Mixed Integer Programming Based Merge Search for Open Pit Block Scheduling

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

[a]*School of Mathematics, Monash University, Clayton, Victoria, Australia.*
[b]*School of Information Technology, Deakin University, Geelong VIC, Australia.*
[c]*BHP Perth*
[d]*School of Information Technology and Electrical Engineering, The University of Queensland, St Lucia QLD 4072, Australia.*
[e]*School of Science, RMIT University, Melbourne, Victoria, Australia*

Abstract

Open-pit mine scheduling is a challenging optimisation problem in the mining industry. It tries to create the best possible open-cut mine plan in order to maximise the net present value of an ore body. This leads to very large mixed integer programming problems that have a strong network structure which can be exploited to obtain a solution to the linear programming relaxation by repeatedly solving maximum flow problems. As these problems are too large and challenging to solve exactly, we have developed an efficient parallel optimisation method to search for good heuristic solutions. The novel matheuristic proposed in this paper, called *Merge Search*, is able to combine a very large pool of solutions via variable aggregation, thereby using their best components to find higher quality solutions. The approach is built around a mixed integer programming formulation, where the formulation is made efficient via preprocessing. A key aspect of this study is to investigate an efficient parallelisation of Merge Search through distributed computing. We demonstrate empirically that this is the best performing method for the mine scheduling problem, finding better quality solutions for a number of problem instances available in the literature. The parallelisation also substantially improves the convergence characteristics of the method, even providing drastic improvements at the beginning of the search. Furthermore, we investigate the efficacy a parallel Branch & Bound search. While the problems are too hard to be solved exactly, this improves upon the best known upper (relaxed) bounds for all problem instances.

1. Introduction

angus: intro is too "mining problem heavy"..
some things to cover:

- large scale problems can be very complicated
- decomposition algorithms important
- local vs global search trade off (hybrid techniques)
- general technique would be nice as most are domain knowledge dependent

then after, introduce problems and say that they will demonstrate the versatility of the technique

In the mining industry in Australia, open-pit mining is a problem of significant interest [Newman et al., 2010]. The planning and production scheduling associated with mines can lead to significant cost savings and profits for the operators of the mines. In particular, determining the value of a mine and the areas of excavation in the shortest possible time-frame can lead to very large gains [Meagher et al., 2014].

Extracting and processing the ore from the pits is the main focus of open pit mining. Specifically, the order in which materials are extracted and processed can lead to significant profits and savings [Meagher et al., 2014]. In determining the order, there are several constraints that must be satisfied, including precedences between blocks

and resource limits. Hence, the open-pit block scheduling problem is referred to as the *precedence constrained production scheduling problem* (PCPSP) [Bley et al., 2010, Espinoza et al., 2012].

The PCPSP requires identifying a schedule to extract blocks in a pit that maximises the net present value (NPV) of the blocks over time. Each block is associated with a positive value (profit) or a negative value (cost) and if mined, it is either processed or discarded (specified as destinations). The mining of blocks is subject to resource and precedence constraints. Resource limits apply to the amount of ore mined and processed during a period, while precedences between blocks exist due to pit slope constraints. That is, in order to reach certain blocks, other blocks on top of them need to be extracted and processed or discarded first.

The PCPSP can be formulated as a Mixed Integer Program (MIP). For real world instances, solving the MIP or even the linear programming (LP) relaxation of the problem is challenging due to the large number of variables and constraints. For example, the problem instances currently available in the literature can be very large with nearly 100,000 blocks and over a million precedence constraints. These need to be replicated over multiple time periods resulting in MIP formulations with an excess of 10 million constraints.

Finding ways to solve large MIPs with reasonable computational resources has been given attention recently. The methods include decomposition approaches, hierarchical methods and MIP-based large neighbourhood search. Among MIP-based decompositions, Lagrangian relaxation [Fisher, 2004], column generation [Wolsey, 1998] and Benders' decomposition [Geoffrion, 1972] have been widely applied and proved very effective. Other approaches for efficiently solving MIPs is to identify aggregations of variables so that a problem of reduced size can be solved [Boland et al., 2009, Litvinchev and Tsurkov, 2003, Rogers et al., 1991]. Another class of methods are based on a large neighbourhood search (LNS) [Ahuja et al., 2002]. The main idea underlying these methods is to start with a feasible solution, identify a neighbourhood (possibly a very large), and find efficient ways to search the neighbourhood. When LNS is combined with MIPs, the resulting methods have proved very effective [Ahuja et al., 2002, Pisinger and Ropke, 2010].

The main contribution of this study is a novel matheuristic algorithm, called *Merge Search*. It combines concepts from LNS, genetic algorithms [Mitchell, 1996] and the efficiency of solving MIPs. However, it is substantially different to any of these or other existing methods in two main aspects. First, the neighbourhood leading to the restricted MIP is obtained from an aggregation of variables in original model using a population of solutions (potentially very large populations). Using different input parameters (e.g. population size), the solving time of the restricted MIP can be systematically controlled. Second, Merge Search is particularly designed for parallel or distributed computing, thus allowing additional computing resources to be used effectively in tackling these challenging problems. Hence, a second contribution of this study is develop and investigate parallel implementations of Merge Search via the Message Passing Interface (MPI) [Gropp et al., 1994]. A third contribution of this study is a Branch & Bound method built around the LP relaxation of the problem. This is intended to show that even though the LP relaxation of the large MIPs can be solved relatively efficiently, the associated problems cannot be solved to optimality by a standard branch and bound approach. Despite this, we find that this method leads to identifying the best known upper bounds for benchmark instances of open pit mining.

Recently, a method which uses a population of solutions to identify a search space for MIP model is construct, solve, merge and adapt (CMSA) [Blum et al., 2016, Blum and Blesa, 2016, Blum, 2016, Lewis et al., 2019, Thiruvady et al., 2019]. Blum et al. [2016] show that CMSA can be effectively applied to the minimum common string partition and minimum covering arborescence problems. Blum and Blesa [2016] investigate CMSA for the repetition-free longest common subsequence problem with very good results can be obtained with this method and Blum [2016] show the same outcomes of CMSA for the unbalanced common string partition problem. Lewis et al. [2019] apply CMSA to happy colourings and show that this approach can find good solutions on hard problem instances where exact approaches struggle. Thiruvady et al. [2019] develop a parallel hyrbid of CMSA and ACO and show that it is effective on project scheduling with the aim of maximising the NPV.

Merge Search was developed independently but uses similar ideas as that of CMSA at a high level. The crucial differences are in its ability to deal with extremely large populations of solutions and parallelisation. Like CMSA, the search iterative builds improving solutions to a problem by using the following steps: (a) maintain a population of feasible solutions, initially found through a heuristic (b) determine active variables in the MIP model from the population of solutions, (c) solve the resulting restricted MIP, thereby merging solutions, (d) use the solution

information from the MIP to update the population, and (e) continue this process until some termination criteria is satisfied. Merge Search can be thought of as a generic matheuristic, combining integer programming and heuristic search, but in this paper we only focus on its application to the PCPSP (Precedence Constrained Pit Scheduling Problem).

This document is organised as follows. Section 8 discusses the details of the problem. This includes MIP formulations considered in this study and equivalences to others published in the literature. Next we introduce our new Merge Search heuristic and show how this can be applied to our problem in Section 2. The details of how to apply this method to the PCPSP are described in Section 10, including preprocessing, parallelisation and other implementational details that are important in obtaining good performance. Finally, we provide detailed computational results in Section 11 and show that we are able to produce both better feasible solutions and tighter bounds for most of the benchmarks that are available in the literature.

2. Merge Search

For extremely large problems such as open pit mining, a method that is able to efficiently combine solutions (potentially obtained from different sources) can be beneficial. This is particularly important in a parallel or distributed framework where multiple threads or processes are independently finding improved solutions. In such cases we do not want to just take the best solution, thereby discarding any improvements made by all other processes, but to learn from each of them. For this purpose, the key procedure proposed in this paper and used throughout the distributed solver method is the *Merge Search*. The aim of this is to combine the best features of multiple solutions that may have been arrived at independently, often by starting from the previously best known solution. Merge Search achieves this by solving an integer program over the space of combinations of solutions. That is it breaks all of the solutions into fragments (a set of variables) and re-assembles a new solution from these fragments.

Algorithm 1 Merge Search Matheuristic

Require: A combinatorial optimisation problem with variables x: $\max f(x) : x \in \mathcal{F}$
Require: Initial solution $x^0 \in \mathcal{F}$
1: **for** $k = 1, 2, \ldots$ **do**
2: **for** $j = 1, 2, \ldots, m$ **do**
3: Generate a solution s^j in a large neighbourhood of 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:

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

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**

2.1. Description

Merge Search is a general matheuristic that operates as outlined in Algorithm 1. It carries out the following steps:

Finding an initial solution

In order to produce a population of solutions, an initial solution x^0 must first be produced. Some problems, such as the PCPSP considered here, are so large and complex, that even producing a feasible solution is quite computationally expensive — let alone one that is of guaranteed good quality. In contrast, other problems are structured such that producing a reasonable quality solution is not computationally expensive at all; however, often

the methods of producing solutions to these problems are deterministic, creating issues with solution diversity. 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 and Steiglitz, 1982].

When considering Merge Search as a general framework for solving constrained optimisation problems, the method by which the initial solution is produced is not important. The ideal circumstance would be when there already exists a custom heuristic for constructing a solution to the problem. 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 required. Equation 1 gives a method of finding a feasible solution for problems which can be formulated as a mixed integer program (MIP) with n variables and m linear constraints.

$$\begin{aligned} \text{minimise} \quad & \boldsymbol{c}^T\boldsymbol{x} + C^{max}\sum_{v_i \in \boldsymbol{v}} v_i \\ \text{subject to} \quad & \boldsymbol{A}\boldsymbol{x} \geq \boldsymbol{b} - \boldsymbol{M}\boldsymbol{v} \\ & \boldsymbol{x} \in \mathbb{Z}_2^n, \quad \boldsymbol{v} \in \mathbb{Z}_2^m. \end{aligned} \tag{1}$$

Here, $\boldsymbol{v} \in \mathbb{Z}_2^m$ is a vector and $\boldsymbol{M}$ is a constant sufficiently big enough such that $v_i = 1$ if constraint i is violated and $C^{max} = \sum_{c_i \in \boldsymbol{c}} |c_i|$ is a penalising constant that is added to the objective value every time a constraint is violated. The value of C^{max} is calculated by summing the magnitude of every value in the $\boldsymbol{c}$ vector, which means that it can be guaranteed that even the worst feasible solution will have a better objective value than the best infeasible one.

By solving this MIP model, a feasible solution to the problem is produced, which can then be used as an initial solution to generate a population[1].

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. Merge search generates its population by defining a local-search operator for the problem and sampling the neighbourhood of a given solution; because it is generally easier to produce a new solution from an existing one, than it is to generate a solution from scratch.

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.

A generic neighbourhood search method can be defined for MIPs by fixing all but u variables (where u is a given fraction of the full set of decision variables) to the value of a given solution and solving this subproblem over the

[1] Actually, for sufficiently large C^{max}, solving (1) is equivalent to solving the original problem (and will produce the same optimal solution) so, theoretically, there is no need to run the algorithm twice. In practice however, solving it in this way would be intractable for any problem that is large enough to be of interest, and so would not be a practical way of finding a feasible, initial solution. Any binary $\boldsymbol{x}$ can be used to produce a feasible initial solution for (1) by simply setting $v_i = 1$ iff the corresponding ith original constraint of $\boldsymbol{A}\boldsymbol{x} \geq \boldsymbol{b}$ is violated.

remaining u variables. Alternatively, a neighbourhood search can be performed generically using the machinery of Merge Search itself: define a random partition of variables, split that partition randomly and solve the subproblem induced by that partition. The number of neighbouring solutions m is an algorithm parameter.

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 subproblem 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 0 or 1.

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 subproblem 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 $s_i^0 = s \setminus s_i$.*

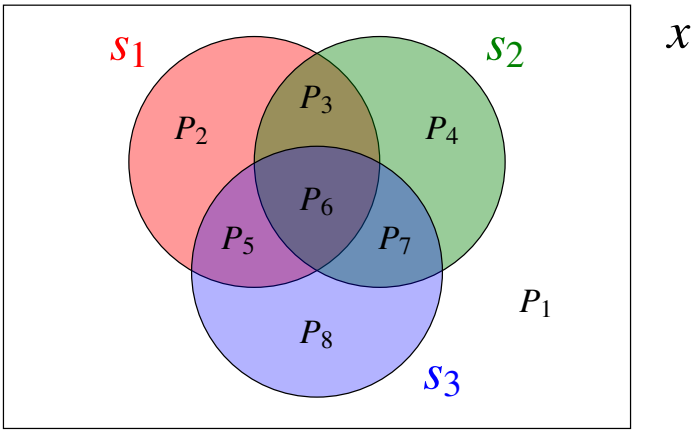


Figure 1: Partitions formed by intersecting solutions.

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 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 subproblem, with each partition being considered as an individual variable.

The theoretical maximum number of partitions (and therefore, decision variables in the reduced subproblem) 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 subproblem, 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).

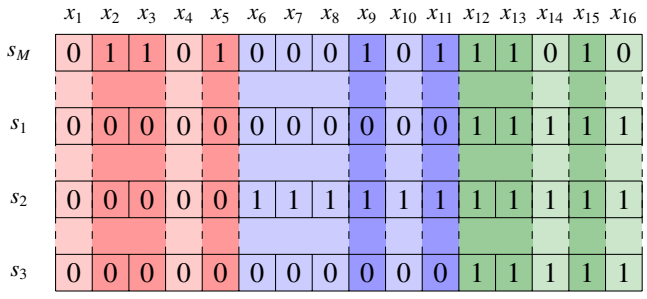


Figure 2: By adding the random bit string s_M to the population before merging, each partition is arbitrarily split into two.

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 introduced; the simplest version of which selects a gene at random and flips its corresponding bit. This can be seen as a special case of the Merge Search heuristic, 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 and 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 subproblem, and all those variables in the m singleton partitions are free to take either 0 or 1 in the solution to the reduced subproblem. 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)

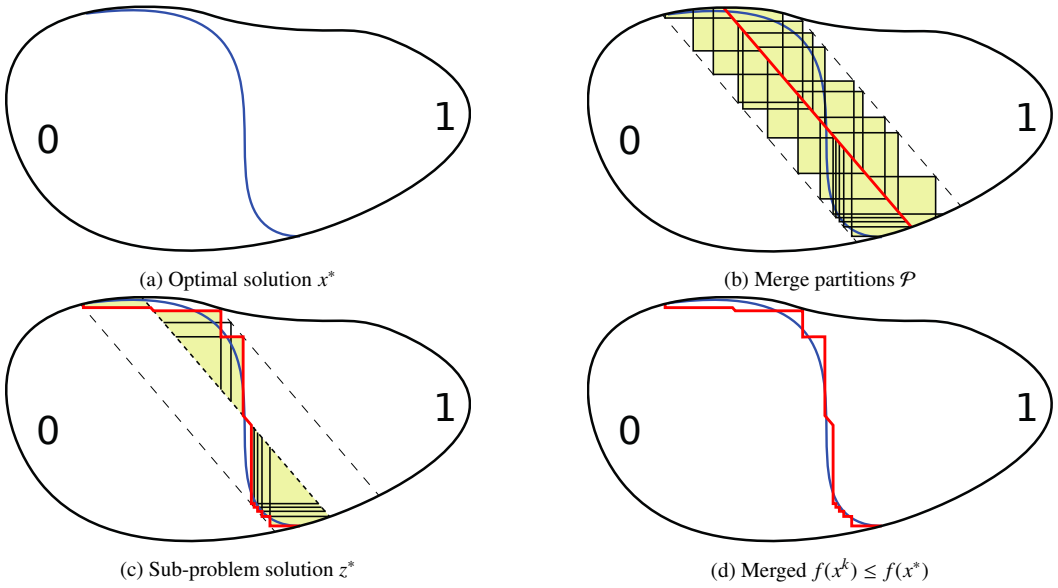
All of the steps leading up to this point have been working to construct a reduced subproblem, which can now be solved, using an exact solver or some other method, to produce a locally optimal solution for use in the next iteration of the process. In very general terms, the reduced subproblem takes the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

$$\begin{aligned} \text{minimise} \quad & f(\boldsymbol{x}) \\ \text{subject to} \quad & \boldsymbol{x} \in \mathcal{F} \subseteq \mathbb{Z}_2^{|\boldsymbol{x}|}, \\ & x_i = x_j \quad \forall i, j \in P, P \in \mathcal{P}. \end{aligned} \tag{2}$$

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 subproblem 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 subproblem will give a locally optimal solution to the master problem Figure 4 gives an illustration of this process.



(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 subproblem induced by the merge partitions $\mathcal{P}$ to optimality, gives an approximate solution to the master problem.

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 subproblem which is expressed in terms of a set of partition variables z. This reduced subproblem is then solved using an exact solver or some other method. Figure 4c shows that the optimal solution to the subproblem 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 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.

2.2. 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.

2.3. 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]. Starting with an empty subinstance of the problem C', solutions are probabilistically generated in this method, from scratch, and their components added to C'. This reduced subinstance is then solved using an exact solver and a so-called "ageing" mechanism is used to remove components from C' that have not been useful in the preceding iterations.

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 subproblem.

The CMSA subproblems 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 PCPSP 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 subproblems. 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 — as discussed previously in this section — 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 subproblem. If a variable is represented by an element of one of the candidate solutions in the population, it is automatically included in the reduced subproblem. These variables are added to the subproblem individually, and as such this aspect functions similarly to large neighbourhood search (LNS). One consequence of this is that the subproblems 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 subproblem 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 subproblems 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.

3. Applying Merge Search to solve problems

Merge Search is presented here as a general hybrid meta-heuristic framework for solving constrained combinatorial optimisation problems. In order to demonstrate its effectiveness and versatility, Merge Search is applied to two problems from two different domains: the constrained pit-limit (CPIT) problem, an abstraction of a real-life problem from open-pit mining; and the Steiner tree problem in graphs (STPG), a famous NP-Complete problem which is very well-studied in the literature.

3.1. 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 the website *minelib* [Espinoza et al., 2012] has 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.

3.1.1. Solving CPIT with Merge Search

Kenny et al. [2017] describe a novel representation for the this type of problem 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 algorithm that operates without variable partitioning or random splitting [Kenny et al., 2018]. By incorporating a variable partitioning mechanism, 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.

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.

angus: is this enough information? should i include the pseudocode or something?

3.2. The Steiner tree problem in graphs

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. Aside from being a fundamental problem in the abstract world of computing theory, the STPG has numerous real-world applications in communications, pipeline and transport network design, computational biology and very large-scale integrated circuit (VLSI) design [Cho, 2001].

Despite being around for centuries [Brazil et al., 2014], the STPG has also gained much attention in recent decades due to it being the mathematical structure behind multicast networking problems [Hwang and Richards, 1992]. Exact methods for solving the STPG have been developed using techniques such as integer linear programming (ILP), lagrangian relaxation and primal-dual strategies [Polzin and 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. When good quality, but not necessarily optimal, solutions are required in a reasonable amount of time, metaheuristics and decomposition techniques have been used to tackle this problem; some of the more successful approaches in this manner have been memetic algorithms [Klau et al., 2004], ant colony optimisation [Singh et al., 2005, Nguyen and Do, 2013], local search techniques [Uchoa and Werneck, 2010, Wade and Rayward-Smith, 2000] and voronoi-based decomposition heuristics [Leitner et al., 2014].

The STPG is extremely well-travelled, with many sophisticated pre-processing techniques and methods of finding good quality solutions already available in the literature. The purpose of including it in this paper is not to improve the current state-of-the-art results; but to provide a simple test-bed, upon which the properties of the proposed merge search algorithm can be investigated — much more easily than with a more complex, real-world problem such as the constrained pit-limit problem, mentioned above.

3.2.1. Solving the STPG with Merge Search

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.

The problem instance is first pre-processed using some of the techniques described in Duin [2000], Uchoa et al. [2002], Kingston and 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 2.

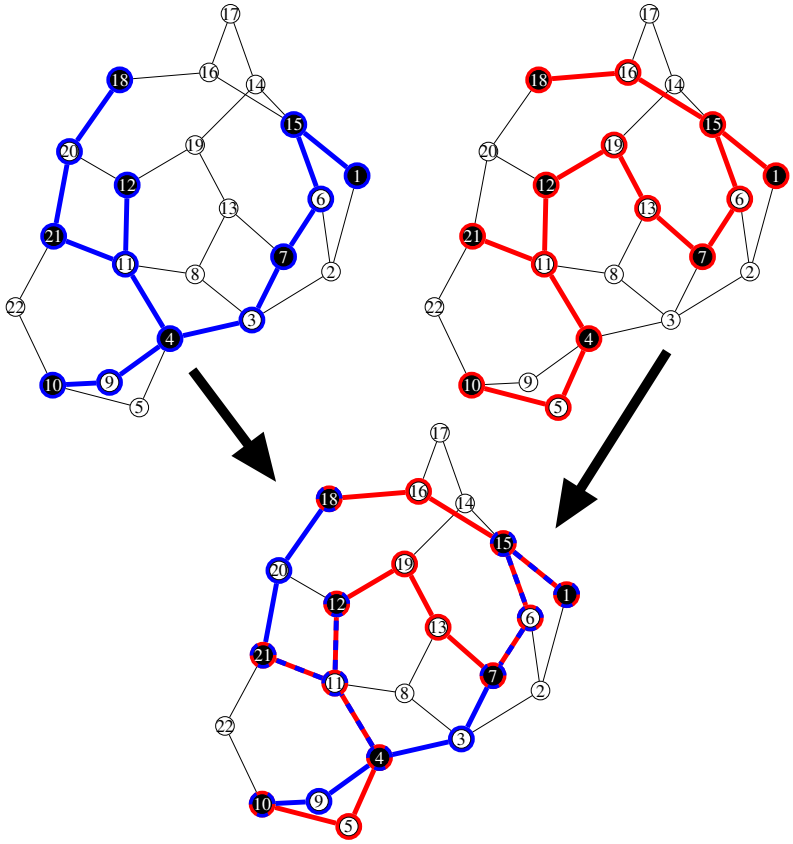


Figure 5: By merging two solutions (top left and right), the decision variables are partitioned into four disjoint sets, indicated here (bottom) by: red; blue; red and blue; and no colour.

These partitions can be split by selecting an arbitrary partition, and an arbitrary variable in that partition, and separating all variables representing vertices and edges that are connected to that arbitrary variable. 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 subproblem can be solved using a version of the Goemans and 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.

angus: should i actually include the formulation (or pseudocode) here?

4. Experiments

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

4.1. 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 2.

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.

4.2. Pre-processing

The CPIT problem

The "time-expanded" method of representing the CPIT problem given in Kenny et al. [2017] has one significant drawback: it exponentially increases the size of the problem instances. In this technique, each block-time pair is associated with a single decision variable and a solution is represented as a closure which partitions this expanded graph. In order to mitigate this effect, the structure of the problem can be exploited to apply pre-processing to reduce the number of variables and constraints in each problem instance.

First, a relaxed version of the problem, called UPIT, is solved to determine the set of blocks that are worth extracting at all; then the earliest and latest possible times for extraction for each block are computed. Because a condition of extracting a block is that all blocks above it must be extracted first, blocks that are deep cannot be extracted too early in the process — because there is not enough time to reach them — and similarly, blocks that are close to the surface cannot be extracted too late — as there will not be enough time to reach the blocks below. More detail about these pre-procesing methods can be found in Kenny et al. [2017].

Table 2: 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

The Steiner tree problem in graphs

There are many techniques available in the literature for pre-processing instances of the STPG, ranging from the very simple to the very complicated. As the experiments on the STPG for this study were not intended to improve the current state-of-the-art, but merely to investigate the properties of merge search and prove its versatility; the purpose of pre-processing the problem instances was simply to make them more manageable for the algorithms being tested. To this end, three basic ones were chosen from the literature: removing non-terminal vertices of degree 1; removing edges where a shorter path exists in the graph; and, replacing the edges incident to non-terminal vertices of degree two with a single edge. For further reading on graph reduction techniques for pre-processing instances of the STPG, see Duin [2000], Uchoa et al. [2002], Kingston and Sheppard [2003].

4.3. 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 results published on the *minelib* [Espinoza et al., 2012] website and the results from using a greedy randomised adaptive search procedure (GRASP) heuristic for the precedence constrained production scheduling problem (PCPSP) [Kenny et al., 2017], adapted for use with the CPIT problem. 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] — these results are included in order to illustrate the effect that random splitting has on solution quality.

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. Finally, the last experiment performed was to use the local improvement heuristic from the GRASP algorithm to "polish" the best result produced by the merge algorithm.

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 the appendix.

angus: (will add appendix later)

angus: should i add some details about the parameters used here?

5. Results and discussion

5.1. The CPIT problem:

Table 3 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 and the current best solution as published on the *minelib* website, ordered from smallest problem instance to largest.

In this table, it can be seen that the two merge search algorithms 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 objective value of 2.418E+07 is quite close to the optimal solution, as all three algorithms agree on this as an average value and when the search is performed without any stopping criteria, the maximum objective value that is found is 2.41798E+07.

Table 3: Results on *minelib* dataset instances. Given are the LP upper bound, current best solution from the *minelib* website, the mean objective values (μ) and standard deviations (σ) for the GRASP heuristic, merge search without random splitting and full merge search heuristic.

	LP UB	*minelib*	GRASP heuristic		merge search (no splitting)		merge search (with splitting)	
Instance			μ	σ	μ	σ		
`newman1`	2.449E+07	2.348E+07	**2.418E+07**	4.331E+03	**2.418E+07**	1.165E+02	**2.418E+07**	1.012E+03
`zuck_small`	8.542E+08	7.887E+08	8.334E+08	2.951E+06	**8.433E+08**	2.078E+06	8.432E+08	1.747E+06
`kd`	4.095E+08	3.969E+08	4.083E+08	6.135E+05	**4.088E+08**	1.881E+05	4.086E+08	4.516E+05
`zuck_medium`	7.106E+08	6.154E+08	6.548E+08	3.764E+06	6.584E+08	8.462E+05	**6.610E+08**	4.707E+06
`marvin`	8.639E+08	8.207E+08	8.368E+08	9.006E+06	8.539E+08	2.575E+05	**8.540E+08**	4.196E+05
`zuck_large`	5.739E+07	**5.678E+07**[*]	5.269E+07	1.297E+06	5.441E+07	2.558E+05	5.524E+07	5.890E+05

Comparing the results of the two variants of merge search 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

[1]This result was beaten by polishing the best merge search solution with the GRASP local improvement heuristic (Table 5).

effect of the random splitting becomes more pronounced, with the biggest effect being on the two largest instances `zuck_medium` and `zuck_large`[2].

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.

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 solution and the resultant one is much larger, merge search is demonstrably more suited to the task.

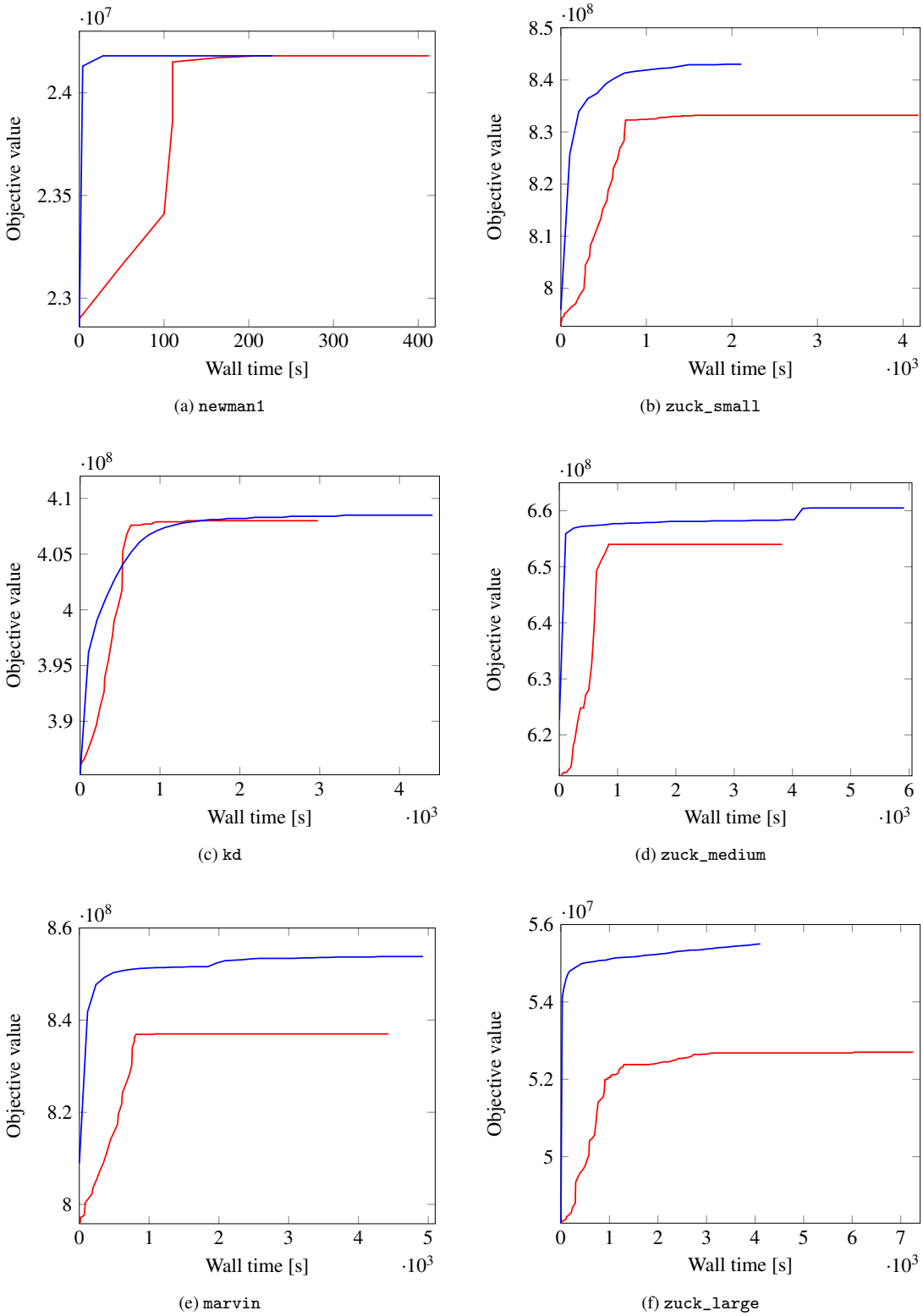
It can also be seen from these plots that once the algorithm has seemingly converged, it can sometimes find a way out of the local optima and find a much better solution. This is evidenced in the plots for `zuck_medium` (Figure 6d), `marvin` (Figure 6e) and also Figure 8a below. This behaviour is expected 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 4 gives the runtime information (wall time and CPU time) for both the GRASP and merge search algorithms. As they use a MIP solver to solve their restricted sub-problems, the runtime of both GRASP and merge search 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.

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!);

[2]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`.



(a) `newman1` (b) `zuck_small`

(c) `kd` (d) `zuck_medium`

(e) `marvin` (f) `zuck_large`

Figure 6: Convergence plots for merge search on CPIT instances. Merge search data is shown in blue and GRASP is shown in red.

Table 4: 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.

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

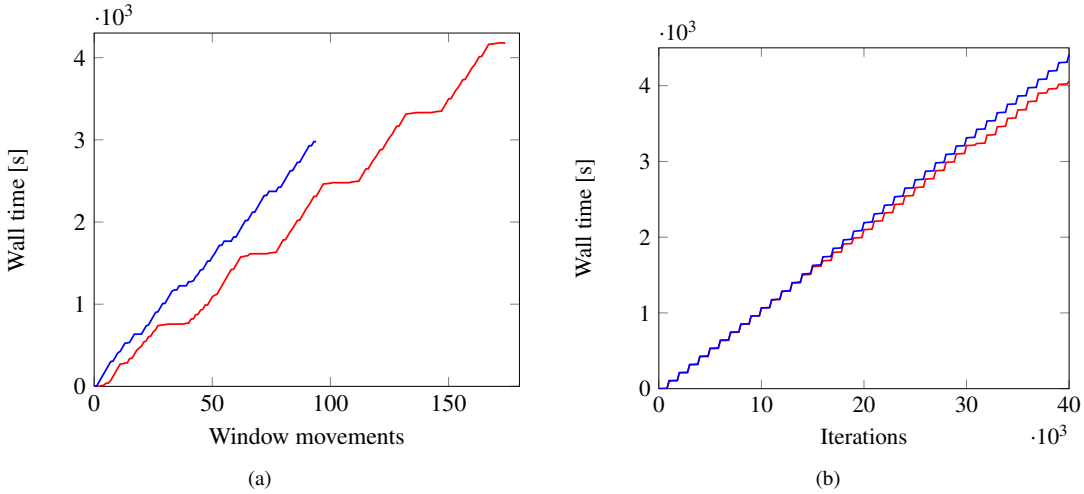


Figure 7: Plot of wall time [s] for `zuck_small` (red) and `kd` (blue) instances. Sub-figure (a) shows GRASP algorithm data and (b) shows merge search data.

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 the difference in time between GRASP and merge search is easy to explain, because GRASP is dependent on the number of periods in the solution as that determines how many window movements will be made; whereas merge search will generate the same size population and perform the same number of merges, no matter how many periods are allowed. This is demonstrated very clearly in Figure 7b, where there is very little difference between 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

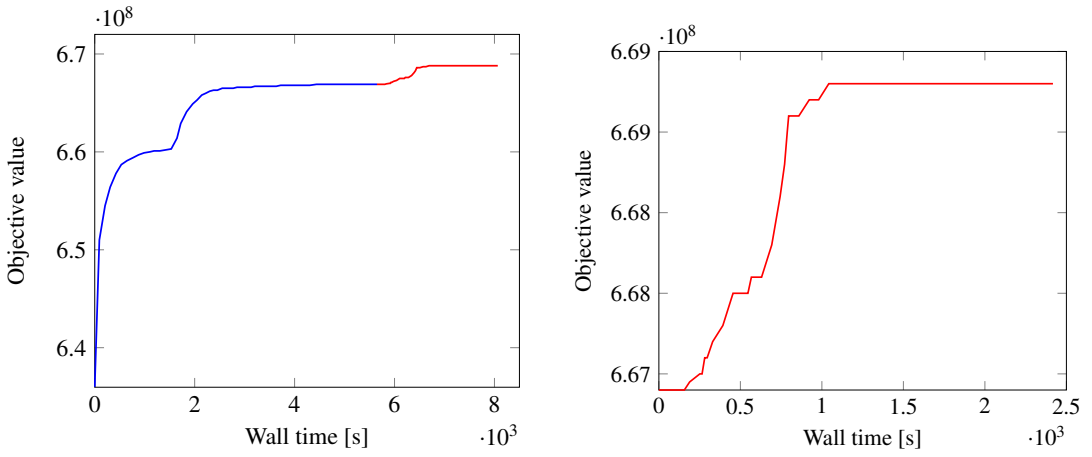
As the quality of the solutions produced by the random construction heuristic is not extremely high, too much of the local improvement phase is spent moving the times that blocks are mined over long temporal distances; so the algorithm is unable to produce high quality solutions in the allowed computational budget. It was observed however, that the quality of the solutions produced by the GRASP algorithm greatly depended on the quality of the initial solution that was constructed. So it was decided to see what would happen if the local improvement heuristic from the GRASP algorithm was applied to "polish" the best solution obtained by the merge search algorithm. These results are given in Table 5.

Table 5: Results of polishing the best merge search solution with the local improvement heursistic from the GRASP algorithm. Given is the LP upper bound and current best solution from the *minelib* website, the objective value of the best solution produced by the merge search algorithm and the value of that solution when polished by the local improvement heuristic.

Instance	LP UB	*minelib*	merge search	polished merge
`newman1`	2.449E+07	2.348E+07	**2.418E+07**	**2.418E+07**
`zuck_small`	8.542E+08	7.887E+08	8.456E+08	**8.471E+08**
`kd`	4.095E+08	3.969E+08	**4.091E+08**	**4.091E+08**
`zuck_medium`	7.106E+08	6.154E+08	6.669E+08	**6.688E+08**
`marvin`	8.639E+08	8.207E+08	8.551E+08	**8.564E+08**
`zuck_large`	5.739E+07	5.678E+07	5.625E+07	**5.730E+07**

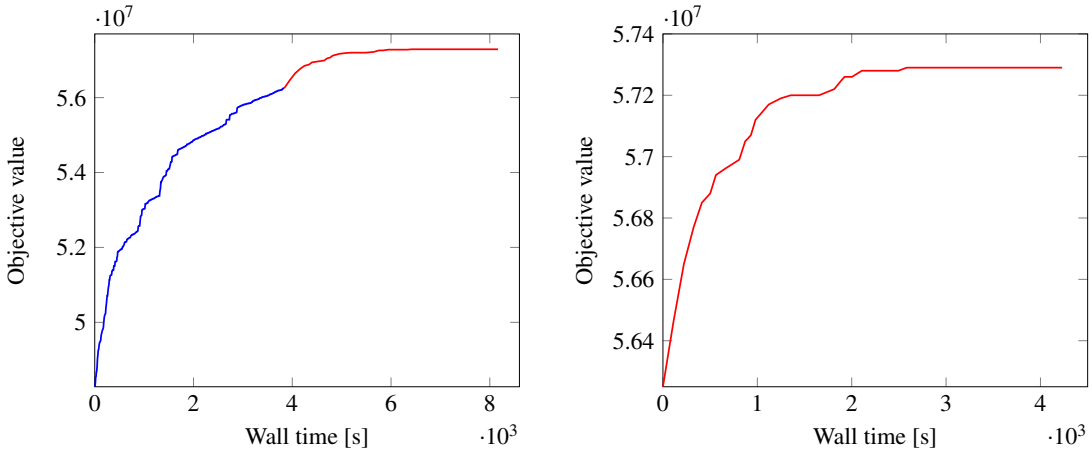
This table clearly demonstrates that the sliding window heuristic is very effective in improving solutions produced by merge search, even surpassing the *minelib* result for `zuck_large` — something that merge search could not achieve on its own, without polishing. For these experiments, the best solution 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.



(a) Merge with polishing on `zuck_medium`

(b) Detail of polishing phase



(c) Merge with polishing on `zuck_large`

(d) Detail of polishing phase

Figure 8: Plot of merge search with solution polishing on `zuck_medium` (top) and `zuck_large` (bottom) instances. The left-hand figures show the entire merge search (blue) and polishing (red) phases; the right-hand figures show details of the polishing phases.

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.

Updated state-of-the-art results

When this research was first undertaken, the current published state-of-the-art results were indeed those available on the *minelib* website. However, since then, there have been several developments as published in a recent paper by Jélvez et al. [2019] that has collated all reported improvements to the *minelib* dataset.

According to this aggregation of results, recently there are three separate studies that have improved the most on the *minelib* results for the CPIT problem instances used for this research [Samavati et al., 2017, 2018, Jélvez et al., 2016]. These improved results are given in Table 6 in terms of their LP gap, and are compared with the original *minelib* results and the polished merge results from earlier in this section.

Table 6: Recent updates to state-of-the-art results for *minelib* CPIT instances. Given is the percentage gap between the LP upper bound and the best result achieved for each respective study, compared with the original *minelib* results and the best results from this research.

				new results	
Instance	LP UB	*minelib*	polished merge	publication	gap
`newman1`	2.449E+07	4.12%	**1.26**%	Samavati et al. [2017]	**1.26**%
`zuck_small`	8.542E+08	7.67%	0.83%	Samavati et al. [2018]	**0.71**%
`kd`	4.095E+08	3.08%	**0.10**%	Samavati et al. [2018]	0.14%
`zuck_medium`	7.106E+08	13.40%	5.88%	Samavati et al. [2018]	**5.24**%
`marvin`	8.639E+08	5.00%	0.87%	Samavati et al. [2018]	**0.64**%
`zuck_large`	5.739E+07	1.05%	**0.17**%	Jélvez et al. [2016]	0.24%

Although the results from this study do not improve on every known bound in the literature, they certainly are competitive with the state-of-the-art and actually do improve the current best-known bound for the `kd` and the `zuck_large` instances.

5.2. The Steiner tree problem in graphs:

Table 7 provides the mean objective value and standard deviation for the experiments performed to compare the merge search algorithm to the three base-line algorithms of GRASP, pure local search and pure MIP. 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 an asterisk and a standard deviation of 0.00, this indicates the optimal solution was found in every run.

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[3], which just edges out pure local search and 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.

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

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

Table 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

alone, using comparable computing resources. This is evidenced by the fact that the standard deviation is 0.00 for many of the instances, suggesting that it had a hard time finding a better solution than the one 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 and 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 the hybrid meta-heuristic 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

Table 8: Representative sample of results across *steinlib* dataset instances, investigating the difference between the deterministic and random construction heuristics. Given are the average objective value and standard deviation for solutions produced by both heuristics over 30 runs and the difference between the averages, expressed as a percentage of the larger value.

	deterministic		random		
Instance	μ	σ	μ	σ	difference
b01	82.00	0.00	103.00	13.70	20.39 %
b08	104.00	0.00	141.15	20.57	26.32 %
b16	137.00	0.00	251.35	32.95	45.49 %
b18	226.00	0.00	339.90	45.29	33.51 %
c01	88.00	0.00	289.60	86.30	69.61 %
c08	528.00	0.00	971.40	193.92	45.65 %
c16	12.00	0.00	142.20	51.42	91.56 %
c20	268.00	0.00	725.65	23.83	63.07 %
d01	107.00	0.00	492.70	145.48	78.28 %
d08	1,147.00	0.00	1,929.80	302.90	40.56 %
d16	16.00	0.00	237.60	60.66	93.27 %
d20	545.00	0.00	1,486.45	119.80	63.34 %
e01	125.00	0.00	785.75	280.13	84.09 %
e08	2,818.00	0.00	4,404.75	111.45	36.02 %
e16	19.00	0.00	415.95	152.94	95.43 %
e20	1,358.00	0.00	3,969.15	502.32	65.79 %

Table 8 illustrates the difference between using the deterministic solution construction heuristic and the random solution construction heuristic. It is very clear to see that the quality of the deterministic heuristic is much better than the average random construction heuristic, especially for the larger datasets, where the difference can be up to over 95%. The full set of results are available in Table **??** of the appendix.

angus: (will add appendix later)

The results in Table 9 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 **??** of the appendix.

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

Table 9: 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

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 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 M_{factor}, which ensures that the reduced sub-problem is some fraction of the total instance size.

angus: should i go into more detail with the parameters? its already pretty long

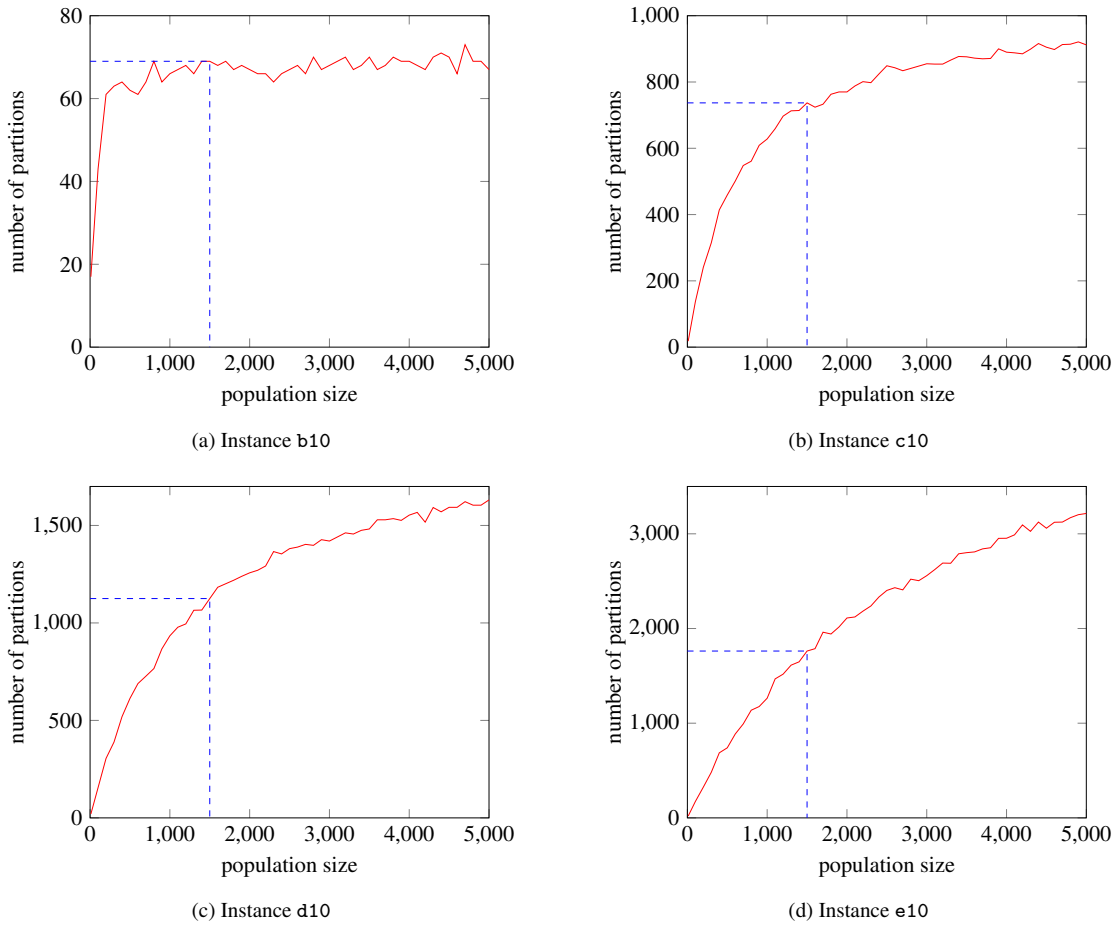
6. Conclusion and future work

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 these ($\mathcal{NP}$-Complete) problems are all theoretically reducible to one another; but in practice, they often require specific domain knowledge that means custom algorithms must be developed for each problem, individually. The discovery of a "one size fits all" solution for this whole class of problems would be a vital achievement in the field of operations research, and the purpose of this thesis was 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;



(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.

- *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 along with a generic version which, although not as efficient as methods which utilise *some* domain-specific knowledge, would serve to operate as a black-box solver for any combinatorial optimisation problem, provided a MIP model exists for it. 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 $\mathcal{NP}$-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 that need to be made when constructing merge search algorithms, specifically solution construction methods, population size, method of variable partitioning and methods of solving the reduced sub-problem. 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.

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.

7. AFTER HERE IS OLD STUFF

angus: everything below here is just from the original paper, i will cut and paste bits from it and also my thesis

8. Open Pit Mining

Reviews of the major problems related to open pit mining have been provided by [Hochbaum and Chen, 2000, Meagher et al., 2014]. The problem was originally formulated in the 1960s. However, due to the complexity of the problem and the lack of computational resources, simpler variants were initially considered. These include the ultimate pit limit (UPIT) and the constrained pit limit (CPIT) problems [Chicoisne et al., 2012, Lerchs and Grossman, 1964, Underwood and Tolwinski, 1998]. Lerchs and Grossman [1964] first introduced the UPIT problem, after which the problem has received some interest. Underwood and Tolwinski [1998] investigate this problem and propose a network flow algorithm, which is solved by mathematical programming. Chicoisne et al. [2012] tackle the CPIT problem with integer programming based decompositions and local search. They show that their method achieves good results (2-3%) for problem instances with up to five million blocks.

Due to the size of the problem instances, studies have focused on heuristics and tighter upper bounds on optimal solutions. For example, Bienstock and Zuckerberg [2010] proposed an efficient method for solving the linear programming relaxation of the problem. A standard set of test instances and best known results for both upper and lower bounds on these have been published in Minelib by Espinoza et al. [2012]. Jélvez et al. [2016] proposed an aggregation approach, where blocks are combined into larger groups of blocks along with the assumption that all blocks belonging to a larger block will all be mined in the same time-period. After running their heuristic, the larger blocks are disaggregated to generate a feasible solution to the original PCPSP.

Several integer programming approaches have been investigated for solving open pit mining problems [Caccetta and Hill, 2003, Ramazan and Dimitrakopoulos, 2004, Boland et al., 2009]. Caccetta and Hill [2003] develop a MIP approach and use branch and cut to efficiently obtain solutions to the problem, though they are unable to prove optimality even for relatively small instances. Ramazan and Dimitrakopoulos [2004] show that they were able to find good improvements computationally by considering part of the formulation as real-valued rather than using a complete binary formulation. Boland et al. [2009] investigate different MIP formulations for an open pit mining problem. The basis of their method is to aggregate blocks and iteratively disaggregate the combined blocks until the LP relaxation of the aggregated model is the same quality as the LP relaxation of the model with individual blocks. They show that with this approach they are able to solve large problem instances maximising the NPV.

Bley et al. [2010] and Kenny et al. [2017] develop alternative MIP formulations to obtain good feasible solutions more efficiently. Bley et al. [2010] tighten the formulation by adding valid inequalities obtained by combining precedence and production constraints which help in finding a feasible solution with a certain optimality gap. However, they created their own instances as the benchmark instances in MineLib were not available at the time. It is likely that this was because it is not tractable to solve the problems with CPLEX. Kenny et al. [2017] propose a GRASP-MIP based heuristic which improves upon the results (lower bounds) in MineLib for a number of the instances.

The results in MineLib come from the thesis of G. Muñoz [Martínez, 2012]. They modify the Bienstock and Zuckerberg algorithm by using a topological sort to construct feasible solutions from the relaxation. The results in MineLib and those in Kenny et al. [2017] are used for comparison in our paper, simply because we consider the same problem and the same problem instances as they do.

8.1. The Open Pit Mining Problem

The ore-body in an open pit mine is split into discrete blocks, each with an associated value. The values are determined depending on the proportion of ore in the block and the costs of extracting the ore. Typically, blocks with a higher proportion of waste will incur costs, while those with a high proportion of ore will provide a net revenue. A key decision in the problem is to decide whether or not to discard a mined block or to extract the ore from it.

In order to mine a block of high value, there may be a number of other blocks in a layer directly above it that need to be mined first. In turn, there may be other blocks in layers above which also have to be mined. This leads to precedence relationships between the blocks which are included in the problem as precedence constraints.

It is only worth considering extracting a sequence of blocks if their combined value leads to profits. Furthermore, when we consider yearly time periods in which only a limited amount of material can be extracted from the mine, we can apply a cumulative discount from the starting point of the horizon to the point at which the block is mined. This leads to the formulation of the NPV objective for the problem and for the main decision problem which can be simply stated as: which blocks should be mined and processes in each period so as to maximise the NPV?

Figure 10 shows a simple example of a two dimensional open-pit mine. There are 11 blocks. Each of these have an associated dollar value. The grey-coloured blocks are waste and incur costs. At the bottom-left and bottom-right of the mine there are two independent blocks of (high-proportion) ore. In order to get to block 9, blocks 5 and 6 need to be mined. In turn, to get to blocks 5 and 6, blocks 1 and 2 need to be mined first. The order in which the tasks must be mined leads to the precedences between them.

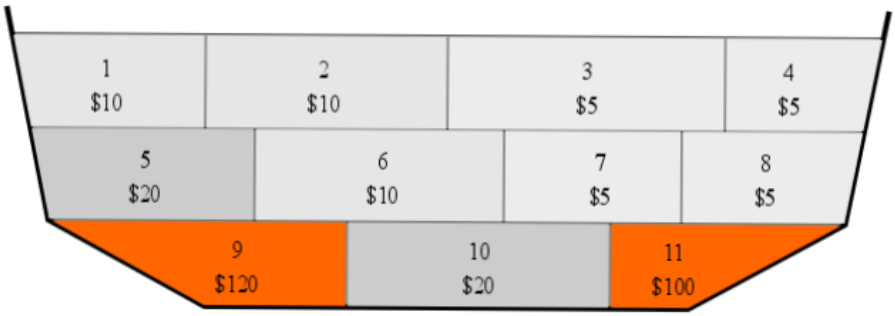


Figure 10: An example of an open-pit mine split into blocks. The blocks are numbered from 1-11 and are associated with values (dollar amounts). At the bottom-left and bottom-right are two blocks of ore which provide profits of $120 and $100, respectively. All other blocks (shades of grey) have negative dollar amounts.

The aim is to extract the ore incurring the least costs and maximising the profit. We see that in order to get to the ore in the bottom-left, a total of $50 is incurred. Whereas, getting to the ore at the bottom-right requires $20. The total profit obtained from getting to block 9 is $70 and $80 for block 11 (assuming a single time period). Hence, if there were competing resources and a single time period, it is preferable to mine the sequence of blocks (3,4,7,8) to get to the ore in block 11.

8.2. Integer Programming Formulation of Open Pit Mining

In this section we provide a MIP formulation of the PCPSP. Due to the complexity of the MIP, we also consider different algorithms to solve different aspects of the problem, e.g. the linear programming relaxation. Hence, we also consider alternative formulations that are useful in solving the problem.

The PCPSP problem is to generate the maximum profit over the life of a mine taking into account the following restrictions: (a) each block is mined at most once, (b) predecessor blocks must be mined in the same period or earlier, (c) each block must be sent (possibly fractionally) to the destinations in the period in which it is mined and (d) the resource limits consumed in sending blocks to their destinations must not satisfied.

The PCPSP problem[4] consists of the following sets and parameters:

B The set of blocks (up to about 100,000 in the data sets we are interested in).

T the set of time periods (typically 20 or 30 years).

[4]In this paper we use the benchmark data sets available in "Minelib: A library of open-pit mining problems" [Espinoza et al., 2012].

D the set of destinations. In the data there are normally just $|D| = 2$, representing whether the a block is discarded ($d = 0$) or processed and sold ($d = 1$).

R the set of resources (typically just 2, one for mining blocks and the other for processing of blocks).

$\mathcal{P}$ the set of precedences. We write $a \to b$ if $(a, b) \in \mathcal{P}$ to mean block a has to be mined before (or in the same period as) block b.

p_{bdt} the profit for sending block b to destination in d in time t (can be negative if mining the block results in a net loss). For the MineLib data this is simply $\frac{p_{bd}}{(1+\alpha)^t}$ for some base cost p_{bd} and α is a discount value that is typically a fraction between 0 and 1.

q_{bdr} the amount of resource r required by block b if sent to destination d. Note, for the datasets we are interested in this is either the tonnage of the block or zero if destination d does not consume resource r.

$\bar{R}_{rt}$ The amount of resource r available in time period t (typically a constant that only depends on r in our data sets).

Given the above definitions of sets and parameters, we now describe the mathematical formulation. We introduce the following decision variables:

$$\begin{aligned} \bar{x}_{bt} &= \begin{cases} 1 & \text{if the block has been removed by the end of period } t \\ 0 & \text{otherwise} \end{cases} \\ y_{bdt} &\in [0, 1] \text{ is the fraction of block } b \text{ sent to destination } d \text{ in period } t. \end{aligned}$$

Using this notation we can write the problem formulation as:

Problem PCPSP

$$\max \quad \sum_{b \in B} \sum_{d \in D} \sum_{t \in T} p_{bdt}\, y_{bdt} \tag{3}$$

$$\text{s.t.} \quad \bar{x}_{bt} \le \bar{x}_{at} \qquad \forall\, (a, b) \in \mathcal{P},\ t \in T \tag{4}$$

$$\bar{x}_{bt} \le \bar{x}_{b,t+1} \qquad \forall\, b \in B,\ t \in T \tag{5}$$

$$\sum_{d \in D} y_{bdt} = \bar{x}_{bt} - \bar{x}_{b,t-1} \qquad \forall\, b \in B,\ t \in T \tag{6}$$

$$\sum_{b \in B} \sum_{d \in D} q_{bdr}\, y_{bdt} \le \bar{R}_{rt} \qquad \forall\, r \in R,\ t \in T \tag{7}$$

$$\bar{x}_{bt} \in \{0, 1\},\ \ y_{bdt} \ge 0 \qquad \forall\, b \in B,\ d \in D,\ t \in T \tag{8}$$

Constraints (4) ensure that blocks are mined according to the precedence relations. The fact that a block cannot be mined more than once is represented by the constraints (5). Constraints (6) ensure that if a block is extracted then it must be sent to one or more destinations. Resource utilization requirements are presented by the constraints (7). For technical correctness note that in (6) for the first time period no previous $\bar{x}$ is to be subtracted.

The MIP formulation is equivalent to the formulation presented in MineLib, but the variable $\bar{x}_{bt}$ has different meaning in the MineLib documentation, i.e., $x_{bt} = 1$ if block b is mined at time-point t. The PCPSP problem can also be modelled with an aggregation formulation, which can be used to efficiently obtain the LP relaxation (details in Section 10.2). Alternative network flow and aggregation based formulations are detailed in Appendix A, Appendix B and Appendix C.

9. Application of Merge Search to the PCPSP

In applying Merge Search to the problem in this paper we implement it as follows. Firstly, the Merge Search method is used recursively as a neighbourhood search mechanism. When used to generate neighbourhood solutions, we simply split the binary variables x into those that are zero and those that are one in the current solution. For the continuous variables, a large number of different fractional solutions are possible, but in practice solutions tend to only have a very small number of different fractional values as observed in the discussion of the Bienstock-Zuckerberg approach (see Section 10.3).

Our implementation uses a distributed computing approach with neighbourhood search being carried out in different processes, each of which maintains its own population of solutions. The best solution found so far is shared amongst processes in an asynchronous manner allowing each process to carry out the merge operation of Step 8.

The merge step solves an aggregated version of the PCPSP. The starting point for this merge step is a slight variant of the MIP formulation (described in detail in Appendix B) which only has one variable $\bar{y}_{bdt}$ for each (block, destination, time) triplet, rather than separate $\bar{x}$ and y variables. We then define a partition as per Steps 6 and 7 of Algorithm 1 by:

1. Start with all of the (block, destination, time) triplets in one large set,
2. Split out all triplets that have been fixed to zero or to one by preprocessing into two separate sets based on their fixed values,
3. For any heuristic solution $\bar{\mathbf{y}}$, split sets in the partition so that all elements have the same value in $\bar{\mathbf{y}}$,
4. Split the partition further in a randomised manner up to a prescribed maximum number of sets.

Based on such a partition, we now set up a variant of the formulation (B.1)–(B.5) in which there are only variables $\bar{y}_S$ for any set of triplets S in the partition. This formulation really only has two types of constraints: precedence constraints $\bar{y}_S \leq \bar{y}_P$ for two different partition sets S, P where at least one pair of triplets are constrained in this way; and resource constraints that are aggregated versions of (B.4) (see Appendix C for details). As noted above, any feasible solution used as input to the splitting process will be feasible for the simplified problem generated in this way. In addition, the optimisation searches over the space of combinations of solutions allowing multiple improvements to the previously best known solution to be merged into a single, even better solution.

The randomised additional splitting (Step 7 of Algorithm 1) generates an additional expanded neighbourhood of the starting solutions in which to search. In order to ensure that this neighbourhood has a reasonable chance of containing improving solutions, this additional partitioning is guided by the global best solution as follows:

1. Sets of triplets that are split out of a partition always form "cones" - that is, they are defined in terms of a (block, destination, time) triplet plus all of the predecessors (or all of the successors) of this triplet that are part of the same set in the current partition.
2. If the current best solution has a $\bar{y}_{b,t,d} = 1$ then all of the successors are considered (as the neighbourhood move being contemplated is setting this variable and its successors to zero). Conversely if $\bar{y}_{b,t,d} = 0$ we consider the predecessor cone.
3. A similar number of cones are generated in both directions, to maximise the probability that some combination of these can be swapped.

Note that these restrictions on the types of random splits that are being considered are simply a way of ensuring that we have meaningful splits with a reasonable chance of improving the solution. For example if we carried out a split of some set $P \in \mathcal{P}$ into P_1 and P_2 in such a way that both a predecessor of some triplet in P_1 and a successor of some triplet in P_1 is in P_2 then the precedence constraints would enforce that $\bar{y}_{P_1} \geq \bar{y}_{P_2}$ and $\bar{y}_{P_1} \leq \bar{y}_{P_2}$ (i.e. the two aggregated variables must take on the same value).

In our implementation we used a "recursive" Merge Search method in which the neighbourhood search, implemented in Step 3, is simply another Merge Search with $m = 0$ (i.e. relying purely on random splitting to generate a neighbouring solution). This clearly results in a method that is purely an ascent approach. However Lemma 2 guarantees that it is at least theoretically possible to find an optimal solution. Furthermore, given the large instance size even a simple hill-climbing scheme would require a long time to reach a local optimum. The empirical evidence indicates that it is not necessary to include any more sophisticated diversification methods in order to obtain good solutions.

9.1. Merge Search Example

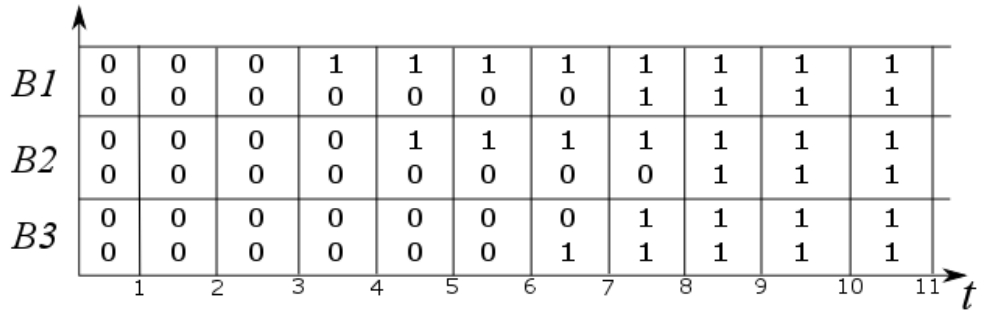


Figure 11: A problem with 3 blocks an two solutions. The mining time of each block varies depending on the solution; for example, Block 1 is mined at time-point 3 in the first solution whereas it is mined at time-point 7 in the second solution.

To make our new algorithm easier to understand, we provide a simple example of how Merge Search can be applied to open pit mining. Figure 11 shows such an example of two solutions and the time-points when the three blocks (B1, B2 and B3) are mined. Given these two solutions, the solution space for the MILP is split into four sets (Figure 12). The white region consists of variables which were 0 in both solutions; light grey, where the first solution is 1 and the second is 0; dark grey, where first solution is 0 and second is 1;and black, where variables from both solutions are 1. When solving the corresponding MILP, all variables in each set are aggregated and required to take on the same value. For example, the variables in light grey region should either be all 0s or all 1s and similarly for the remaining sets.

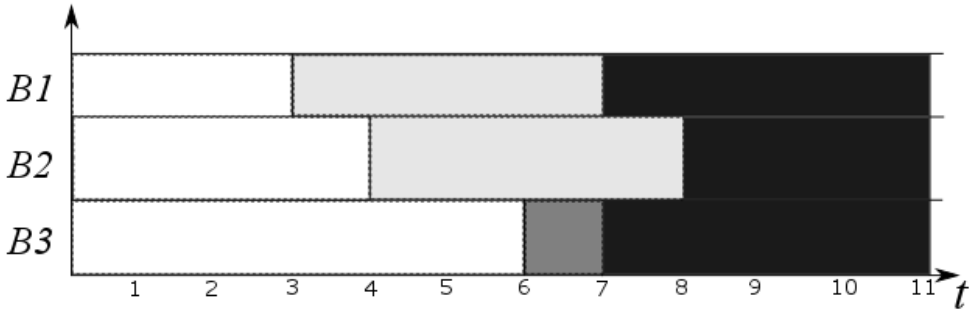


Figure 12: The sets generated from the solutions in Figure 11. The white region corresponds to both solutions having all variables 0 and the black region is one where all variables are 1. The grey areas are where the variables take on different values in both solutions; light grey is where the first solution is 1 and the second solution is 0; dark grey is the opposite, i.e., the first solution takes on value 0 and the second solution 1.

The fours sets in Figure 12 can be too small to generate new and potentially improving solutions. Hence, we apply random splitting to the regions. An example of this can be seen in Figure 13. The light grey area is split into

smaller sets allowing the possibility of generating a number of new solutions.[5] Note that in this example if we split the variables so that there exist partitions P and Q with $(B1, 4) \in P$ & $(B1, 7) \in P$ while the middle part $(B1, 5)$ & $(B1, 6)$ are in Q, then the two partitions P and Q are forced to take on the same value by the requirement that a block that has been mined stays mined ($\bar{y}_{b,d,t+1} \geq y_{b,d,t}$ in constraint (B.3)). In general, a larger number of sets allow more possibilities, leading to improved diversity. In the limit, splitting into sufficiently many subsets makes each P a singleton, so that the problem to be solved in the merge step is just the original MIP.

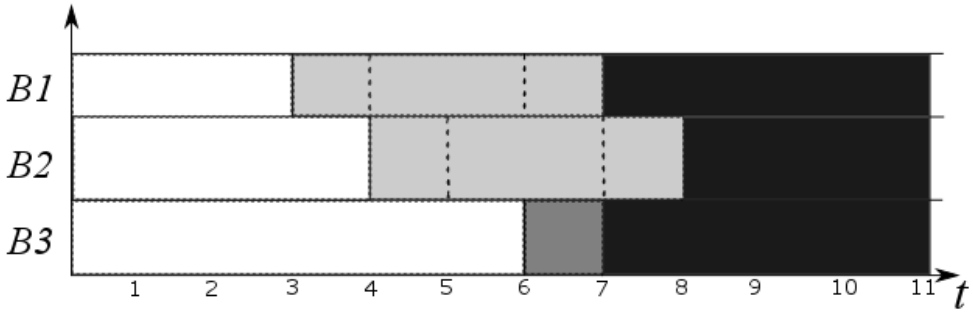


Figure 13: The sets generated from the solutions in Figure 11 and the sets are the same as that of Figure 12. The set in the light grey region is split into additional sets along the dotted lines.

10. Efficient Implementation of Merge Search and a Branch & Bound Method

As we are solving a large integer linear program it is advantageous to solve the LP relaxation of the problem. This is used in several ways: (a) to provide a relaxed bound (quality guarantee), (b) for a rounding heuristic to generate a promising initial solution, and (c) as a basis for a simple branch and bound method that could in principle produce optimal solutions. In this section, we outline some of the methods that have been used. While these are less novel than Merge Search itself, they make a significant difference to the empirical performance reported in this paper. Furthermore, some of these small algorithmic improvements have not to the best of our knowledge been described briefly in the literature. Hence, we describe these briefly for completeness. Specifically, the methods discussed are:

- Preprocessing: in order to make the solvers efficient, preprocessing is used to substantially reduce the problem size.
- Max closure solver: this algorithm is often used as a part of the other methods and we hence provide the details of its implementation, including the best approach for solving it.
- LP relaxation: due to large run-time requirements of obtaining the LP relaxation, we implement the more efficient Bienstock & Zuckerberg method [Bienstock and Zuckerberg, 2010]. It uses the max closure solver as its basis. Through this method, we obtain both relaxed bounds, and via a rounding heuristic, an initial heuristic solution.
- Branch & Bound: using the LP relaxation as a basis, we implement a Branch & Bound method.
- Distributed implementation: Allowing multiple CPUs (computers) to solve these challenging problems was a significant factor in motivating the development of Merge Search. We discuss how this has been implemented for our computational experiments.

[5] The dark grey area may not be split further as there are no integer time points between 6 and 7.

10.1. Preprocessing

In order to reduce the instance size as much as possible, some preprocessing is performed to eliminate variables that can never appear in the optimal solution. The preprocessing is carried out on all instances before any other solvers are used.

The first preprocessing method is to solve the *ultimate pit problem* (UPIT); a mine planning problem with no resource constraints. This computes all blocks that would be considered if everything could be mined in the first period, and with no restriction on the number of blocks that can be processed. Any blocks that are not included in this unrestricted version would never be profitable to be mined in the PCPSP and are thus removed.

Secondly, the number of possible destinations for some of the blocks can be reduced. Essentially, if a block contains little valuable ore, processing it will be both less profitable and more resource intensive to process the block than to discard it. In this case, the number of destinations can be reduced, again simplifying the problem and eliminating a continuous variable (for the fraction of the block to be processed).

The final preprocessing method considers, for each block, the cone of predecessors. We can then compute the minimum amount of resources required to mine both the block and all of its predecessors. From this we can calculate the earliest possible period when the block is to be mined. For some of the deeper blocks this can significantly restrict the number of time periods that have to be considered.

10.2. Maximum Closure Solver

As a sub-algorithm in many of the other methods proposed, it is necessary to solve the *Maximum Closure Problem* which is equivalent to the UPIT problem. This is defined as

$$\max \quad \sum_{v \in V} p_v x_v \tag{9}$$

$$\text{Subject to} \quad x_v \geq x_w \qquad \forall\ (v, w) \in A \tag{10}$$

$$x_v \in \{0, 1\} \qquad \forall\ v \in V \tag{11}$$

Where V is a set of vertices (in our case these correspond to block, time, destination triplets) and A a set of arcs. For the problem we are interested in, these are based on the precedence restrictions and the requirement that once a block is mined, it stays mined.

As has been noted by Dorit S. Hochbaum and Anna Chen [2000], the dual of the maximum closure problem can be modelled as a network flow or maximum flow problem allowing it to be solved comparatively easily. See Appendix A for details of this transformation. It should be noted though that the instances are *very* large with many millions of arcs for the largest data sets. We tested three different algorithms:

P: The "Push-Relabel" method for maximum flow problems. This is the method recommended in the literature (see Dorit S. Hochbaum and Anna Chen [2000]). We used a standard implementation from the Boost Graph Library[6] with a theoretical complexity of $O(n^3)$, rather than a custom implementation as discussed in the literature. This method performed worst in our experiments.

N: The Network simplex algorithm as implemented in the commercial CPLEX library. This is expected to be a slightly less efficient than the maximum flow algorithms but has the advantage that re-solving with changed costs is sometimes faster than solving the problem the first time.

B: The Boykov-Kolmogorov Algorithm [Boykov and Kolmogorov, 2001], which is related to augmenting paths, but improves efficiency by maintaining two trees that are grown from the source and sink nodes. It has a theoretical complexity of $O(mn^2|C|)$ though it has been found to perform much better in practice. The implementation from the Boost Graph Library[7] has been used.

[6] See `http://www.boost.org/doc/libs/1_63_0/libs/graph/doc/push_relabel_max_flow.html`

[7] See `http://www.boost.org/doc/libs/1_63_0/libs/graph/doc/boykov_kolmogorov_max_flow.html`

Table 10 below provides brief experimental results that indicate the relative effectiveness of these methods for solving instances of max closure problems encountered in this work. Each method was run repeatedly during the solution of the LP relaxation using the algorithm described in Section 10.3 (approximately 30 times each). See Table 11 for more information about the instances.

Table 10: CPU time (seconds) of 3 maximum closure methods with preprocessing of 30 solves (with different objective values) each on 3 data sets.

Instance	Method	fastest	average	slowest
Zuck_small	P	8.47	10.43	14.14
Zuck_small	N	0.22	6.60	37.44
Zuck_small	B	1.22	4.10	8.73
KD	P	4.30	7.97	10.19
KD	N	0.15	5.78	15.33
KD	B	0.56	5.17	13.29
Zuck_medium	P	64.27	86.70	122.04
Zuck_medium	N	1.14	70.99	334.31
Zuck_medium	B	17.61	57.50	150.20

Based on these results we note that the method recommended in the literature is the slowest. We would recommend method **B** as having the best performance on average. However, given the smaller memory footprint (observed anecdotally but not measured systematically) and the occasionally faster run times of the network flow method (N), this could also be considered. For the implementation of these methods it should be noted that:

Numerical Stability: Due to the large range of floating point numbers involved and the size of the instances, numerical stability is an issue. The use of scaling helps to alleviate this to some degree. Nevertheless, for the largest instances, the solution of the maximum closure problem may only be an approximation. This is particularly problematic for Lagrangian or dual type methods, where small changes in cost are used to try to push the solution towards feasibility (not described here). Given the large magnitude of some of the costs involved, small changes to the costs can get lost based on the numerical accuracy limits of the computer.

Preprocessing: The graph structure arising from the Bienstock-Zuckerberg formulation contains many chains of nodes (connected sets of nodes with indegree and outdegree at most 1). These particularly occur where there are multiple destinations for the same block. For any such chain of nodes there are three possible options:

1. All of the nodes are included, if the end of the chain is in the closure;
2. All of them are excluded, if the start is not in the closure; or
3. The chain an be broken into two parts A and B, with part A included in the closure and B excluded. The optimal division of the chain into parts A and B can be precomputed independently for each chain.

Hence, we can modify the problem to remove the chain and replace it with a single arc from the node at the start of the chain to the node at the end. The value of the start node is increased by the total value of nodes in A ($\sum_{v \in A} p_v$), while the value of nodes in B is added to the node at the end of the chain. All intermediate nodes can be removed. This preprocessing step has a significant positive effect on the efficiency with which the max closure subproblems can be solved.

During the solution process we are interested branching on x_v variables or fixing some of these based on preprocessing the overall problem. However, all of the solution approaches solve the linear programming dual of the formulation (9)–(11). Thus it is not possible to fix variables directly (fixed variables correspond to constraints in the dual problem). Again, we resort to a preprocessing step that removes variables in the maximum closure problem before passing it to the solver.

10.3. Solving the LP relaxation

The LP relaxation of the PCPSP is solved using the Bienstock–Zuckerberg algorithm [Bienstock and Zuckerberg, 2010] has been developed to solve the LP relaxation of large scale precedence constrained production scheduling problems. This is a type of decomposition approach with some similarity to column generation. It iterates between solving a master problem, that is essentially the LP relaxation of the merge problem (that is it is a reduced problem defined by a partition of the variables) and a subproblem that is equivalent to the Lagrangian relaxation obtained by dualizing the resource constraints (B.4). The problem that has to be solved for a given Lagrange multiplier $\mu \geq 0$, is

Problem $L(\mu)$

$$\max \quad \sum_{b\in B}\sum_{d\in D}\sum_{t\in T}\left(\bar{p}_{bdt} - \sum_{r\in R} q_{bdr}\mu_{rt} - q_{bd'r}\mu_{rt'}|_{(d',t')\to(d,t)}\right)\bar{y}_{bdt} \tag{12}$$

$$\text{s.t.} \quad \bar{y}_{b0t} \leq \bar{y}_{a0t} \qquad \forall\ (a,b)\in\mathcal{P},\ t\in T \tag{13}$$

$$\bar{y}_{bd't'} \leq \bar{y}_{bdt} \qquad \forall\ b\in B,\ d\in D,\ t\in T,\ \text{if } (d,t)\to(d',t') \tag{14}$$

$$0 \leq \bar{y}_{bdt} \leq 1 \qquad \forall\ b\in B,\ d\in D,\ t\in T \tag{15}$$

This problem is exactly the maximum closure problem. In the master problem a partition $\mathcal{S}$ of the variables is used to solve the LP relaxation of PCPSP($\mathcal{S}$), that is the aggregated problem also used during the merge step. The dual values of the resource constraints (C.3) are used to update the Lagrange multipliers μ. The subproblem solution $\bar{y}^*$ is then used to refine the partition $\mathcal{S}$ by splitting any sets that are only partly in the optimal closure defined by $\bar{y}^*$. The algorithm is presented in Algorithm 2. Note that this algorithm iterates between finding valid upper (relaxed) bounds via Lagrangian relaxation and valid LP lower bounds (feasible fractional solutions y^k) by solving aggregated LPs. The master problem LPs simply require that any pair of variables (corresponding to block-destination-time triplets) that have had the same solution value in each of the Lagrangian iterations so far, must also take the same value in the LP (See Step 7 of Algorithm 2). For more details see Bienstock and Zuckerberg [2010].

Algorithm 2 Bienstock–Zuckerberg Algorithm

1: **Initialize:** $\mu^0 = 0$, $\mathcal{S}^0 = \{\mathcal{N}\}$ ($\mathcal{N} = B\times D\times T$), $r^0 = 0$, $z^0 = -\infty$, $k = 1$.
2: **for** Iteration $k = 1, 2, \ldots$ **do**
3: **Use Max–Closure Algorithm** to obtain $\bar{y}^k$ to $L(\mu^{k-1})$
4: **Let:** $I(\bar{y}^k) = \{n\in\mathcal{N} \mid \bar{y}^k_n = 1\}$ and $O(\bar{y}^k) = \{n\in\mathcal{N} \mid y^k_n = 0\}$
5: **Let:** $\mathcal{S}^k = \left\{S\cap I(\bar{y}^k) \mid S\in\mathcal{S}^{k-1}\ \&\ S\cap I(\bar{y}^k)\neq\emptyset\right\}\cup\left\{S\cap O(\bar{y}^k) \mid S\in\mathcal{S}^{k-1}\ \&\ S\cap O(\bar{y}^k)\neq\emptyset\right\}$
6: **if** $\mathcal{S}^k = \mathcal{S}^{k-1}$ **then** STOP ▷ i.e. finish if the partition is unchanged.
7: Solve the LP relaxation of PCPSP($\mathcal{S}^k$) to get optimal solution y^k. ▷ See (C.1)–(C.4)
8: ▷ Note that $y^k_{b,d,t} = y^k_{b',d',t'}$ whenever (b,d,t) & (b',d',t') are in the same set $S\in\mathcal{S}^k$
9: **Let** μ^k be the optimal dual values of the resource constraints (C.3).
10: **if** $\mu^k = \mu^{k-1}$ **then** STOP
11: **end for**

10.3.1. Rounding Heuristic

The LP relaxation of the PCPSP provides valuable information regarding the 'preference' of when the blocks should be mined. Here, a greedy approach is implemented to construct an integer solution from a fractional solution to the LP relaxation. Each time, the block with highest preference is selected from among the available blocks. The preference relation is defined using lexicographical order of

1. Accessibility – number of remaining predecessors
2. Accumulated weight – cumulative sum of fractional values
3. Start period in fractional solution

4. Precedence relations between blocks

5. Maximum marginal profit per unit of resource

In this greedy approach, we consider periods one by one, starting from the first period, until all blocks with positive marginal profits are scheduled; or we have reached the end of the planning period. In each period, blocks are added until resources are fully utilized. For each selected block, the destinations are determined using a greedy approach based on marginal profits (as large as possible a fraction of the block sent to the destination with the highest marginal profit). Once a schedule is obtained, the distribution of the blocks in each period is refined using an LP approach to improve the total profit, if possible.

The rounding heuristic on its own does not perform very well. However, it runs reasonably quickly once the LP solution has been obtained and allows us to produce both a lower bound and an upper bound based on the Bienstock-Zuckerberg algorithm.

10.4. Branch & Bound

The LP solution obtained using the Bienstock-Zuckerberg algorithm can be used in a branch and bound method to search for an optimal integer solution. A depth-first branch and bound method has been implemented for comparison purposes, since none of the MILP formulations presented above can be used directly with a MILP solver (the instances are far too large). While theoretically this would allow the problems to be solved exactly, in practice this cannot be expected within a reasonable time limit. No attempt has been made to explore alternative branching strategies, and we leave this aspect to future work.

10.5. Parallel and Distributed Computing

Two different types of parallelisation were considered in this paper. Firstly, a multi-core shared memory architecture via OpenMP [Chapman et al., 2007] is used by any MIP based implementation, either by Gurobi or CPLEX. Either of these solvers will always use as many cores as possible from those that are made available to them.

Secondly, we use an MPI distributed framework (Gropp et al. [1994]). The implementation for this study is not a traditional master/slave framework. Rather, each node consists of a master and a slave thread. The master thread ensures the communication requests are processed in a timely manner, while the slave does most of the computational work.

Figure 14 demonstrates the architecture. Both threads maintain copies of the solution information, while the master solution (within grey region of a node) is the only one that is broadcast (grey links). It is also the only one updated with solution information from other nodes (blue links). Periodically, the solution information from the slave is updated to the master's solution (links within nodes). Conversely, when there is a new solution which is an improvement over the master's solution, the master's solution is updated and broadcast to the other nodes.

11. Computational Experiments & Results

All code was implemented in C++ with GCC-4.8.0. For the parallel implementations, Intel and GCC compilers were tested and we settled on GCC. For CPLEX or Gurobi, a maximum of 6 threads were made available as we found no improved performance with a larger number of threads. In the case of MPI runs, the number of nodes used are specified with the results.

The parallel experiments were run on a cluster comprising 35 nodes of which we made use of 12 of these nodes which had up to 228 GB resident memory.[8] To solve the MIPs, Gurobi 6.5.1[9] and CPLEX 12.6.3[10] were used.

11.1. Problem Instances

As discussed earlier, the problem instances were taken from MineLib (`mansci-web.uai.cl/minelib/`). The instances and their characteristics can be seen in Table 11. The Newman1 dataset can be solved to optimality with

[8] Each algorithm requires at least 10 GB memory.

[9] `http://www.gurobi.com/`

[10] `https://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/`

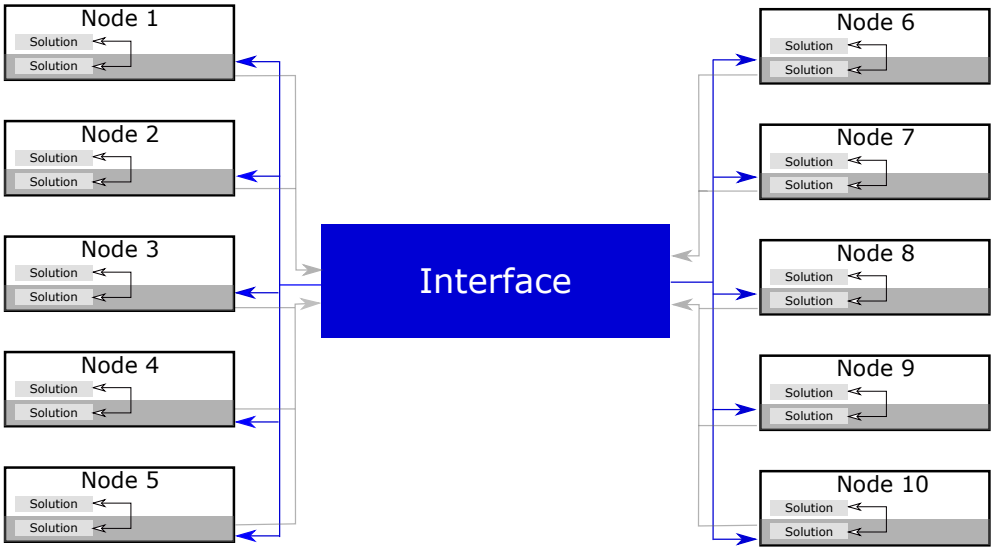


Figure 14: Communication in the MPI framework with 10 nodes. Each node consists of at least two threads, a master (grey) and a slave (white). Via the interface, solutions are broadcast from each node to every other node.

Table 11: Selected problem instances from MineLib

Instance	No. Blocks	No. Precs	No. Periods
Newman1	1,060	3,922	6
Zuck_small	9,200	145,640	20
Zuck_medium	29,277	1,271,207	15
Zuck_large	96,821	1,053,105	30
Kd	14,153	219,778	12
Marvin	53,271	650,631	20
Sm2	99,014	96,642	30

the full MIP model within 10 minutes. All the remaining instances are too large to even obtain the LP relaxation in reasonable time. Hence, the Bienstock & Zuckerberg method has been used for this purpose. Note, we do not consider three problems instances from MineLib in this study: p4hd, w23 and mclaughlin. The first two include lower bounds on the resources constraints, for which the methods proposed here cannot easily deal with. The mclaughlin instance proves too large, which proves too memory intensive for the MS-based methods.

11.2. Computational Results

In the following discussion, we have two main comparisons. The first is the lower bound, produced by all the algorithms. Second, some of the algorithms produce upper bounds (e.g. Branch & Bound), and we compare these to the results in Minelib. All the methods we compare are:

BZ: Bienstock & Zuckerberg. This solves the linear programming relaxation only and constructs a heuristic solution with the rounding heuristic described in Section 10.3.1. This is essentially a serial algorithm. It uses the maximum closure solver with the Boykov-Kolmogorov algorithm.

MS: Merge Search starts with the BZ algorithm and then applies the Merge Search Algorithm 1 on a single computer. We use $m = 100$ and $K = 10000$

B&B: Branch & Bound as described in Section 10.4

P-MS: Parallel Merge Search runs the same MS method in multiple (12) processes and periodically merges the best solutions found across all of the distributed runs.

WS: The results obtained in the "Window Search" method described in the study by Kenny et al. [2017].

MineLib: This refers to the best known results as published in Espinoza et al. [2012].

The results are presented in Table 12, which includes all the algorithms being compared and also the results from Minelib. For the algorithms proposed in this study (BZ, MS, B&B and P-MS), a maximum of five hours of wall-time was allowed. The reason for using this limit is that we are able to achieve as good if not better results in shorter run-times than previous studies ($\geq$ 10 hours, e.g. the study by Kenny et al. [2017]). In the case of P-MS, 12 nodes were allowed. The *Best LB* is the best lower bound achieved across all the algorithms including also the results from Kenny et al. [2017] and MineLib. The gaps to the best upper bound are also provided (Gap % = $\frac{(UB*-LB)\times 100}{UB*}$, where UB* is the upper bound found by B&B). Note, due to very high communication overheads, no solution could be obtained P-MS for *Newman*1.[11]

First, comparing lower bounds, we see that P-MS is consistently the best performing algorithm (finds best solutions on 5 out of 7 problem instances). MS on its own is effective, also finding best solutions for two instances (*Zuck_small* and *Marvin*). This shows that the parallel implementation provides substantial benefit when problem instances start becoming large. For *Zuck_medium*, the window search of Kenny et al. [2017] performs best and for *Zuck_large* the result from MineLib is the best. In these two cases, the number of constraints proves to complex for the MS-based algorithms (or Branch & Bound) as the short time limit proves too restrictive. Specifically, *Zuck_large* happens to be the instance with the largest number of edges in the time-expanded precedence graph and *Zuck_medium* consist of the largest number of precedences overall.

A final point about the lower bounds is that, among the methods proposed in this study (BZ, MS, B&B, P-MS), BZ performs best on *Zuck_large*. This is due to the rounding heuristic, which proves to be effective when the problem sizes are too large for the MS-based algorithms to cope with.

The second result of interest is regarding the upper bounds (comparing UB within BZ, B&B and MineLib). The BZ algorithm provides upper bounds very close to the of the original LP upper bounds, which is expected as

[11] Since *Newman*1 is relatively small (approx. 1000 blocks and 4000 precedences) and can be solved to optimality within 10 minutes by original MILP, there is no need for an efficient solution method. However, we still provide the results for this instance for the sake of completeness.

Table 12: Results for the serial and parallel runs for each algorithm with a time limit of 5 hours. The problems Zuck_small, Zuck_medium, and Zuck_large are listed as small, medium and large, respectively. The best MineLib results are also provided for comparisons. The last row specifies the best lower bound (Best LB) found across all the algorithms, including the MineLib results. Gaps (Gap %) to the upper bounds are also provided for each instance.

Algorithm		Newman1	small	medium	large	Kd	Marvin	Sm2
BZ	LB	2.37E+07	7.21E+08	5.53E+08	5.08E+07	3.73E+08	7.29E+08	1.64E+09
	Gap %	2.07	19.89	26.07	11.34	8.80	19.54	0.61
	UB	**2.42E+07**	9.05E+08	**7.48E+08**	5.79E+07	4.11E+08	9.12E+08	1.652E+09
MS	LB	**2.42E+07**	**8.95E+08**	6.81E+08	4.69E+07	4.06E+08	**9.00E+08**	1.63E+09
	Gap %	**0.00**	**0.56**	8.96	18.15	0.73	**0.66**	1.21
B&B	LB	2.42E+07	8.50E+08	6.38E+08	5.02E+07	1.82E+08	8.28E+08	1.65E+09
	Gap %	0.00	5.56	14.71	12.39	55.50	8.61	0.00
	UB	**2.42E+07**	**9.00E+08**	**7.48E+08**	**5.79E+07**	**4.09E+08**	**9.06E+08**	**1.65E+09**
P-MS	LB	-	**8.95E+08**	7.12E+08	4.06E+07	**4.07E+08**	**9.00E+08**	**1.65E+09**
	Gap %		**0.56**	**4.81**	29.14	**0.49**	**0.66**	**0.00**
	Nodes	-	12	12	12	12	12	12
WS	LB	2.41E+07	8.91E+08	**7.28E+08**	5.70E+07	4.02E+08	5.70E+07	-
MineLib	LB	2.37E+07	7.99E+08	6.76E+08	**5.73E+07**	4.07E+08	8.86E+08	**1.65E+09**
	Gap %	2.07	11.22	9.63	0.00	0.49	2.21	0.00
	UB	2.45E+07	9.06E+08	7.51E+08	5.79E+07	4.11E+08	9.12E+08	1.652E+09
Best LB		2.42E+07	8.95E+08	7.28E+08	5.73E+07	4.07E+08	9.00E+08	1.65E+09

this is what the BZ method is designed to do. However, in the case of *Zuck_medium*, the upper bound has been improved substantially, demonstrating the effectiveness of the preprocessing describe in Section 10.1. The B&B algorithm improves upon the upper bounds of BZ (except *Zuck_medium*, where the upper bounds found are the same) but cannot close the gap to the best known solutions for most instances.

11.2.1. Convergence of MS and P-MS

We examine the convergence behaviour of MS and P-MS, the two best methods for obtaining lower bounds. For this purpose, we limit both algorithms to two hours of run-time. The results are presented in Figure 15, for which we consider only the instances where MS and P-MS were most successful (*Zuck_small*, *Kd, Marvin* and *Sm2*). The result is computed as follows. For each instance at each time point, the difference to the best solution at the end of the run is computed as $\frac{LB^* - LB}{LB^*}$. Then at each time point these values are averaged. Interestingly, initially, there is already a very large difference in favour of P-MS. This shows that a large amount of diversity leads to MS improving substantially, even for a single MILP solve. Overall, the parallel implementation with the assistance of the additional nodes allows the algorithm to converge more quickly to the best solution. Even though MS catches up at some time points, P-MS usually continues its improvement over the horizon.

The next experiment aims to determine how P-MS converges with a differing number of nodes (processes run in parallel, not necessarily on different computers). The results are presented in Figure 16. Compared to Figure 15, this figure additionally includes *Zuck_medium* and we have run the algorithms for five hours. Like Figure 15, the result is computed as the difference to the best solution at the end of the run $\frac{LB^* - LB}{LB^*}$. Again, initially there is a large difference depending on the number of cores. The only real stand-out at five minutes is the 12-core run. After about 35 minutes, we see increasing improvements with an increasing number of cores. This is not surprising since increasing the cores leads to improved diversity, which in turn leads to identifying more promising areas of the search space.

11.2.2. Investigating Random Splitting

A critical component of MS is random splitting, where too few splits can lead to poor solutions and too many sets render the MILPs intractable. Hence, we investigate the 'ideal' amount of random splitting. That is, we wish to

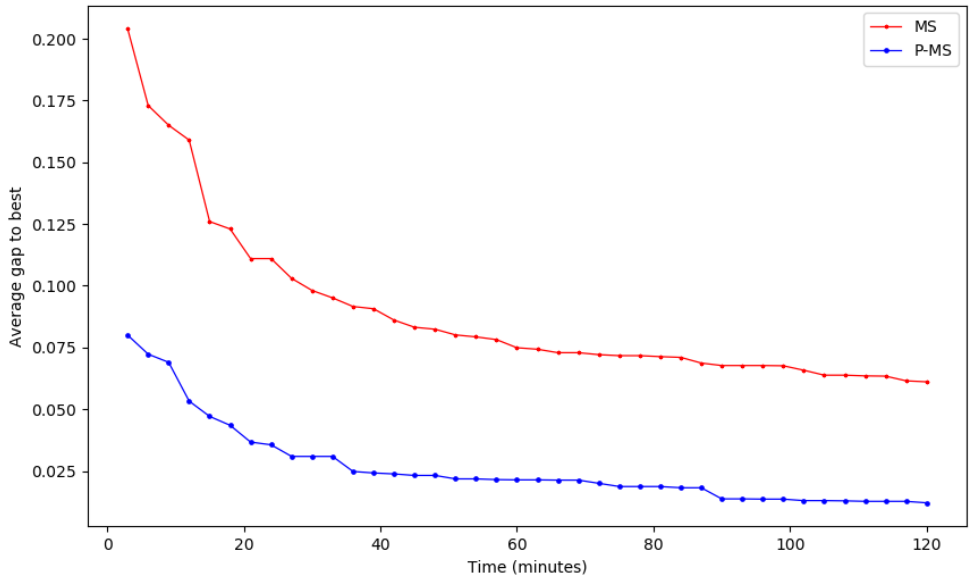


Figure 15: Convergence of MS and P-MS for the smaller instances including Zuck_small, Kd, Marvin and Sm2. The y-axis show the gap to the best solutions found across all these instances.

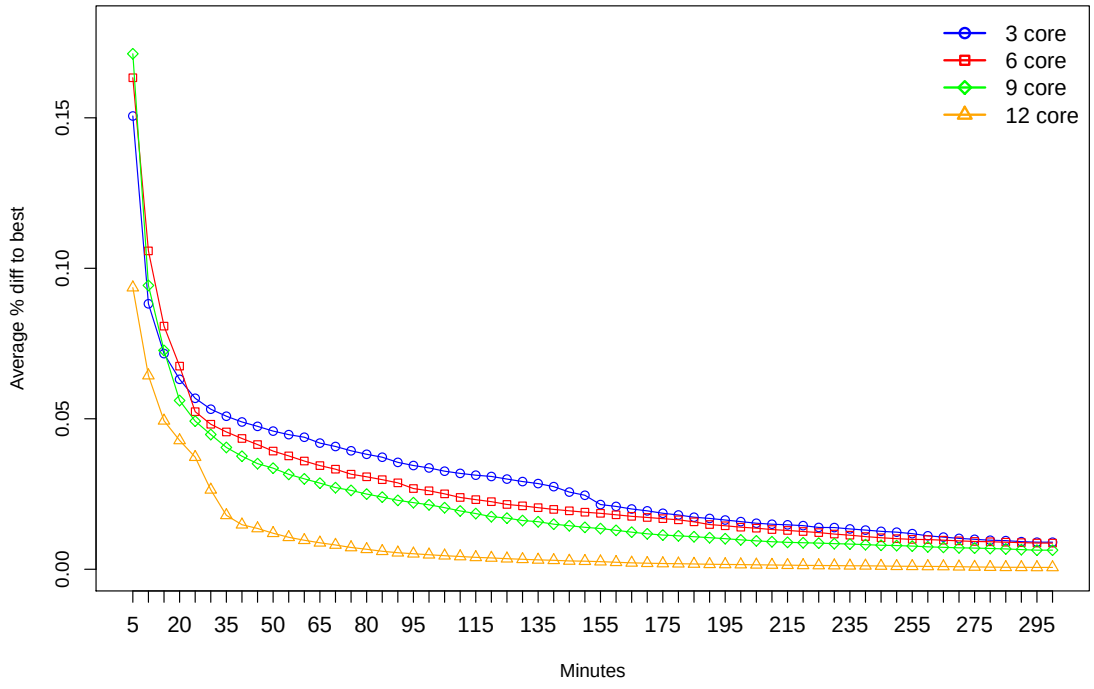


Figure 16: Convergence of P-MS by varying the number of processes (3,6,9,12) for the instances including *Zuck_small, Zuck_medium, Kd, Marvin* and *Sm2*. The y-axis show the gap to the best solution found by all runs across all these instances.

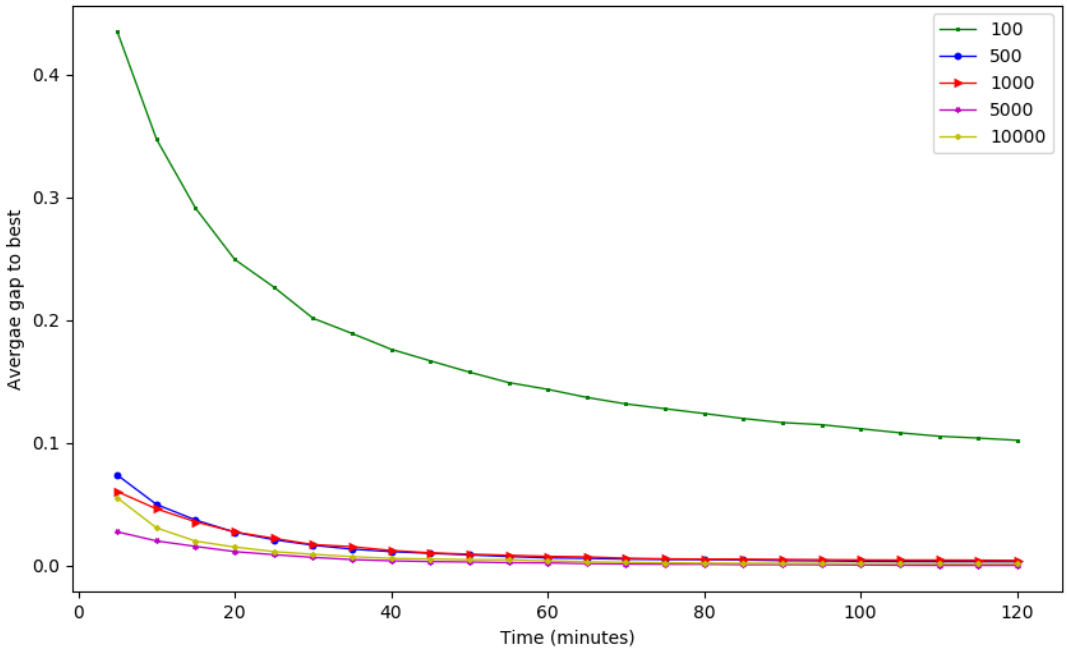


Figure 17: The performance of P-MS on the *Kd* dataset with different levels of random splitting: 100, 500, 1000, 5000, 10000. The y-axis shows the gap to the best solution found across all runs.

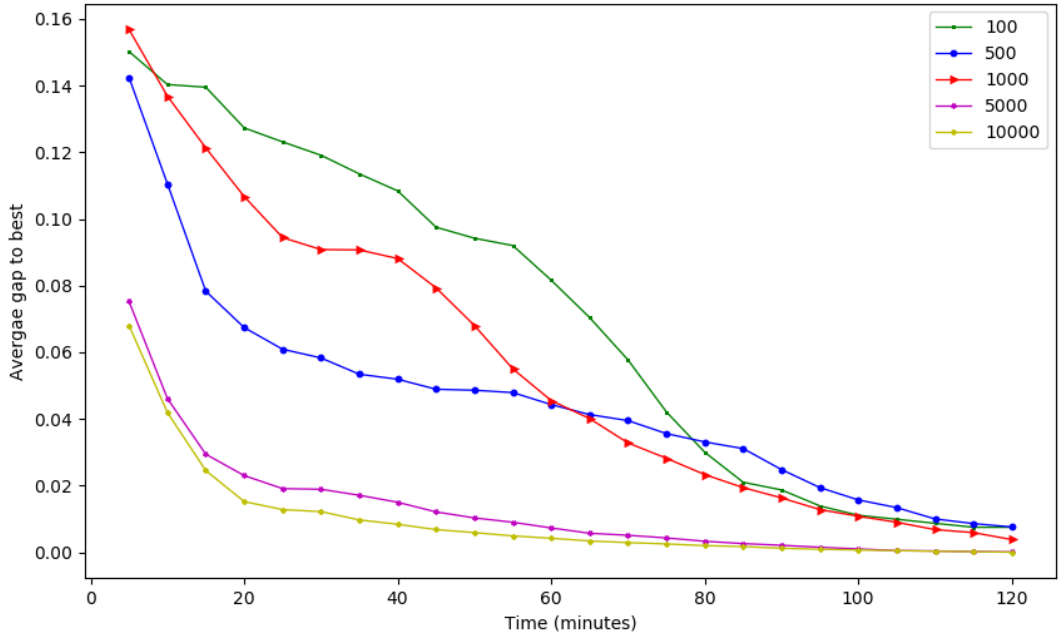


Figure 18: The performance of P-MS on the *Marvin* dataset with different levels of random splitting: 100, 500, 1000, 5000, 10000. The y-axis shows the gap to the best solution found across all runs.

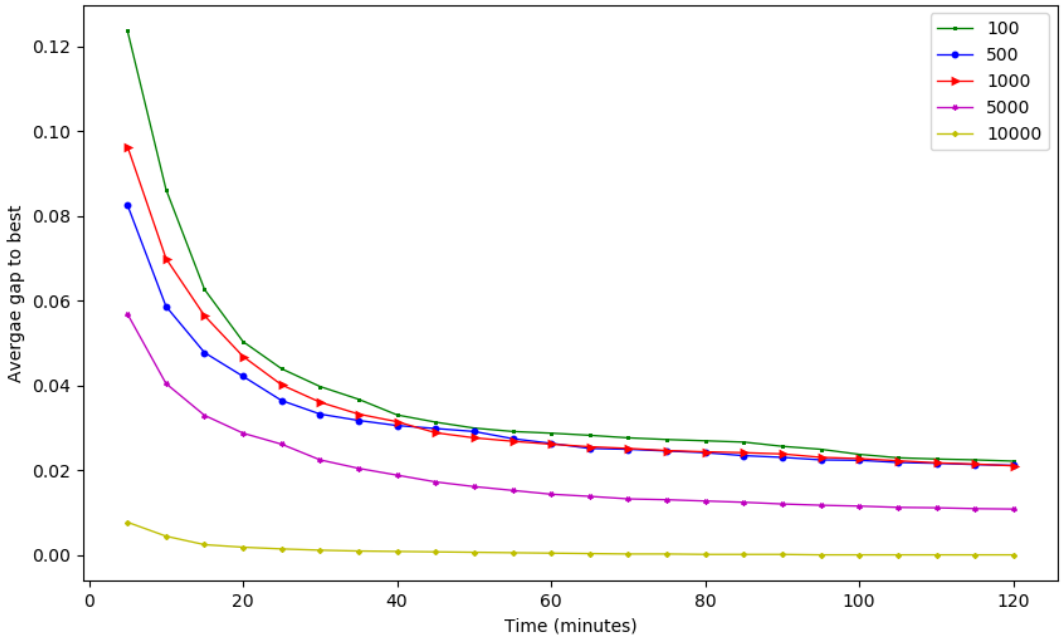


Figure 19: The performance of P-MS on the *Zuck_small* dataset with different levels of random splitting: 100, 500, 1000, 5000, 10000. The y-axis shows the gap to the best solution found across all runs.

identify what level random splitting with lead to the best overall solution and how quickly do the runs converge. For this purpose, we investigate three instances: *Kd*, *Zuck_small* and *Marvin*. For each dataset, we consider splitting levels of 100, 500, 1000, 5000 and 10000 sets per iteration. Like the previous experiments, every run was given 12 nodes. Note that the memory requirements increasing substantially with an increase in the number of sets. Hence, splitting beyond 10,000 sets is infeasible, since any further splitting leads to very large memory requirements (> 150 GB) when running 12 instances of MS concurrently.

Figures 17, 18 and 19 show the results of the experiments of different levels of random splitting. For each dataset, the figures show the progression of the gap of P-MS relative to the best solution found by any of the runs. Like the previous results, initially, more random splitting leads to improved results. Again, this is attributable to a more diverse population directly as a results of considering more sets. Overall, increasing random splitting leads to faster convergence across all datasets. Interestingly, with *Kd*, 5000 sets provide the best result, though 10000 sets also converge nearly as quickly. For all the datasets, 100, 500 and 1000 perform worse than when a larger number of sets are used, but for KD the convergence is a lot more close when using 500 and 1000 sets.

The trend as we see from the random splitting experiments is that increasing diversity is crucial in identifying good solutions. However, the trade-off is with the memory requirements, which for very large problems can be infeasible. This could be the key to achieving much better results for *Zuck_medium* and *Zuck_large* (i.e., much greater diversity with increased random splitting), which will be a possible direction to investigate in our future work.

11.3. Investigating Synchronisation Frequency

Another aspect of the algorithms that warrants investigation is the regularity of syncing between the nodes (processes). That is, the solutions obtained at each node need to be broadcast to other nodes, and in turn, the receiving nodes need to update the list of solutions with any better solution information. Synchronisation happens at regular intervals. Here we investigate if syncing within shorter time-periods can lead to better solutions and/or faster convergence.

Like the previous section, we investigate P-MS on the Kd, Marvin and *Zuck_small* datasets. Figures 20, 21 and 22 show the results for each of these datasets, respectively. The figures show the gap to the best solution (found by any of the runs) for 120 minutes. The syncing times considered are 0.1, 0.5, 1.0, 5.0 and 10 seconds.

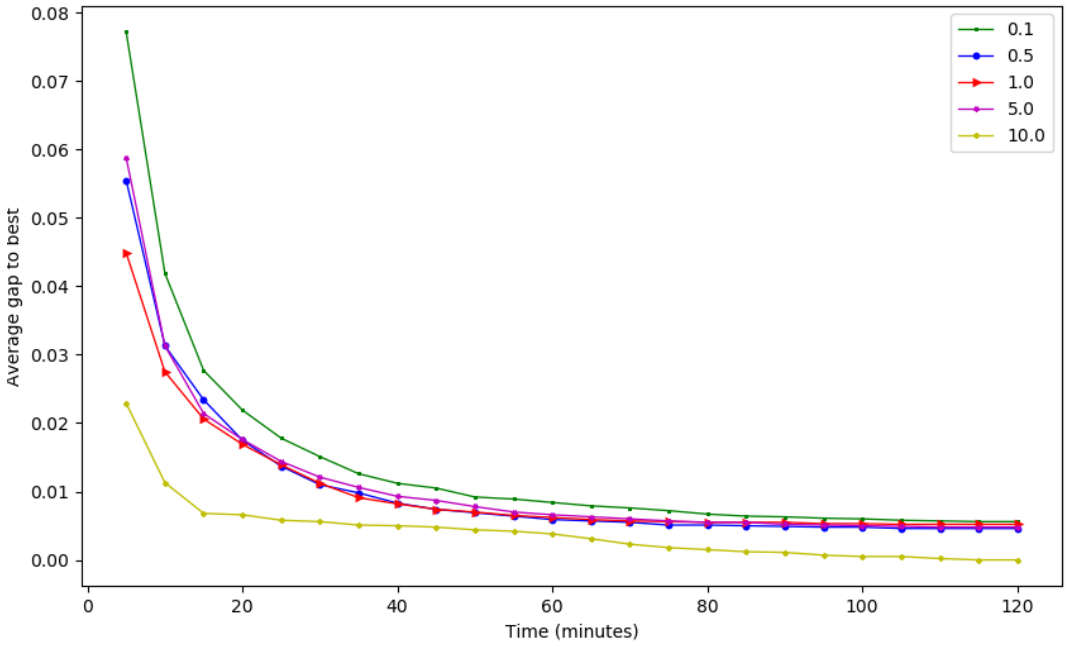


Figure 20: The performance of P-MS on the Kd dataset with different syncing times (in seconds): 0.1, 0.5, 1.0, 5.0, 10.0. The y-axis show the gap to the best solution found by all runs across.

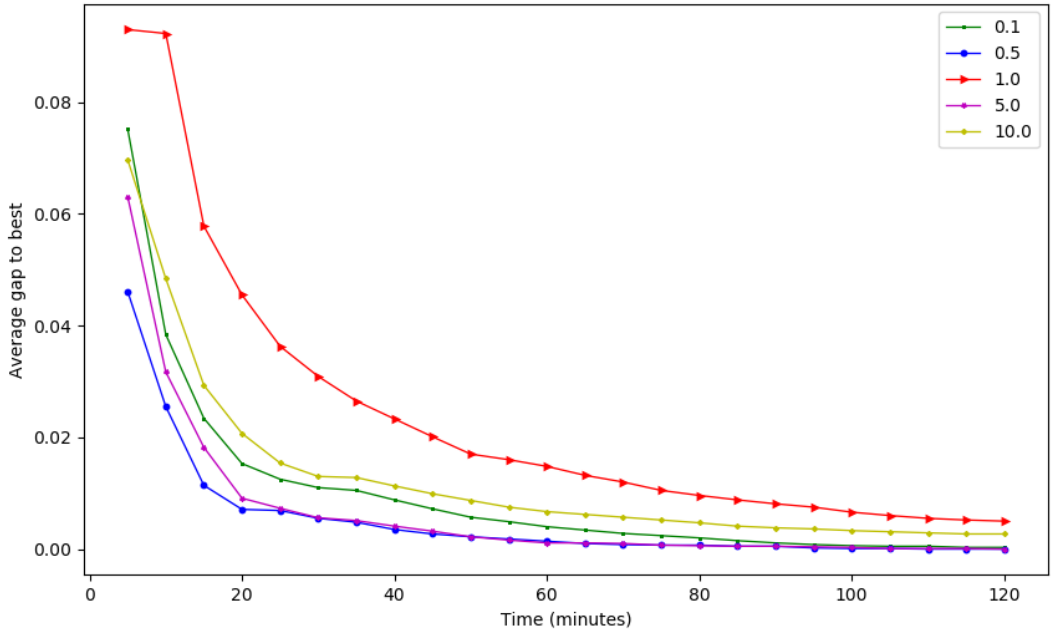


Figure 21: The performance of P-MS on the Marvin dataset with different syncing times (in seconds): 0.1, 0.5, 1.0, 5.0, 10.0. The y-axis show the gap to the best solution found by all runs across.

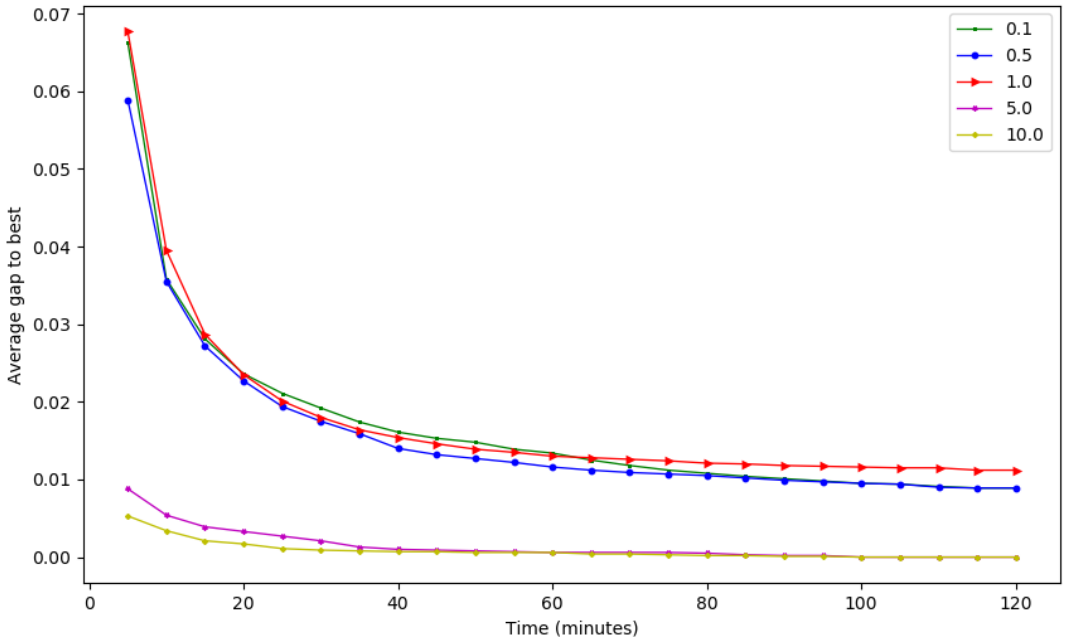


Figure 22: The performance of P-MS on the Zuck_small dataset with different syncing times (in seconds): 0.1, 0.5, 1.0, 5.0, 10.0. The y-axis show the gap to the best solution found by all runs across.

The results in these figures show that there are advantages with respect to solution quality and convergence rate. For Kd, we see that syncing too often ($\leq$ 5 seconds) is not useful and 10 or more seconds is beneficial. However, for Marvin, syncing at 0.5 or 5 seconds works well. Like, Kd, there is a distinct advantage for larger sync times ($\geq$ 5 seconds) for *Zuck_small*. The Marvin dataset consists of many more blocks and precedence constraints than Kd or *Zuck_small*, and there are also many more solution improvements generated during the course of a run. Hence, these results show that if the starting solution is poor or far away from a local or global optimal solution, syncing often can lead to finding better solutions and a faster rate of convergence.

12. Conclusion

This study investigates open pit mining, a complex optimisation problem arising in the mining industry. The problem instances can be very large with over 100,000 blocks and 1 million precedence constraints. Hence, to find good solutions in reasonable times, we propose a novel distributed approach – parallel Merge Search. The key idea behind this method is its ability to combine large populations of solutions (via variable aggregation). When applied in a distributed setting, the approach proves to be very effective. We also implement a depth first Branch & Bound approach with the aim of finding good upper bounds.

We see that Merge Search on its own finds very good solutions to known benchmark instances of open pit mining, finding better solutions on four out of seven instances. Furthermore, its parallel implementation (via MPI) proves to be the best performing method, finding best known solutions in five out of seven problem instances. On investigating its convergence characteristics, we see that solution diversity is crucial in achieving good performance. The Branch & Bound approach, while not able to find the best heuristic solutions, has produced the best known upper bounds for all the problem instances.

12.1. Future Work

Diversity in the solution population is key part of ensuring that Merge Search is implemented efficiently. Moreover, it is very flexible with the ability to combine solutions from different approaches (like heuristics, meta-heuristics, exact methods, etc.) easily. Combining these aspects, Merge Search - heuristic/meta-heuristic hybrids can be expected to achieve superior performance compared to Merge Search on its own. Hence, for future work,

we are extending Merge Search for open pit mining in this direction, where we are developing hybrids with ant colony optimisation [Dorigo and Stűtzle, 2004], particle swarm optimisation [Kennedy and Eberhart, 1995] and scheduling heuristics [Thiruvady et al., 2013]. The diversity achieved from these approaches and the ability to implement all them in a fully parallel setting, is likely to provide substantial improvements in solution quality and quicker convergence in terms of wall-clock time.

The generic nature of Merge Search means that it can be easily applied to other problems. Several optimisation problems have different solution approaches, for example, those derived from heuristics, meta-heuristics, constraint programming [Marriott and Stuckey, 1998], etc. Merge Search can be used to efficiently combine the solutions found by these methods, and will be particularly advantageous for very large problems. In fact, Merge Search is proven on the constrained pit mining problem [Kenny et al., 2018] and we have conducted preliminary studies that show that it can be applied to resource constrained job scheduling [Singh and Ernst, 2011] and resource constrained project scheduling [Kimms, 2001].

A key motivation for the development of Merge Search was distributed processing. The parallel MPI framework proposed here can be further extended to consider a very large number of nodes, which will present its own challenges. It can be expected that by carefully designing the hybrids, very large improvements can be achieved. Parallelisation via a shared memory architectures could also be of great potential. For example, within a node, individual metaheuristic methods could be implemented in parallel (e.g. multiple solutions constructed in parallel in ACO [Thiruvady et al., 2014, Ernst, 2010]).

Appendix A. Network Flow Structure

Consider the problem without resource constraints (7). Firstly, in this case the optimal solution is simply to use the more profitable option for each block b so

$$y_{b\bar{d}t} = \begin{cases} \bar{x}_{bt} - \bar{x}_{b,t-1} & \text{if } \bar{d} = \arg\max_{d \in D} p_{bdt} \\ 0 & \text{otherwise} \end{cases}$$

Hence, the problem reduces to one of maximising over $\bar{x}$ variables with profit $\bar{p}_{bt} = p_{b\bar{d}t} - p_{b,\bar{d},t+1}$. Hence, in general, since constraints (5) have the same structure as (4) we can simply consider variables $\bar{x}_i$ where $i = (b,t) \in N$ and precedence relationship $(i,j) \in \mathcal{P}$ where either $i = (a,t)$ and $j = (b,t)$ with $a \rightarrow b$ or $i = (b, t+1)$ and $j = (b,t)$. That is, the predecessors for block b at time t are the blocks a "above" b and the block b at the next time period t. Using the notation p_i to denote the arbitrary profit of i we get:

Problem P

$$\max \quad \sum_{i \in N} p_i \, \bar{x}_i \tag{A.1}$$

$$\text{s.t. } \bar{x}_j - \bar{x}_i \leq 0 \quad \forall \, i \rightarrow j \tag{A.2}$$

$$\bar{x}_i \leq 1 \quad \forall \, i \in N \tag{A.3}$$

$$\bar{x}_i \geq 0 \quad \forall \, i \in N \tag{A.4}$$

Note that we really only require (A.3) for a significantly reduced set $N^0 \subseteq N$ that have no predecessors. So this might means all of the nodes corresponding to the last time period and blocks at the top of the open cut mine. The optimal solution to this problem is a set $C \subseteq N$ of nodes to be selected such that all arcs in the cut point out of C ($i \in C$ and $j \rightarrow i \Rightarrow j \in C$) which maximises $\sum_{i \in C} p_i$.

Let ϕ_{ij} be the dual variable associated with (A.2) and μ_i correspond to (A.3) then the dual problem is:

Problem D

$$\min \quad \sum_{i \in N^0} \mu_i \tag{A.5}$$

$$\text{s.t.} \sum_{j : i \rightarrow j} \phi_{ij} - \sum_{j : j \rightarrow i} \phi_{ji} - \mu_i \leq -p_i \quad \forall \, i \in N \tag{A.6}$$

$$\phi_{ij}, \, \mu_i \geq 0 \quad \forall \, i, j \in N \tag{A.7}$$

Note that we have multiplied (A.6) by -1 to make it clearer that this is a standard network flow equation where the net outflow (sum of ϕ_{ij} going out of i minus what is coming in via arcs pointing into i and μ_i) must be at most the 'production' amount $-p_i$. In other words this is a network flow problem with the following characteristics:

- Any node can "discard" flow but each node can produce at most $-p_i$. Where $-p_i < 0$ (i.e., positive profit in Problem P), the node has a minimum amount of net flow p_i that needs to be provided (go into node i).
- Flow can be moved at no cost and with no capacity restrictions between nodes but only along the precedence arcs $i \to j$.
- A "dummy" node can be used to make up any shortfall μ_i at unit cost. That is, we are minimising the total shortfall after any supply is used to meet demand.

Alternatively we can think of this as a maximum flow problem where

- We start with the basic network defined by the $i \to j$ arcs and add a source α and sink node ω.
- Each node i for which $-p_i > 0$ (i.e. a loss in our original problem) we create an arc from the source node to i ($\alpha \to i$) with capacity $-p_i$.
- For all other nodes $p_i \geq 0$ we add an arc from i to a sink ($i \to \omega$) with capacity p_i.
- Maximising the flow gives an optimal solution f^{max} such that $\sum_{i:p_i>0} p_i - f^{max}$ equals the optimal solution value of Problems P and D and the corresponding optimal cut matches (is the complement of) the optimal cut of C.

Appendix B. Incremental destination variables

If we define the amount sent to destinations incrementally we obtain the cumulative variables $\bar{y}_{bdt}$. These specify the amount of flow that has been sent to any destination before time t, or to a higher indexed destination d during t:

$$\bar{y}_{bdt} = \sum_{\tau<t} \sum_{\delta \in D} y_{b\delta\tau} + \sum_{\delta \geq d} y_{b\delta t}$$

Note that now $\bar{x}_{bt} = \bar{y}_{b0t}$. Also in our data sets we have only two destinations with $\bar{y}_{b0t}$ being the total amount mined (irrespective of destination) and $\bar{y}_{b1t}$ the amount sent to be processed (destination 1) for block b by time t.

To simplify the presentation and match the notation for block predecessors we define a "predecessor" (really later in time or lower indexed destination) relationship with $(d,t) \to (\delta,\tau)$ if

$$d = \delta - 1 \wedge t = \tau; \text{ or } \delta = 0 \wedge d = |D| - 1 \wedge t > \tau.$$

It can be seen that here (δ,τ) is really a unique "successor" to (d,t). Also we have $y_{bdt} = \bar{y}_{bdt} - \bar{y}_{b,d',t'}$ where $(d,t) \to (d',t')$ is the predecessor. Using these new variables and expanded precedence relationships we obtain a new formulation for problem PCPSP:

$$\max \quad \sum_{b \in B} \sum_{d \in D} \sum_{t \in T} \bar{p}_{bdt}\, \bar{y}_{bdt} \tag{B.1}$$

$$\text{s.t.} \quad \bar{y}_{b0t} \leq \bar{y}_{a0t} \qquad \forall\ (a,b) \in \mathcal{P},\ t \in T \tag{B.2}$$

$$\bar{y}_{bd't'} \leq \bar{y}_{bdt} \qquad \forall\ b \in B,\ d \in D,\ t \in T,\ \text{if } (d,t) \to (d',t') \tag{B.3}$$

$$\sum_{b \in B} \sum_{d \in D} q_{bdr} \left(\bar{y}_{bdt} - \bar{y}_{bd't'}|_{(d,t)\to(d',t')}\right) \leq \bar{R}_{rt} \qquad \forall\ r \in R,\ t \in T \tag{B.4}$$

$$\bar{y}_{b0t} \in \{0,1\}, 0 \leq \bar{y}_{bdt} \leq 1 \qquad \forall\ b \in B,\ d \in D,\ t \in T \tag{B.5}$$

This has just precedence ($\leq$) style constraints plus resource constraints similar to the formulation based on $\bar{x}$ above. Here $\bar{p}_{bdt} = p_{bdt} - p_{bd't'}$ where $(d',t') \to (d,t)$ to ensure we get the right cost. As for the previous formulation, we only need $\bar{y}_{b,0,0} \geq 0$ and $\bar{y}_{b,D,T} \leq 1$ with all other bounds on $\bar{y}_{b,d,t}$ variables implied by the relationship (B.3).

Appendix C. Aggregation Formulation

Let S be a set of (b,d,t) triplets and $\bar{y}_S$ be a corresponding variable that indicates that all $\bar{y}_{bdt}$ have the same value $\bar{y}_S$. Assume we have a collection of such sets $\mathcal{S}$ that is the union of one or more partitions of the set $\mathcal{N}$ of all such triplets. Typically, we are only interested in connected subsets of $\mathcal{N}$ (by the precedence and predecessor arcs $\rightarrow$). In particular, we have special cases:

- $\mathcal{S} = \{\ \{(b,d,t)\} \mid (b,d,t) \in \mathcal{N}\}$ where the $\bar{y}_S$ variables are identical to the original $\bar{y}_{bdt}$ variables.
- $\mathcal{S} = 2^{\mathcal{N}}$ where every connected subset of $\mathcal{N}$ is included (of theoretical interest only).
- Some non-trivial partition of $\mathcal{N}$ leading to a column generation master problem that may or may not have the same optimal solution as our original problem.

We want to set up this kind of master problem and incrementally refine the partition to get the optimal solution to the LP relaxation of our problem. To enable a compact definition of this formulation we define:

$B(S)$ the set of blocks for which some triplet $(b,d,t) \in S$ so that $B(S) \subseteq B$.

$\alpha(b,S), \omega(b,S)$ $\alpha(b,S)$ the first triplet not in S and the last triplet in S of the chain involving block b. That is, assume that we have:

$$\alpha(b,S) = (b,d_0,t_0) \rightarrow (b,d_1,t_1) \rightarrow \ldots \rightarrow (b,d_k,t_k) = \omega(b,S)$$

Then $(b,d_0,t_0) \notin S$, and $(b,d_i,t_i) \in S$ for $i = 1,\ldots,k$.

- Alternative ways to set up these chains are possible, but we use the same ordering as for the incremental formulation described in Section Appendix B.
- $\alpha(b,S)$ may not exist - S may contain the first triplet for a block. This just means that in the sums below, the corresponding constant is not included.

$S \rightarrow S'$ if $\exists (b,d,t) \in S, (b',d',t') \in S' : (b,d,t) \rightarrow (b',d',t')$
With this definition it is possible to have $S \rightarrow S'$ and $S' \rightarrow S$. We restrict ourselves to only consider partitions $\mathcal{S}$ where this does not occur.

$\bar{p}_S$ The total profit due to all $(b,d,t) \in S$:

$$\bar{p}_S = \sum_{(b,d,t)\in S} \bar{p}_{bdt} = \sum_{b\in B(S)} p_{b\omega(b,S)} - p_{b\alpha(b,S)}$$

$\bar{q}_{Srt}$ the amount of resource r consumed in t if we select all nodes in S.

$$\bar{q}_{Srt} = \sum_{b,d:(b,d,t)\in S} \bar{q}_{b,d,t} = \sum_{b,d:\omega(b,S)=(d,t)} q_{b,d,t} - \sum_{b,d:\alpha(b,S)=(d,t)} q_{b,d,t}$$

Problem PCPSP($\mathcal{S}$)

$$\max \quad \sum_{S\in\mathcal{S}} \bar{p}_S\, \bar{y}_S \tag{C.1}$$

$$\text{s.t.} \quad \bar{y}_{S'} \le \bar{y}_S \quad \forall\, S \rightarrow S' \in \mathcal{S} \tag{C.2}$$

$$\sum_{S\in\mathcal{S}} q_{Srt}\bar{y}_S \le \bar{R}_{rt} \quad \forall\, r \in R,\ t \in T \tag{C.3}$$

$$0 \le \bar{y}_S \le 1 \quad \forall\, S \in \mathcal{S} \tag{C.4}$$

$$\bar{y}_S \in \{0,1\} \quad \forall\, S \in \mathcal{S} : \exists (b,0,t) \in S \tag{C.5}$$

Any solution to PCPSP($\mathcal{S}$) clearly corresponds to a primal feasible solution of the original PCPSP (with all of the $\bar{y}_{bdt}$ set according to the corresponding $\bar{y}_S$). To prove that a solution is *optimal* we need to ensure dual feasibility. That is, we need to ensure we have a way of assigning dual variables (flows) for all of the internal edges that have been dropped by using the aggregation $\mathcal{S}$. For a particular $S \in \mathcal{S}$ this means we need a feasible solution to a flow problem with supply equal to the negative of reduced costs based on duals of λ_{rt} of (C.3). Thus supply σ_{bdt} is given by:

$$\sigma_{bdt} = -\bar{p}_{bdt} - \sum_{rt:\omega(b,S)=(d,t)} \lambda_{rt} q_{bdt} + \sum_{rt:\alpha(b,S)=(d,t)} \lambda_{rt} q_{bdt} \tag{C.6}$$

In addition, we introduce dummy supply nodes S' for all other sets $S' \neq S \in \mathcal{S}$ with supply $\phi_{S'S}$ (dual of (C.2) where $S' \rightarrow S$) and arcs $S' \rightarrow (b,d,t)$ if $\exists (b',d',t') \in S' : (b',d',t') \rightarrow (b,d,t) \in S$. Also, potentially dummy sink nodes S'' for any set $S'' \in \mathcal{S}$ with supply $-\phi_{SS''}$ (dual of (C.2) where $S \rightarrow S''$) and arcs $(b,d,t) \rightarrow S''$ if $\exists (b'',d'',t'') \in S'' : (b,d,t) \rightarrow (b'',d'',t'')$. Note that we can in principle check this feasibility by solving max flow from the dummy source to dummy sink nodes (with arcs limited by ϕ). If this produces a cut with all of the $(b,d,t) \in S$ on the same side of the cut (i.e., we are just cutting off the source or sink nodes) then the solution is dual feasible, otherwise we have a way of partitioning S into two subsets which could lead to an improvement of the solution. This forms the basis for the Bienstock-Zuckerberg Algorithm.

Acknowledgements

This research was supported in part by the Monash eResearch Centre and eSolutions-Research Support Services through the use of the MonARCH HPC Cluster.

References

References

Ahuja, R.K., Ergun, O., Orlin, J.B., Punnen, A.P., 2002. A Survey of Very Large-scale Neighborhood Search Techniques. Discrete Appl. Math. 123, 75–102.

Bienstock, D., Zuckerberg, M., 2010. Solving lp relaxations of large–scale precedence constrained problems, in: International Conference on Integer Programming and Combinatorial Optimization, Springer–Verlag. pp. 1–14.

Bley, A., Boland, N., Fricke, C., Froyland, G., 2010. A Strengthened Formulation and Cutting Planes for the Open Pit Mine Production Scheduling Problem. Computers and Operations Research 37, 1641–1647.

Blum, C., 2016. Construct, merge, solve and adapt: Application to unbalanced minimum common string partition, in: Blesa, M.J., Blum, C., Cangelosi, A., Cutello, V., Di Nuovo, A.G., Pavone, M., Talbi, E.G. (Eds.), Proceedings of HM 2016 – 10th International Workshop on Hybrid Metaheuristics, Springer Berlin Heidelberg. pp. 17–31.

Blum, C., Blesa, M.J., 2016. Construct, merge, solve & adapt: Application to the repetition-free longest common subsequence problem, in: Chicano, F., Hu, B. (Eds.), Proceedings of EvoCOP 2016 – 16th European Conference on Evolutionary Computation in Combinatorial Optimization, Springer Berlin Heidelberg. pp. 46–57.

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.

Boland, N., Dumitrescu, I., Froyland, G., Gleixner, A.M., 2009. LP-based Disaggregation Approaches to Solving the Open Pit Mining Production Scheduling Problem with Block Processing Selectivity. Computers & Operations Research 36, 1064 – 1089.

Boykov, Y., Kolmogorov, V., 2001. An Experimental Comparison of Min-cut/Max-flow Algorithms for Energy Minimization in Vision, in: Figueiredo, M., Zerubia, J., Jain, A.K. (Eds.), Energy Minimization Methods in Computer Vision and Pattern Recognition, Springer Berlin Heidelberg.

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, ACM. pp. 583–592.

Caccetta, L., Hill, S.P., 2003. An Application of Branch and Cut to Open Pit Mine Scheduling. Journal of Global Optimization 27, 349–365.

Chakrabarty, D., Könemann, J., Pritchard, D., 2010. Hypergraphic lp relaxations for steiner trees, in: International Conference on Integer Programming and Combinatorial Optimization, Springer. pp. 383–396.

Chapman, B., Jost, G., van der Pas, R., 2007. Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation). The MIT Press.

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. Springer, pp. 101–173.

Dorigo, M., Stűtzle, T., 2004. Ant Colony Optimization. MIT Press, Cambridge, Massachusetts, USA.

Dorit S. Hochbaum, Anna Chen, 2000. Performance Analysis and Best Implementations of Old and New Algorithms for the Open-Pit Mining Problem. Operations Research 48, 894–914.

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

Duin, C., 2000. Preprocessing the steiner problem in graphs, in: Advances in Steiner Trees. Springer, pp. 175–233.

Ernst, A.T., 2010. A hybrid Lagrangian Particle Swarm Optimization Algorithm for the degree-constrained minimum spanning tree problem, in: IEEE Congress on Evolutionary Computation, pp. 1–8. doi:http://dx.doi.org/10.1109/CEC.2010.5585939. 00009.

Espinoza, D., Goycoolea, M., Moreno, E., Newman, A., 2012. MineLib: a library of open pit mining problems. Annals of Operations Research 206, 93–114.

Fisher, M., 2004. The Lagrangian Relaxation Method for Solving Integer Programming Problems. Management Science 50, 1861–1871.

Geoffrion, A.M., 1972. Generalized Benders decomposition. Journal of Optimization Theory and Applications 10, 237–260.

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

Gropp, W., Lusk, E., Skjellum, A., 1994. Using MPI: Portable Parallel Programming with the Message-passing Interface. MIT Press, Cambridge, MA, USA.

Hochbaum, D.S., Chen, A., 2000. Performance Analysis and Best Implementations of Old and New Algorithms for the Open-Pit Mining Problem. Operation 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, Springer. pp. 223–232.

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. Springer, pp. 85–103.

Kennedy, J., Eberhart, R., 1995. Particle swarm optimization, in: Neural Networks, 1995. Proceedings., IEEE International Conference on, pp. 1942–1948.

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, ACM, New York, NY, USA. pp. 316–323.

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, 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, ACM, New York, NY, USA. pp. 1137–1144. doi:10.1145/3071178.3071241.

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.

Kimms, A., 2001. Maximizing the Net Present Value of a Project Under Resource Constraints Using a Lagrangian Relaxation Based Heuristic with Tight Upper Bounds. Annals of Operations Research 102, 221–236.

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

Klau, G.W., Ljubić, I., Moser, A., Mutzel, P., Neuner, P., Pferschy, U., Raidl, G., Weiskircher, R., 2004. Combining a memetic algorithm with integer programming to solve the prize-collecting steiner tree problem, in: Genetic and Evolutionary Computation Conference, Springer. pp. 1304–1315.

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

Leitner, M., Ljubić, I., Luipersbeck, M., Resch, M., 2014. A partition-based heuristic for the steiner tree problem in large graphs, in: International Workshop on Hybrid Metaheuristics, Springer. pp. 56–70.

Lerchs, H., Grossman, I.F., 1964. Optimum Design of Open-Pit Mines. Inst. Operations Research Management Sciences.

Lewis, R., Thiruvady, D., Morgan, K., 2019. Finding Happiness: An Analysis of the Maximum Happy Vertices Problem. Computers & Operations Research 103, 265–276.

Litvinchev, I., Tsurkov, V., 2003. Aggregation in Large-Scale Optimization. Springer, Springer USA.

Luke, S., 2009. Essentials of metaheuristics .

Marriott, K., Stuckey, P., 1998. Programming With Constraints. MIT Press, Cambridge, Massachusetts, USA.

Martínez, G.I.M., 2012. Modelos de optimización lineal entera y aplicaciones a la minería. Ph.D. thesis. Facultad de Ciencias Fśicas y Matemáticas. Chile.

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.

Mitchell, M., 1996. An Introduction to Genetic Algorithms. MIT Press, Cambridge, MA, USA.

Newman, A.M., Rubio, E., Caro, R., Weintraub, A., Eurek, K., 2010. A review of operations research in mine planning. Interfaces 40, 222–245.

Nguyen, T.D., Do, P.T., 2013. An ant colony optimization algorithm for solving group steiner problem, in: The 2013 RIVF International Conference on Computing & Communication Technologies-Research, Innovation, and Vision for Future (RIVF), IEEE. pp. 163–168.

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

Pisinger, D., Ropke, S., 2010. Large Neighborhood Search. Springer US, Boston, MA. pp. 399–419.

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, Springer. pp. 214–225.

Ramazan, S., Dimitrakopoulos, R., 2004. Recent Applications of Operations Research in Open Pit Mining. Transactions of the Society for Mining, Metallurgy and Exploration 316, 73–78.

Rogers, D.F., Plante, R.D., Wong, R.T., Evans, J.R., 1991. Aggregation and Disaggregation Techniques and Methodology in Optimization. Oper. Res. 39, 553–582.

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., Das, S., Gosavi, S.V., Pujar, S., 2005. Ant colony algorithms for steiner trees: an application to routing in sensor networks, in: Recent developments in biologically inspired computing. Igi Global, pp. 181–206.

Singh, G., Ernst, A.T., 2011. Resource Constraint Scheduling with a Fractional Shared Resource. Operations Research Letters 39, 363 – 368.

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.

Thiruvady, D., Blum, C., Ernst, A.T., 2019. Maximising the Net Present Value of Project Schedules Using CMSA and Parallel ACO, in: Blesa Aguilera, M.J., Blum, C., Gambini Santos, H., Pinacho-Davidson, P., Godoy del Campo, J. (Eds.), Hybrid Metaheuristics, Springer International Publishing, Cham. pp. 16–30.

Thiruvady, D., Ernst, A.T., Singh, G., 2014. Parallel Ant Colony Optimization for Resource Constrained Job Scheduling. Annals of Operations Research , 1–18.

Thiruvady, D., Singh, G., Ernst, A.T., Meyer, B., 2013. Constraint-based ACO for a Shared Resource Constrained Scheduling Problem. International Journal of Production Economics 141, 230–242. Meta-heuristics for manufacturing scheduling and logistics problems.

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

Uchoa, E., Werneck, R.F., 2010. Fast local search for steiner trees in graphs, in: Proceedings of the Meeting on Algorithm Engineering & Expermiments, Society for Industrial and Applied Mathematics. pp. 1–10.

Underwood, R., Tolwinski, B., 1998. A Mathematical Programming Viewpoint for Solving the Ultimate Pit Problem. European Journal of Operational Research 107, 96 – 107.

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

Wade, A., Rayward-Smith, V., 2000. Effective local search techniques for the steiner tree problem, in: Advances in Steiner Trees. Springer, pp. 255–281.

Wolsey, L.A., 1998. Integer programming. Wiley-Interscience, New York, NY, USA.