPV of the mine is given by: $p_i = \sum_{b \in \mathcal{B}} \hat{p}_{b\alpha_{bi}} - \hat{p}_{b\omega_{bi}+1}$. If $\omega_{bi} = |\mathcal{T}|$ then $\hat{p}_{b\omega_{bi}+1} = 0$ and if $\alpha_{bi} = \emptyset$ then $\hat{p}_{b\alpha_{bi}} - \hat{p}_{b\omega_{bi}+1} = 0$.*

Because the z_i variables are not cumulative, the time-precedence constraints can be merged with the space-precedence ones.

Proposition 2. *The set of group precedences $\mathcal{P}^G \subseteq \mathcal{G}^p \times \mathcal{G}^p$ is: $\{(G_i, G_j) \mid (a,b) \in \mathcal{P}, x_{at} \in G_i, x_{bt} \in G_j, \forall t \in \mathcal{T}\} \cup \{(G_i, G_j) \mid x_{bt} \in G_i, x_{bt+1} \in G_j, \forall b \in \mathcal{B}\}$.*

The cumulative resource usage for a given group G_i can be computed in much the same way as the cumulative profit. This differs to the Espinoza formulation in that the sum of the cumulative resource usage must be less than or equal to the total resource limit over the entire life of the mine, not per period.

Proposition 3. *The total amount that G_i contributes to the resource consumption is: $q_{ir} = \sum_{b \in \mathcal{B}} q_{br} x_{b\alpha_{bi}} - q_{br} x_{b\omega_{bi}+1},\ \forall r \in \mathcal{R}$. If $\omega_{bi} = |\mathcal{T}|$ then $q_{br} x_{b\omega_{bi}+1} = 0$ and if $\alpha_{bi} = \emptyset$ then $q_{br} x_{b\alpha_{bi}} - q_{br} x_{b\omega_{bi}+1} = 0$.*

Having constructed this reduced sub-problem in terms of the new z_i group variables, it can be solved using a MIP solver and the solution is then used as the seed solution for a new population.

Appendix B. STPG Formulations

Goemans and Myung formulation

The following formulation is from Goemans and Myung Goemans & Myung (1993). This formulation was chosen for its relative simplicity and is by no means the best formulation available in the literature; however, it has already

been stated that the purpose of including the STPG in this research is not to improve the current state-of-the-art results, but to provide a test-bed upon which to investigate the proposed merge search algorithm.

$$
\begin{aligned}
\text{minimise} \quad & \sum_{(i,j)\in E} c_{ij}x_{ij}, && && \text{(B.1)}\\
\text{subject to} \quad & \sum_{k\in V} y_k - \sum_{(i,j)\in E} x_{ij} = 1, && &&\\
& \sum_{k\in S\setminus\{l\}} y_k - \sum_{(i,j)\in E(S)} x_{ij} \geq 0 && \forall\, l \in S \subseteq V, &\\
& y_k = 1 && \forall\, k \in T, &\\
& x_{ij} \in \mathbb{Z}_2,\ y_k \in \mathbb{Z}_2 && \forall\, (i,j) \in E, k \in V. &
\end{aligned}
$$

In this formulation, for ease of writing, V is the set of vertices $V(G)$, E is the set of edges $E(G)$ and $E(S) = \{(i,j) \in E \mid i,j \in S \subseteq V\}$. There are two binary decision variable vectors, $\boldsymbol{x} \in \mathbb{Z}_2^{|E|}$ and $\boldsymbol{y} \in \mathbb{Z}_2^{|V|}$, representing the set of edges and vertices, respectively.

Merge search formulation

By merging the population into groups and performing some arbitrary splitting, a reduced sub-problem can be constructed and solved with far fewer variables than the original problem instance. Before it can be solved however, the problem needs to be mapped from the original vertices and edges to a new set of variables, representing each of the groups in the merge partition.

Given a set of weighted edges E, vertices V, terminals T and merge partitions $\mathcal{P}$, the formulation for the reduced sub-problem can be constructed, based on the Goemans and Myung formulation above. This formulation uses one set of decision variables z_p, $\forall p \in \mathcal{P}$, which correspond to the merge partitions.

$$
\begin{aligned}
\text{minimise} \quad & \sum_{p\in\mathcal{P}} \left(\sum_{(i,j)\in E\cap p} c_{ij} \right) z_p, && && \text{(B.2)}\\
\text{subject to} \quad & \sum_{p\in\mathcal{P}} \left(|V\cap p| - |E\cap p|\right) z_p = 1, && &&\\
& \sum_{p\in\mathcal{P}} \left(|(S\setminus\{l\})\cap p| - |E(S)\cap p|\right) z_p \geq 0 && \forall\, l \in S \subseteq V, &\\
& z_p \leq z_q && \forall p,q \in \mathcal{P}, (i,j) \in E : p \ni (i,j), q \ni i, p \neq q, &\\
& z_p = 1 && \forall\, p \in \mathcal{P} : p \cap T \neq \emptyset, &\\
& z_p \in \mathbb{Z}_2 && \forall\, p \in \mathcal{P}. &
\end{aligned}
$$

Here, $E(S) = \{(i,j) \in E \mid i \in S \wedge j \in S\}$.

Appendix C. *steinlib* Datasets

The datasets used for the Steiner tree problem in graphs (STPG) experiments were taken from the *steinlib*Koch et al. (2001) library.

Table C.6: Characteristics of category B and C instances from the *steinlib* database. $|V|$: number of vertices; $|E|$: number of edges; $|T|$: the number of terminals; and, opt: the total cost of the optimal solution.

Category B					Category C					Category D					Category E				
Inst.	$\|V\|$	$\|E\|$	$\|T\|$	opt	Inst.	$\|V\|$	$\|E\|$	$\|T\|$	opt	Inst.	$\|V\|$	$\|E\|$	$\|T\|$	opt	Inst.	$\|V\|$	$\|E\|$	$\|T\|$	opt
*b*01	50	63	9	82	*c*01	500	625	5	85	*d*01	1,000	1,250	5	106	*e*01	2,500	3,125	5	111
*b*02	50	63	13	83	*c*02	500	625	10	144	*d*02	1,000	1,250	10	220	*e*02	2,500	3,125	10	214
*b*03	50	63	25	138	*c*03	500	625	83	754	*d*03	1,000	1,250	167	1,565	*e*03	2,500	3,125	417	4,013
*b*04	50	100	9	59	*c*04	500	625	125	1,079	*d*04	1,000	1,250	250	1,935	*e*04	2,500	3,125	625	5,101
*b*05	50	100	13	61	*c*05	500	625	250	1,579	*d*05	1,000	1,250	500	3,250	*e*05	2,500	3,125	1,250	8,128
*b*06	50	100	25	122	*c*06	500	1,000	5	55	*d*06	1,000	2,000	5	67	*e*06	2,500	5,000	5	73
*b*07	75	94	13	111	*c*07	500	1,000	10	102	*d*07	1,000	2,000	10	103	*e*07	2,500	5,000	10	145
*b*08	75	94	19	104	*c*08	500	1,000	83	509	*d*08	1,000	2,000	167	1,072	*e*08	2,500	5,000	417	2,640
*b*09	75	94	38	220	*c*09	500	1,000	125	707	*d*09	1,000	2,000	250	1,448	*e*09	2,500	5,000	625	3,604
*b*10	75	150	13	86	*c*10	500	1,000	250	1,093	*d*10	1,000	2,000	500	2,110	*e*10	2,500	5,000	1,250	5,600
*b*11	75	150	19	88	*c*11	500	2,500	5	32	*d*11	1,000	5,000	5	29	*e*11	2,500	12,500	5	34
*b*12	75	150	38	174	*c*12	500	2,500	10	46	*d*12	1,000	5,000	10	42	*e*12	2,500	12,500	10	67
*b*13	100	125	17	165	*c*13	500	2,500	83	258	*d*13	1,000	5,000	167	500	*e*13	2,500	12,500	417	1,280
*b*14	100	125	25	235	*c*14	500	2,500	125	323	*d*14	1,000	5,000	250	667	*e*14	2,500	12,500	625	1,732
*b*15	100	125	50	318	*c*15	500	2,500	250	556	*d*15	1,000	5,000	500	1,116	*e*15	2,500	12,500	1,250	2,784
*b*16	100	200	17	127	*c*16	500	12,500	5	11	*d*16	1,000	25,000	5	13	*e*16	2,500	62,500	5	15
*b*17	100	200	25	131	*c*17	500	12,500	10	18	*d*17	1,000	25,000	10	23	*e*17	2,500	62,500	10	25
*b*18	100	200	50	218	*c*18	500	12,500	83	113	*d*18	1,000	25,000	167	223	*e*18	2,500	62,500	417	564
					*c*19	500	12,500	125	146	*d*19	1,000	25,000	250	310	*e*19	2,500	62,500	625	758
					*c*20	500	12,500	250	267	*d*20	1,000	25,000	500	537	*e*20	2,500	62,500	1,250	1,342

Appendix D. STPG results

Table D.7: Results on *steinlib* dataset B, C, D and E instances. Given are the known optimal solutions and mean objective values and standard deviations for pure local search, pure MIP, GRASP heuristic and merge search heuristic. Algorithms which managed to find the optimal solution in at least one of its runs are indicated by * next to the objective value.

		pure LS		pure MIP		GRASP		merge search	
Inst.	Opt.	μ	σ	μ	σ	μ	σ	μ	σ
b01	82	**82.00***	0.00	**82.00***	0.00	83.20*	2.40	**82.00***	0.00
b02	83	**83.00***	0.00	**83.00***	0.00	**83.00***	0.00	**83.00***	0.00
b03	138	**138.00***	0.00	**138.00***	0.00	**138.00***	0.00	**138.00***	0.00
b04	59	**59.00***	0.00	**59.00***	0.00	59.80*	1.60	**59.00***	0.00
b05	61	**61.00***	0.00	**61.00***	0.00	**61.00***	0.00	**61.00***	0.00
b06	122	124.00	0.00	**122.00***	0.00	123.80*	1.60	**122.00***	0.00
b07	111	**111.00***	0.00	**111.00***	0.00	**111.00***	0.00	**111.00***	0.00
b08	104	**104.00***	0.00	**104.00***	0.00	**104.00***	0.00	**104.00***	0.00
b09	220	**220.00***	0.00	**220.00***	0.00	**220.00***	0.00	**220.00***	0.00
b10	86	**86.00***	0.00	88.85*	4.29	**86.00***	0.00	**86.00***	0.00
b11	88	**88.00***	0.00	**88.00***	0.00	89.20*	0.98	**88.00***	0.00
b12	174	**174.00***	0.00	**174.00***	0.00	**174.00***	0.00	**174.00***	0.00
b13	165	167.00*	2.45	**165.00***	0.00	168.20*	1.60	**165.00***	0.00
b14	235	236.00	0.00	235.50*	0.74	239.40	1.20	**235.00***	0.00
b15	318	**318.00***	0.00	**318.00***	0.00	322.20	1.47	**318.00***	0.00
b16	127	130.00*	2.45	134.30*	3.80	130.40*	3.38	**127.00***	0.00
b17	131	**131.00***	0.00	131.25*	0.43	131.40*	0.80	**131.00***	0.00
b18	218	**218.00***	0.00	220.05*	1.99	219.40*	1.02	**218.00***	0.00
d01	106	**106.00***	0.00	106.95*	0.22	106.20*	0.40	**106.00***	0.00
d02	220	**220.00***	0.00	237.00	0.00	226.00*	7.35	**220.00***	0.00
d03	1565	1575.20	1.60	1653.00	0.00	1611.80	9.58	**1568.27**	1.14
d04	1935	1941.00	1.26	2008.00	0.00	1976.40	5.57	**1935.36***	0.64
d05	3250	3253.80	0.75	3311.00	0.00	3293.20	7.57	**3252.50**	0.92
d06	67	69.40*	1.20	71.65	2.20	69.20*	1.83	**67.60***	1.20
d07	103	**103.00***	0.00	104.20*	0.98	**103.00***	0.00	**103.00***	0.00
d08	1072	1090.00	3.63	1147.00	0.00	1129.40	11.88	**1077.70**	2.05
d09	1448	1464.20	4.07	1543.00	0.00	1517.40	13.71	**1450.80***	1.60
d10	2110	2121.00	1.79	2161.00	0.00	2185.60	11.89	**2113.60**	1.11
d11	29	29.40*	0.49	31.60*	0.92	31.00	1.10	**29.00***	0.00
d12	42	**42.00***	0.00	**42.00***	0.00	**42.00***	0.00	**42.00***	0.00
d13	500	510.40	2.06	533.00	0.00	531.80	8.01	**504.20**	1.54
d14	667	677.80	0.75	703.00	0.00	709.40	1.36	**671.50**	0.81
d15	1116	1127.20	1.47	1150.00	0.00	1194.80	9.39	**1121.70**	1.55
d16	13	13.40*	0.49	15.25*	0.99	13.20*	0.40	**13.00***	0.00
d17	23	**23.00***	0.00	24.35*	0.91	24.20*	0.98	**23.00***	0.00
d18	223	239.00	1.41	255.00	0.00	274.40	7.63	**234.80**	0.98
d19	310	335.00	0.63	347.00	0.00	393.60	9.44	**326.10**	1.81
d20	537	543.20	0.40	545.00	0.00	646.80	8.98	**541.70**	0.78

		pure LS		pure MIP		GRASP		merge search	
Inst.	Opt.	μ	σ	μ	σ	μ	σ	μ	σ
c01	85	**85.00***	0.00	86.90*	1.41	**85.00***	0.00	**85.00***	0.00
c02	144	**144.00***	0.00	**144.00***	0.00	146.40*	2.94	**144.00***	0.00
c03	754	756.40*	1.62	780.00	0.00	767.40	4.27	**754.50***	1.12
c04	1079	1084.40	1.85	1115.00	0.00	1095.40	4.76	**1081.92**	0.95
c05	1579	1580.40	0.49	1603.00	0.00	1595.40	5.71	**1579.67***	1.11
c06	55	**55.00***	0.00	58.00*	2.26	56.40*	2.80	**55.00***	0.00
c07	102	102.40*	0.49	114.10	2.14	105.40*	1.74	**102.00***	0.00
c08	509	513.40	1.02	528.00	0.00	528.40	3.20	**511.67**	0.62
c09	707	711.00	0.89	728.00	0.00	724.80	2.14	**708.17***	0.90
c10	1093	1095.40	0.49	1127.00	0.00	1116.00	4.86	**1094.75**	1.09
c11	32	33.00*	0.63	34.90	0.44	32.40*	0.49	**32.00***	0.00
c12	46	46.20*	0.40	48.90	0.44	46.80*	1.17	**46.08***	0.28
c13	258	263.00	1.26	276.00	0.00	269.80	3.31	**260.09***	1.08
c14	323	327.40	1.36	337.00	0.00	336.40	2.94	**324.55**	0.99
c15	556	556.80*	0.40	567.00	0.00	582.40	3.83	**556.00***	0.00
c16	11	12.00	0.00	12.00	0.00	12.40	0.80	**11.00***	0.00
c17	18	19.00	0.00	19.60	0.49	19.80	0.75	**18.00***	0.00
c18	113	120.00	0.89	128.00	0.00	128.40	1.36	**117.64**	0.88
c19	146	153.00	0.89	160.00	0.00	164.40	2.94	**151.09**	0.67
c20	267	268.00	0.00	268.00	0.00	298.20	5.04	**267.91***	0.29
e01	111	**111.00***	0.00	124.70	0.90	**111.00***	0.00	**111.00***	0.00
e02	214	226.60	1.85	234.30	6.29	214.40*	0.80	**214.00***	0.00
e03	4013	4066.60	6.34	4238.00	0.00	4194.80	16.68	**4032.00**	3.41
e04	5101	5138.60	5.68	5310.00	0.00	5292.20	14.52	**5108.60**	2.87
e05	8128	8157.40	3.07	8321.00	0.00	8298.00	21.46	**8134.70**	2.65
e06	73	74.00*	2.00	82.85	2.54	75.00*	4.00	**73.00***	0.00
e07	145	148.40*	3.20	165.65	3.53	148.60*	3.88	**145.90***	1.14
e08	2640	2700.00	5.97	2818.00	0.00	2909.20	29.31	**2658.50**	2.77
e09	3604	3660.40	7.99	3795.00	0.00	3938.60	15.45	**3620.50**	3.14
e10	5600	5649.00	6.42	5760.00	0.00	5996.80	9.99	**5613.70**	2.15
e11	34	34.60*	1.20	39.00	0.00	35.40*	1.74	**34.00***	0.00
e12	67	68.40	0.49	71.00	0.00	68.60*	1.85	**67.40***	0.80
e13	1280	1329.60	5.95	1382.00	0.00	1508.60	20.26	**1302.80**	3.06
e14	1732	1775.00	2.00	1828.00	0.00	2059.00	30.13	**1747.10**	2.26
e15	2784	2824.20	1.94	2867.00	0.00	3351.20	104.52	**2800.00**	1.90
e16	15	16.20	0.40	19.00	0.00	16.80*	1.47	**15.00***	0.00
e17	25	25.80*	0.40	28.00	0.00	27.60*	2.42	**25.00***	0.00
e18	564	626.60	2.06	651.00	0.00	821.00	13.13	**600.70**	2.28
e19	758	803.00	2.83	814.00	0.00	1108.00	20.01	**785.20**	1.47
e20	1342	1356.20	0.40	1358.00	0.00	1969.20	25.67	**1353.40**	0.66

Table D.8: Results on *steinlib* dataset problem instances, investigating the effect that population size has on the number of partitions in the reduced sub-problem. Given are the size of the instance ($|V| + |E|$) and average number of partitions and standard deviation over 30 runs for population sizes 50, 100, 1,000 and 10,000.

		50		100		1,500		10,000	
Instance	size	μ	σ	μ	σ	μ	σ	μ	σ
b01	57	12.40	2.06	15.60	0.80	19.60	0.49	20.00	0.00
b02	75	16.60	2.15	18.20	1.47	22.00	0.00	22.00	0.00
b03	100	21.40	2.58	27.80	1.94	35.80	0.40	35.40	0.49
b04	132	22.40	3.26	34.40	5.68	49.20	0.98	50.80	0.40
b05	134	27.40	2.24	35.20	3.06	47.80	0.75	48.40	0.49
b06	143	48.00	3.16	67.20	2.93	94.00	0.89	94.00	1.26
b07	93	13.60	1.74	18.60	2.94	29.40	0.49	30.00	0.00
b08	106	18.40	5.35	24.20	2.14	32.80	0.40	33.00	0.00
b09	139	24.20	3.76	30.00	0.89	50.80	0.75	53.20	0.40
b10	194	32.00	4.94	44.20	3.37	67.20	2.14	70.00	1.26
b11	209	37.00	3.41	50.20	3.49	76.60	1.96	76.20	3.97
b12	221	43.40	5.16	74.40	4.59	124.00	0.63	126.20	0.40
b13	122	31.20	2.64	36.60	3.26	50.60	1.96	51.00	2.53
b14	148	43.80	2.79	56.40	3.50	76.20	0.40	77.00	0.00
b15	183	42.40	2.58	63.20	2.93	83.40	0.49	84.00	0.00
b16	261	43.20	2.56	61.20	6.65	109.40	6.37	117.40	7.81
b17	257	37.80	2.93	54.40	3.26	101.80	0.40	105.00	0.63
b18	282	54.60	7.20	82.20	5.98	155.60	2.42	163.60	1.20
d01	799	20.20	4.96	29.60	0.80	30.60	0.80	30.20	0.40
d02	837	29.00	4.29	47.40	6.83	89.40	2.58	99.00	3.90
d03	1263	71.60	3.67	122.20	9.06	464.40	11.74	604.00	7.48
d04	1415	77.00	2.53	129.60	6.95	493.40	12.03	624.40	2.42
d05	1821	66.25	3.96	111.50	11.19	527.00	9.19	728.50	2.50
d06	2504	42.00	4.06	54.75	4.82	79.40	4.63	81.00	6.82
d07	2504	25.75	6.61	43.25	7.79	72.20	3.31	78.25	4.82
d08	2703	95.50	2.18	193.00	4.18	1103.00	17.41	1661.25	7.95
d09	2735	101.75	2.68	186.75	5.72	1099.80	22.81	1703.00	7.87
d10	2877	85.75	8.07	160.75	12.68	1118.20	12.70	1798.75	5.40
d11	5667	33.00	5.34	38.50	10.11	46.60	3.93	50.25	3.63
d12	5671	46.25	1.09	70.00	4.06	123.40	2.06	125.25	0.43
d13	5637	99.75	6.72	184.50	9.60	1407.60	37.45	2542.25	18.74
d14	5658	93.75	3.63	183.25	4.60	1426.80	17.59	2696.75	22.00
d15	5640	89.75	5.07	172.00	13.64	1515.20	20.62	3040.00	35.34
d16	11595	41.50	5.68	42.25	3.56	55.60	10.61	69.75	6.80
d17	11542	59.25	2.38	91.25	8.23	160.00	7.24	167.50	6.10
d18	11312	103.75	5.12	200.25	2.49	1967.60	27.25	4149.00	23.95
d19	11242	100.50	2.50	207.25	11.76	2174.80	21.25	4712.00	12.43
d20	11471	98.00	1.58	201.50	2.96	2241.00	10.60	5009.00	27.50

		50		100		1,500		10,000	
Instance	size	μ	σ	μ	σ	μ	σ	μ	σ
c01	421	19.80	2.23	19.20	1.72	28.00	0.63	28.80	0.40
c02	399	47.20	5.91	58.60	2.87	77.40	0.49	79.00	0.00
c03	649	67.20	5.42	119.60	3.88	295.80	3.43	327.80	1.60
c04	733	61.80	2.79	109.00	15.24	304.60	6.86	340.80	4.71
c05	912	55.80	6.01	98.20	8.63	312.80	4.71	357.80	1.17
c06	1220	27.80	2.48	32.40	1.36	37.00	2.83	34.80	3.43
c07	1259	39.80	3.49	50.20	5.42	70.40	2.15	68.40	1.85
c08	1332	77.60	3.72	141.20	7.30	605.40	8.55	763.80	7.08
c09	1388	80.00	5.40	172.20	6.37	714.20	14.36	895.60	8.33
c10	1453	78.20	12.43	144.60	7.34	738.40	17.42	964.60	5.16
c11	2683	34.00	1.67	43.00	5.25	60.60	7.36	55.80	1.60
c12	2738	47.40	7.14	74.40	7.39	145.60	4.18	146.60	3.56
c13	2708	89.40	4.13	186.00	13.22	882.00	12.62	1247.40	6.02
c14	2695	96.60	7.68	176.20	8.33	912.80	14.76	1377.00	7.92
c15	2674	89.20	4.45	170.60	4.41	1078.00	15.71	1668.80	14.65
c16	5240	32.60	1.74	36.40	2.50	41.00	0.00	41.00	0.00
c17	5204	53.00	7.92	87.60	2.87	152.60	6.28	160.60	5.85
c18	5281	103.60	5.85	198.80	7.70	1217.80	6.52	1937.80	13.54
c19	5229	101.60	3.72	190.40	4.03	1333.20	16.68	2246.60	12.29
c20	5270	97.60	2.42	188.40	6.15	1545.80	24.11	2860.80	27.35
e01	1972	27.75	7.46	32.75	2.05	37.00	3.41	35.75	2.49
e02	2069	52.00	4.74	67.50	3.50	115.40	6.62	116.00	9.35
e03	3183	81.25	4.32	157.00	5.70	924.00	16.52	1476.00	5.43
e04	3573	77.75	2.17	156.00	9.25	809.20	16.14	1408.00	11.05
e05	4567	76.00	3.67	146.75	10.21	942.20	25.13	1680.75	10.59
e06	6165	28.50	4.15	40.00	4.53	47.60	12.03	49.00	9.51
e07	6277	56.50	7.09	86.50	3.84	231.00	25.79	245.75	15.07
e08	6652	102.50	6.10	211.50	14.84	1831.20	20.74	3777.25	42.86
e09	6899	101.75	0.83	197.50	2.69	1858.20	39.74	3955.25	43.68
e10	7164	91.00	5.79	178.00	8.63	1763.60	38.72	3946.75	18.67
e11	14591	35.25	4.82	38.00	7.91	56.00	10.37	56.75	12.40
e12	14623	54.25	3.11	88.75	4.66	198.80	17.27	216.50	17.17
e13	14572	97.50	4.03	206.25	7.05	2389.60	64.36	5746.50	24.67
e14	14617	96.00	2.55	191.25	7.50	2281.80	22.48	5835.25	45.78
e15	14538	96.75	8.04	188.75	8.01	2181.40	44.44	5897.25	40.26
e16	31832	41.50	7.57	56.25	6.83	87.00	13.49	68.25	11.23
e17	31590	65.00	2.00	108.75	6.57	215.80	15.95	215.50	11.50
e18	31570	104.00	4.95	204.00	5.92	2905.20	29.19	9203.50	61.98
e19	31781	99.50	1.12	199.00	7.58	2947.00	20.78	9437.50	49.14
e20	31593	97.75	6.02	190.00	5.10	2951.40	18.10	9532.50	19.93