Scheduling arc maintenance jobs in a network to maximize total flow over time[*]

Natashia Boland Thomas Kalinowski Hamish Waterer Lanbo Zheng

Abstract

We consider the problem of scheduling a set of maintenance jobs on the arcs of a network so that the total flow over the planning time horizon is maximized. A maintenance job causes an arc outage for its duration, potentially reducing the capacity of the network. The problem can be expected to have applications across a range of network infrastructures critical to modern life. For example, utilities such as water, sewerage and electricity all flow over networks. Products are manufactured and transported via supply chain networks. Such networks need regular, planned maintenance in order to continue to function. However the coordinated timing of maintenance jobs can have a major impact on the network capacity lost to maintenance. Here we describe the background to the problem, define it, prove it is strongly NP-hard, and derive four local search-based heuristic methods. These methods integrate exact maximum flow solutions within a local search framework. The availability of both primal and dual solvers, and dual information from the maximum flow solver, is exploited to gain efficiency in the algorithms. The performance of the heuristics is evaluated on both randomly generated instances, and on instances derived from real-world data. These are compared with a state-of-the-art integer programming solver.

1 Introduction

We consider a problem in which a network with arc capacities is given, together with, for each arc of the network, a set of maintenance jobs that need to be carried out on the arc. Each maintenance job has a duration, and a time window during which it must start. A maintenance job cannot be pre-empted; once started it will continue for its duration. This situation could arise in a range of network infrastructure settings, for example, when considering maintenance on pipe sections in a water network, or track sections in a rail network. Such maintenance causes network arc outages, leading to capacity reduction in the network. Here we measure network capacity as the value of the maximum flow in the network. This has the advantage of being the simplest way of measuring network capacity. It is also the approach taken by our industry partner in the application that motivated this research. The objective of the problem is to schedule all the maintenance jobs so that the total flow over time is maximized.

We were led to consider this problem through our collaboration with the Hunter Valley Coal Chain Coordinator Limited (HVCCC). The Hunter Valley Coal Chain (HVCC) constitutes mining companies, rail operators, rail track owners and terminal operators, together forming the world's largest coal export facility. In 2008, the throughput of the HVCC was about 92 million tonnes, or more than 10% of the world's total trade in coal for that year. The coal export operation generates around \$15 billion in annual export income for Australia. As demand has increased significantly in recent years and is expected to increase further in the future, efficient supply chain management

[*] *Discrete Applied Mathematics*, in press, `doi:10.1016/j.dam.2012.05.027`

is crucial. Our industry partner, the HVCCC was founded to enable integrated planning and coordination of the interests of all involved parties, so as to improve the efficiency of the system as a whole. More details on the HVCC can be found in [1].

The problem discussed in this paper was motivated by the annual maintenance planning process carried out by the HVCCC. Supply chain components such as railway track sections, terminal equipment and load points have to undergo regular preventive and corrective maintenance, causing a significant loss in system capacity (up to 15%). The HVCCC had observed that careful scheduling of the maintenance jobs – good alignment of them – could reduce the impact of maintenance on the network capacity, and established a regular planning activity to carry it out, called "capacity alignment". Currently capacity alignment for the approximately 1500 maintenance jobs planned each year is a labour-intensive, largely manual process, achieved by iterative negotiation between the HVCCC and the individual operators.

The HVCCC currently uses an automated rule-based calculator to evaluate the quality of candidate maintenance schedules. In-depth analysis of both the calculator and the HVCC coal handling system revealed this to be well modelled as a maximum flow problem in a network in which the coal flows from the mines to the ships. The arcs represent the relevant pieces of infrastructure: load points, rail track and different machines at the terminals. A maintenance job on a piece of the infrastructure simply means that the corresponding arc cannot carry any flow for the duration of the job. The natural objective is to schedule the maintenance tasks such that the total flow over the time horizon is maximized. This corresponds to, e.g., annual throughput capacity of the HVCC.

The maintenance jobs themselves are scheduled initially according to standard equipment requirements, which typically dictate particular types of maintenance jobs be performed at particular time points. After discussions with the maintenance planners, it emerged that they would be prepared to move the jobs, usually for intervals of plus or minus 7 days, in order to achieve better overall throughput of the system. We initially expected there would be some inter-maintenance constraints, for example, that a type of job carried out at four-week intervals could not be carried out more than 5 weeks apart. But the maintenance planners were not concerned about this issue, and preferred the simple assumption that jobs could not deviate more than some fixed number of days around their initial scheduled time. This gives rise to a simple release date and due date job scheduling structure.

The problem of scheduling maintenance jobs in a network so as to maximize the total flow over time has some aspects of dynamic maximum flow. The concept was introduced by Ford and Fulkerson [5]: given a network with transit times on the arcs, determine the maximum flow that can be sent from a source to a sink in T time units. In the application of interest to us, there are no transit times on arcs, but the capacities vary over time. This leads to a different type of dynamic flow problem. Variations of the dynamic maximum flow problem with zero transit times are discussed in [3, 8, 9], while piecewise constant capacities are investigated by Ogier [14] and Fleischer [4]. For a comprehensive survey on dynamic network flow problems we refer the reader to [11, 18], and for a recent, very general treatment of maximum flows over time to [10]. For a given maintenance schedule, the capacities on the arcs jump between zero and their natural capacity, and so are piecewise constant. Thus the problem of evaluating a maintenance schedule could be viewed as a dynamic maximum flow problem of this type. However, in our case the piecewise constant function is a function of the maintenance schedule, and hence of the schedule decision variables. This makes our problem quite different.

The problem does have a superficial resemblance to machine scheduling problems (see, e.g., the book by Pinedo [15]), but there is no underlying machine, and the association of jobs with network arcs and a maximum flow objective give it quite a different character. Classical machine scheduling seeks to carry out jobs as quickly as possible (in some sense). The maximum flow objective motivates quite different strategies. For example, if arcs are "in sequence" in some sense, it is better to overlap the corresponding maintenance jobs in time as much possible, whereas if they are "in parallel", it is better to schedule them with as little overlap as possible.

There is also some resemblance to network design problems (fixed charge network flows), see e.g. Nemhauser and Wolsey [12] and references therein, but in such problems the arcs are either designed in, or out, of the network in a single-period setting. Even a multi-period variant (see for example the recent work of Toriello *et al.* [19]) would not capture the need for consecutive period outages implied by a maintenance activity.

An emerging research area that also blends network flow and scheduling elements arises in restoration of network services in the wake of a major disruption. For example, Nurre *et al.* [2] schedule arc restoration tasks so as to maximize total weighted flow over time. They consider dispatch rule based heuristics and integer programming approaches. The latter performed well in sewerage, small power infrastructure, and emergency supply chain cases, solving most instances to optimality in a matter of seconds, but the heuristic was competitive in terms of more quickly finding good quality solutions. The heuristic was also especially effective in a large power infrastructure case, finding nearly as good solutions as the exact approach in far less time (see also [13]). We note that the scheduling part of the problem considered in [2] is more similar to a classical scheduling setting than ours is: the restoration activity for each arc needs to be scheduled on a machine, (work group), and one wants to complete all jobs as quickly as possible.

Thus although there are connections of our problem to existing problems, we believe that this is the first time that the problem has been considered. We believe it has a wide range of natural applications, a very attractive structure, with tractable special cases (a few of which we discuss), and some interesting extensions. We thus hope that this paper will stimulate further research on the problem and its variants. Our contributions in this paper are first to define and introduce the problem, prove it is strongly NP-hard, and discuss some tractable special cases. We then propose four different local search heuristics. The heuristics integrate exact maximum flow solutions within a local search framework, exploiting the max flow objective function structure, the availability of both primal and dual solvers, and dual information, to gain efficiency in the algorithms. The heuristics proved to be very effective on both randomly generated and real-world instances, significantly out-performing a pure integer programming approach, particularly on larger, harder problems.

The paper is organized as follows. In Section 2, the problem is formally defined, formulated as an integer program, and proved to be NP-hard. We also outline some tractable special cases. In Section 3, our local search algorithms for solving the problem are presented. Section 4 contains computational results on randomly generated test instances, as well as on two instance derived from real world data. Finally, we summarize the paper in Section 5 and point out some directions for further investigation.

2 Problem Definition and Complexity Results

Throughout we use the notation $[k,l] = \{k, k+1, \dots, l\}$ and $[k] = \{1, 2, \dots, k\}$ for $k, l \in \mathbb{Z}$. Let (N, A, s, s', u) be a network with node set N, arc set A, source s and sink s', and capacities $u_a \in \mathbb{N}$ for $a \in A$. Also, for a node $v \in N$ let $\delta^-(v)$ and $\delta^+(v)$ denote the set of arcs entering and leaving node v, respectively. We consider the network over a time horizon $[T]$. A *maintenance job* j is specified by its associated arc $a_j \in A$, its processing time $p_j \in \mathbb{N}$, its release date $r_j \in [T]$, and its deadline $d_j \in [T]$. Let J be a set of maintenance jobs, and let let J_a denote the set of jobs $j \in J$ with $a_j = a$. For each job $j \in J$ we have to choose a start time $S_j \in [r_j, d_j - p_j + 1]$ within the *time window* for the job. In our model, jobs cannot be preempted, i.e. scheduling a maintenance job to start at time S_j makes the arc a_j unavailable at times S_j, S_j+1,…, S_j+p_j-1. Thus for a given maintenance job schedule $(S_j)_{j\in J}$, the arc a has capacity zero at time t if for some $j\in J_a$, $t\in[S_j,S_j+p_j-1]$, and u_a otherwise, for each time $t\in[T]$. The problem we consider is to schedule a set J of maintenance jobs so as to maximize the total throughput over the interval $[T]$, i.e. so as to maximize the sum over t of the flows that can be sent from s to s' in the network, given the arc capacities at time $t\in[T]$ implied by the maintenance schedule. In this paper we assume unlimited resources in terms of workforce and machines, i.e. all jobs could be processed at the same time as far as their time windows allow. It is in principle straightforward to add constraints, for instance limiting the number of jobs requiring use of a given resource processed at any given time. We did not do that because in the HVCCC context the input for the optimization consists of initial maintenance schedules for the different parts of the system (rail network and terminals) with relevant resource constraints already taken into account.

For this paper we make the additional assumption that the different jobs associated with an arc do not overlap, i.e. we assume that for any two jobs j and j' on arc a, $[r_j,d_j]\cap[r_{j'},d_{j'}]=\varnothing$. This assumption can be made without loss of generality, as the general case can be reduced to this case by replacing any arc violating the assumption by a path, distributing the intersecting jobs among the arcs of the path. The reason for making the assumption is to simplify the presentation of the heuristics below: the local effect of moving a job j (i.e. the effect on the capacity of the arc associated with j) depends only on job j.

We formally define the problem via an integer programming formulation, which we also use to provide a baseline for computational testing. We introduce the following variables.

- For $a\in A$ and $t\in[T]$
  - $\phi_{at}\in\mathbb{R}_+$ is the flow on arc a over time interval t,
  - $x_{at}\in\{0,1\}$ indicates the availability of arc a at time t. These variables are not strictly needed, but are included for convenience.
- For $j\in J$ and $t\in[r_j,d_j-p_j+1]$, $y_{jt}\in\{0,1\}$ indicates if job j starts at time t.

Now we can write down the problem *maximum total flow with flexible arc outages* (**MaxTFFAO**).

$$z=\max\sum_{t=1}^{T}\sum_{a\in\delta^+(s)}\phi_{at} \tag{1}$$

$$\text{s.t.}\ \sum_{a\in\delta^-(v)}\phi_{at}-\sum_{a\in\delta^+(v)}\phi_{at}=0 \qquad \big(v\in N\setminus\{s,s'\},\ t\in[T]\big), \tag{2}$$

$$\phi_{at}\leqslant u_a x_{at} \qquad (a\in A,\ t\in[T]), \tag{3}$$

$$\sum_{t=r_j}^{d_j-p_j+1} y_{jt}=1 \qquad (j\in J), \tag{4}$$

$$x_{at}+\sum_{t'=\max\{r_j,t-p_j+1\}}^{\min\{t,d_j\}} y_{jt'}\leqslant 1 \qquad (a\in A,\ t\in[T],\ j\in J_a). \tag{5}$$

The objective (1) is to maximize the total throughput. Constraints (2) and (3) are flow conservation and capacity constraints, respectively, (4) requires that every job j is scheduled exactly once, and (5) ensures that an arc is not available while a job is being processed.

Example 1. Consider the network in Figure 1 over a time horizon $T = 6$ with the job list given in Table 1. Figure 2 shows that the total throughput can vary significantly depending on the scheduling of the jobs. Observation of this example shows that, all other things being equal, it is better for jobs on arcs that are "in series" to overlap as much as possible, and for jobs on arcs that are "in parallel" to overlap as little as possible. Thus the job on d should overlap as little as possible with the jobs on e and f, which should overlap as much as possible, and the job on a should overlap as much as possible with those on e and f. This is achieved in the second schedule in Figure 2. Of course the situation is more complex for general networks, but the insight can be useful.

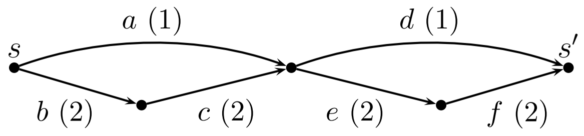


Figure 1: An example network. Capacities are indicated in brackets.

j	arc	p_j	r_j	d_j
1	a	3	1	5
2	d	2	2	5
3	e	2	2	5
4	f	2	3	6

Table 1: Example job list.

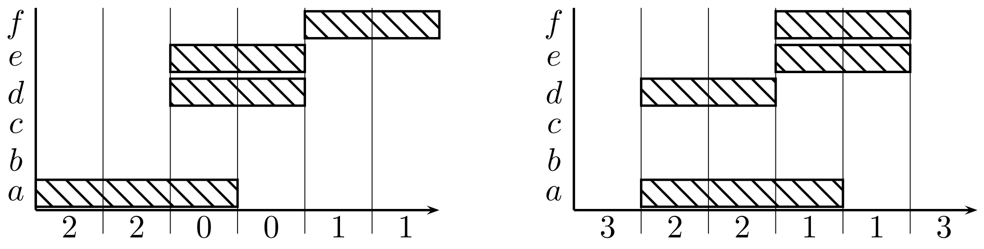


Figure 2: Two schedules for the example problem. In the horizontal direction, we have the 6 unit time intervals, and in the vertical direction there are the 6 arcs. The shaded rectangles indicate the jobs, and below the x-axis is the maximum flow for each time period. The left schedule yields a total flow of 6, while for the right schedule we obtain a total flow of 12.

Next we observe that the problem **MaxTFFAO** is strongly NP-hard, suggesting that in order to tackle instances of practical relevance efficient heuristics might be needed.

Proposition 1. *The problem **MaxTFFAO** is strongly NP-hard.*

Proof. Reduction from 3-partition (see [6]).

Instance. $B \in \mathbb{N}$, $u_1, \dots, u_{3m} \in \mathbb{N}$ with $B/4 < u_i < B/2$ for all i and $\sum_{i=1}^{3m} u_i = mB$.

Problem. Is there a partition of $[3m]$ into m triples (i, j, k) with $u_i + u_j + u_k = B$?

The corresponding network has 3 nodes: s, v and s'. There are $3m$ arcs from s to v with capacities u_i $(i = 1, \dots, 3m)$ and one arc from v to s' with capacity $(m-1)B$ (see Fig. 3).

There is one job with unit processing time for each arc from s to v, with release dates $r_j = 1$ and deadlines $d_j = m$ for all j. It is easy to see that the 3-partition instance has a positive answer if and only if there is a schedule allowing a total flow of $m(m-1)B$. If there is a 3-partition then the i-th of the m triples corresponds to three jobs to be processed in time period i. □

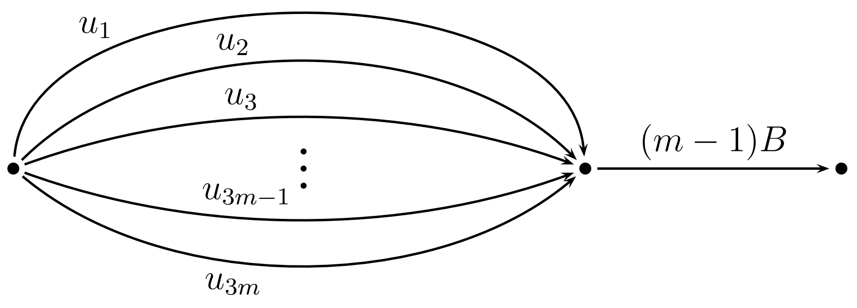


Figure 3: The network for the NP-hardness proof.

We conclude this section with some remarks on certain special cases.

1. If the network is a directed path and all the jobs have release date $r_j = 1$ and deadline $d_j = T$, it is optimal to start all jobs at the same time, say $S_j = 1$ for all j. This follows since the max flow equals the minimum of the arc capacities if all arcs are available, and 0 otherwise. So
$$\min_{a\in A} u_a \cdot \left(T - \max_{j\in J} p_j\right)$$
is an upper bound for the objective z which is attained for the described solution. More generally, if $\bigcap_{j\in J}[r_j, d_j - p_j + 1] \neq \varnothing$, any element t of this intersection determines an optimal solution by putting $S_j = t$ for all $j \in J$.

2. In general, if the network is a path and all jobs have unit processing time, the problem is equivalent to the vertex cover problem on the hypergraph with vertex set $[T]$ and edge set $\{[r_j, d_j] \ : \ j \in J\}$. This is a 0-1 integer programming problem with an interval matrix as coefficient matrix. So it is totally unimodular and can be solved efficiently by linear programming. Another interpretation of this case is that we are looking for a smallest set of time periods, such that all jobs can start at a time given in the set.

3. Inspired by the construction in the hardness proof in Proposition 1, we can ask under what conditions an instance of **MaxTFFAO** with unit processing times and jobs that can move freely ($r_j = 1$ and $d_j = T$) is optimally solved by scheduling all jobs at the same time. For a set $A' \subseteq A$ of arcs let $z_{A'}$ denote the max flow in the network with arc set $A \setminus A'$. Then scheduling all jobs at the same time is always optimal iff
$$\forall A_1, A_2 \subseteq A \qquad A_1 \cap A_2 = \varnothing \implies z_\varnothing + z_{A_1\cup A_2} \geqslant z_{A_1} + z_{A_2}. \tag{6}$$
The if part follows, since if the implication is true, and we are given a solution scheduling jobs at times $t_1 \neq t_2$, we can always shift all the jobs scheduled at t_2 to t_1 without decreasing the objective function. Conversely, if there are disjoint arc sets A_1 and A_2 with $z_\varnothing + z_{A_1\cup A_2} < z_{A_1} + z_{A_2}$, then for an instance with one job on every arc in $A_1 \cup A_2$, it is better to schedule the jobs on A_1 at a different time than the jobs on A_2. Note that the first example of the directed path is a special case of this.

4. Using the characterization (6), we can generalize the path example. Suppose the network $N - s'$ (i.e. the original network without the sink) is a tree, all arcs pointing away from the source, and in the full network precisely the leaves of this tree are connected to the sink. Assume also that there are no bottlenecks, i.e. for every node $v \neq s$ the capacity of the

arc entering v is at least as large as the sum of the capacities of the arcs leaving v. Under these conditions (6) is satisfied, so freely movable jobs with unit processing times should be scheduled at the same time.

3 Local search for MaxTFFAO

3.1 Evaluating the objective function

We consider a solution of **MaxTFFAO** to be specified by the start time indicator variables y_{jt} for all jobs $j \in J$. For given $\boldsymbol{y}$, the values x_{at} can be fixed by

$$x_{at} = 1 - \max_{j \in J_a} \sum_{t'=\max\{r_j, t-p_j+1\}}^{\min\{t, d_j-p_j+1\}} y_{jt'},$$

and then the best solution for the given $\boldsymbol{y}$ can be determined by solving T max flow problems. As a local search framework requires the frequent evaluation of the objective function, we try to make use of the problem structure to design a more efficient method. The following four simple observations indicate potential for such an improvement.

1. A time interval with constant network structure requires only a single max flow computation.
2. If there is a change in the network between time t and time $t+1$, the solution for t can be used as a warm start for $t+1$. As a consequence, the objective function can be evaluated by solving at most $2|J|+1$ max flow problems, and if this number is really necessary, consecutive networks differ in exactly one arc.
3. To update the flows after a change of the schedule we can restrict our attention to the time intervals where the network structure actually changed due to the modification. That means the effect of local changes in the schedule can be determined by solving a short sequence of max flow problems on very similar networks.
4. How the similarity of the networks for different time periods can be used depends on the way the max flow problems are solved. In our LP based implementation (see discussion in Section 3.2 for details) it is natural to reoptimize from the current solution using the primal simplex method if an arc is added and the dual simplex method if an arc is removed.

To make this more precise we introduce more notation for the start times associated with a solution vector $\boldsymbol{y}$: Let $S_j(\boldsymbol{y})$ be the start time of job j, i.e. $S_j(\boldsymbol{y})$ is the unique t with $y_{jt} = 1$. If there is no danger of confusion we will omit the argument $\boldsymbol{y}$ in the notation and just write S_j. Now we can associate with each solution a set of times

$$R = \{S_j, S_j + p_j \ : \ j \in J\} \cup \{1, T+1\}$$

containing exactly the set of times t such that there is a change of capacity on at least one arc between time $t-1$ and time t. The times $t = 1$ and $t' = T+1$ can be interpreted this way by adding virtual networks at times 0 and $T+1$ with zero capacities. We denote the elements of R by

$$1 = t_0 < t_1 < \cdots < t_{M-1} < t_M = T+1.$$

The set $[t_{i-1}, t_i - 1]$ is called *time slice* i and its length is denoted by $l_i = t_i - t_{i-1}$. The time slicing is illustrated in Figure 4. In this setup the above observations imply that the objective function can be evaluated as described in Algorithm 1.

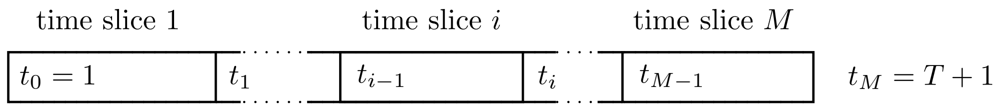


Figure 4: Time slicing.

Algorithm 1 Objective evaluation.

```
Input: Schedule given by $S_j$ for $j \in J$
$R = \{S_j, S_j + p_j \ : \ j \in J\} \cup \{1, T+1\} = \{1 = t_0 < t_1 < \cdots < t_{M-1} < t_M = T+1\}$
Construct the network $(N, A, s, s', u)$
for $i = 1$ to $M$ do
    Update upper bounds of the flow variables according to the outages in time slice $i$
    (Re)solve the network flow problem and store the max flow $z_i$
Output: $z = \sum_{i=1}^{M} z_i \cdot l_i$
```

3.2 Moving single jobs

The feasible region is the set of all binary vectors $\boldsymbol{y} = (y_{jt})_{j \in J, r_j \leqslant t \leqslant d_j}$ satisfying (4). Note that the generation of an initial solution is easy, as we can choose arbitrary start times in the corresponding time windows. A simple neighbourhood is one that is induced by single job movements:

$$N_1(\boldsymbol{y}) = \left\{\boldsymbol{y'} \ : \ S_j(\boldsymbol{y'}) \neq S_j(\boldsymbol{y}) \text{ for at most one job } j\right\}.$$

The size of this neigbourhood is

$$|N_1(\boldsymbol{y})| = 1 + \sum_{j \in J} (d_j - p_j - r_j + 1).$$

In the following we give a characterization of the optimal neighbours, implying an exact method to determine an optimal neighbour.

3.2.1 Preliminary considerations

Moving a job from its current start time S_j to another start time S'_j has two different effects:

1. For any time $t \in [S_j, S_j + p_j - 1] \setminus [S'_j, S'_j + p_j - 1]$ the arc a_j is released and we gain capacity on this arc which could lead to an increase in the max flow for time t.
2. For any time $t \in [S'_j, S'_j + p_j - 1] \setminus [S_j, S_j + p_j - 1]$ we lose the arc, and if it has positive flow in the current max flow, the max flow for time t might decrease.

In order to characterize the impact of a job movement on the objective value we introduce the following parameters measuring the effect of changing the availability status of arc a in time slice i for all $a \in A$ and $i \in [M]$:

- z^+_{ai} is the max flow in the network of time slice i, with arc a added (with capacity u_a) if it is missing in the current solution.

- z^-_{ai} is the max flow in the network of time slice i, with arc a removed if it is present in the current solution.

We start with some simple observations.

- $z^-_{ai} \leqslant z_i \leqslant z^+_{ai}$ for all $a \in A$ and $i \in [M]$.
- $x_{at} = 1$ for $t \in [t_{i-1}, t_i - 1] \implies z^+_{ai} = z_i$.
- $x_{at} = 0$ for $t \in [t_{i-1}, t_i - 1] \implies z^-_{ai} = z_i$.
- For an unavailable arc a (i.e. $x_{at} = 0$ for $t \in [t_{i-1}, t_i - 1]$), releasing arc a increases the max flow by $\Delta^+_{ai} := z^+_{ai} - z_i$.
- For an available arc a (i.e. $x_{at} = 1$ for $t \in [t_{i-1}, t_i - 1]$), removing arc a decreases the max flow by $\Delta^-_{ai} := z_i - z^-_{ai}$.

To efficiently calculate the net effect on the objective, $\Delta_j(S'_j)$, of moving a job j from start time S_j to start time S'_j, one need only consider the set of time slices $\tau^+_j(S'_j)$, defined to be those which are covered by $[S_j, S_j + p_j - 1]$ and that will be (at least partially) uncovered by the move, and the set of time slices $\tau^-_j(S'_j)$, defined to be those which are not covered by $[S_j, S_j + p_j - 1]$ but that will be (at least partially) covered by $[S'_j, S'_j + p_j - 1]$. We also need for each $i \in \tau^+_j(S'_j) \cup \tau^-_j(S'_j)$, the length of the time slice covered by $[S'_j, S'_j + p_j - 1]$, denoted by $l^-_{ij}(S'_j)$. Then

$$\Delta_j(S'_j) = \sum_{i \in \tau^+_j(S'_j)} \Delta^+_{ai} \cdot (l_i - l^-_{ij}(S'_j)) - \sum_{i \in \tau^-_j(S'_j)} \Delta^-_{ai} \cdot l^-_{ij}(S'_j). \tag{7}$$

Provided Δ^+_{ai} and Δ^-_{ai} have been calculated for the appropriate time slices, it is thus straightforward to calculate $\Delta_j(S'_j)$ for any j and S'_j, and hence to determine an optimal neighbour.

Proposition 2. *Finding an optimal neighbour of the given schedule $(S_j)_{j \in J}$ is equivalent to*

$$\max \left\{ \Delta_j(S'_j) \ : \ j \in J, \ S'_j \in [r_j, d_j - p_j + 1] \right\}.$$

If $\Delta_j(S'_j) \leqslant 0$ for all pairs (j, S'_j), there is no improving solution in the neighbourhood of the current schedule.

3.2.2 The basic method

Proposition 2 immediately suggests a local search strategy: compute $\Delta_j(S'_j)$ for (a subset of) all pairs (j, S'_j), choose one or more pairs with a high value of this bound, perform the corresponding changes of the schedule, and iterate. This could be done naively by first calculating Δ^+_{ai} and Δ^-_{ai} for each time slice i and arc a. The formula (7) shows that we can then easily calculate $\Delta_j(S'_j)$ as required. This approach would appear at first sight to be computationally prohibitive, requiring the solution of two max flow problems to calculate z^+_{ai} and z^-_{ai} for each arc a and time slice i. A number of mitigating factors make this approach more attractive than appearances suggest. First, each arc in a given time slice is either missing or present in the current solution, and so from observations in the previous section, one of z^+_{ai} and z^-_{ai} is given by z_i; it is only for the other that a max flow problem needs to be solved. More importantly,

1. if an arc is added in a time slice where it was previously blocked, the flow stays primal feasible but might no longer be optimal, and

2. similarly, if an arc with nonzero flow is taken out, the dual stays feasible.

Thus the maximum flow problems to be solved in calculating z^+_{ai} and z^-_{ai} can use a primal (dual) method respectively "hot started" from the existing solution for the time slice i. We also observe from (7) that only jobs j with $\Delta^+_{a_j i} > 0$ for some time slices i covered by $[S_j, S_j + p_j - 1]$ can be moved to give a better solution: these are the *promising* jobs. So we should first determine Δ^+_{ai} to discover the promising jobs, and then only calculate Δ^-_{ai} values as needed for these jobs. Furthermore, Δ^+_{ai} can only be positive if the reduced cost of arc a in the maximum flow problem is positive; otherwise it must be zero. Thus even if the arc is missing from the network, as long as it is included in the original max flow calculation (with zero capacity), and we use a max flow method which makes reduced costs available, we can avoid further max flow calculations (z^+_{ai} can simply be set to z_i if the reduced cost of a in time slice i is not positive).

Algorithm 2 describes how the effects Δ^+_{ai} (adding an arc) and Δ^-_{ai} (blocking an arc) are determined. We do not make explicit here how z^+_{ai} and z^-_{ai} are calculated, since these depend on the specific max flow method used; these implementation issues are discussed in Section 4.1.2. Finally,

Algorithm 2 Effects of change.

```
PromisingJobs= $\varnothing$
for $i = 1$ to $M$ do
    $A_i^{\text{out}} = \{a \in A \ : \ x_{at} = 0 \text{ for } t \in [t_{i-1}, t_i - 1]\}$
    for $a \in A_i^{\text{out}}$ do
        $\Delta^+_{ai} = z^+_{ai} - z_i$
        if $\Delta^+_{ai} > 0$ then
            Add the job $j$ with $a_j = a$ and time window containing slice $i$ to PromisingJobs
for $j \in$ PromisingJobs do
    Put $i_0 = \min\{i \ : \ t_i \geqslant r_j\}$ and $i_1 = \max\{i \ : \ t_i \leqslant d_j + 1\} - 1$
    for $i = i_0$ to $i_1$ do $\Delta^-_{a_j i} = z_i - z^-_{ai}$
Output: $\Delta^+_{ai}$ – benefit (per time unit) of releasing arc $a$ in time slice $i$
        $\Delta^-_{ai}$ – loss (per time unit) of removing arc $a$ in time slice $i$
        PromisingJobs – set of jobs whose movement could give an improvement
```

Algorithm 3 describes the complete procedure of the greedy rescheduling algorithm which will be denoted by `GreedyResched`. In our implementation we use the following three stopping criteria:

Algorithm 3 `GreedyResched`.

```
Initialize time slicing and flow problems (Algorithm 1)
while not STOP do
    Determine PromisingJobs and the values $\Delta^+_{ai}$ and $\Delta^-_{ai}$ (Algorithm 2)
    for $j \in$ PromisingJobs and $S'_j \in [r_j, d_j - p_j + 1]$ do calculate $\Delta_j(S'_j)$
    if $\max_{j,S'_j} \Delta_j(S'_j) < 0$ then STOP
    else
        Choose $(j, S'_j)$ with maximal $\Delta_j(S'_j)$
        Update time slicing and and resolve the max flow problems with changed input data
```

1. a time limit,
2. 100 iterations without improvement, and
3. 2 consecutive iterations without improvement and only a single pair (j, S'_j) with $\Delta_j(S'_j) = 0$.

The reason for the last criterion is that in this situation the algorithm just alternates between two solutions having the same objective value.

3.2.3 Variations

Here we present some natural modifications of the algorithm `GreedyResched`.

Randomization. Instead of choosing the best neighbour in each iteration one can choose randomly from a set of candidates, similar to the strategy applied in the construction phase of *greedy randomized adaptive search procedures* [16]. More precisely, we order the pairs (j, S'_j) by nonincreasing value of $\Delta_j(S'_j)$ and choose randomly from the first k of this list (uniformly distributed), where k depends on the total number of possibilities, for instance with $K = \#\{(j, S'_j) \ : \ \Delta_j(S'_j) \geqslant 0\}$ denoting the number of moves with nondecreasing objective value we can take

$$k = \max\left\{\min\left\{\kappa_1, K\right\}, \lceil \kappa_2 K \rceil\right\},$$

where $\kappa_1 \in \mathbb{N}$ and $\kappa_2 \in \{\kappa \in \mathbb{R} \ : \ 0 \leqslant \kappa \leqslant 1\}$ are parameters of the algorithm. After satisfying the stopping criterion, we can restart the algorithm from the initial solution, iterate this, and finally choose the best solution from all runs. We denote this randomized variant by `GreedyRandResched`(κ_1, κ_2). In Figure 5 we plot the behaviour of K in `GreedyResched` for the randomly generated instances we used in our computational experiments (see Section 4). For these experiments we choose $(\kappa_1, \kappa_2) = (5,\ 0.15)$. Some further possible modifications to randomization

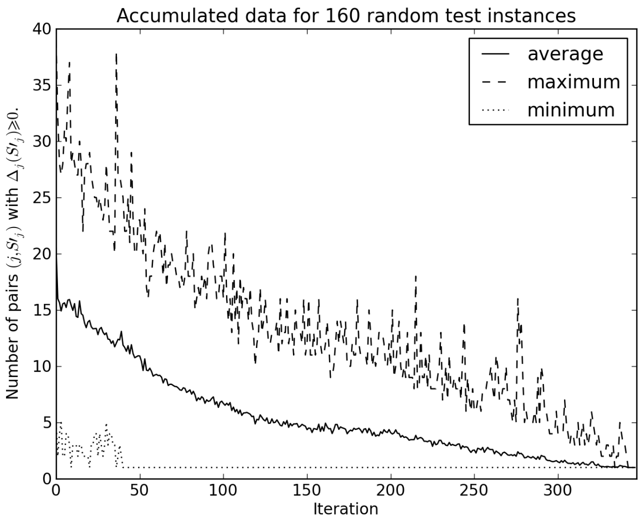


Figure 5: Number of possible moves, i.e. pairs (j, S'_j) with $\Delta_j(S'_j) \geqslant 0$.

are as follows.

1. Instead of going back to the initial solution each time the stopping criterion is met, we can collect the intermediate solutions with a large value of K (indicating many improving directions) in a solution pool, and choose the starting point for each run from the solution pool (randomly or deterministically).
2. If the computation time until reaching the stopping criterion is large the following combination of the ideas underlying `GreedyResched` and `GreedyRandResched` might be beneficial.
   (a) Do a small number, say k_1, improvements randomly choosing from the improving moves (as in `GreedyRandResched`).
   (b) Repeat the step (a) a small number, say k_2, times.
   (c) Choose the best of the k_2 solutions obtained and continue with step (a).

Testing the effectiveness of these further ideas will be the subject of future research.

Making several moves at a time. In order to speed up the progress of the method we can do several moves corresponding to pairs (j, S'_j) with nonnegative value of $\Delta_j(S'_j)$ simultaneously, if they do not affect the same time slices. This can be implemented by looping over the list of pairs (j, S'_j), ordered by nonincreasing $\Delta_j(S'_j)$, and choosing a pair if its affected time slices do not overlap with those of the pairs already chosen. The benefit of this approach is that it saves recalculations of the values $\Delta_j(S'_j)$, which may be relatively expensive. An iteration of this algorithm can be considered as a greedy accumulation of `GreedyResched` steps, and so we denote the algorithm by `GreedyAccResched`. We also consider a randomized version of this approach. While the list of pairs (j, S'_j), ordered by nonincreasing $\Delta_j(S'_j)$, is non-empty, we choose at random a pair from the first k in the list, and then remove from the list all pairs with affected time slices overlapping those of the chosen pair, before looping again to choose a random pair from the first k. We call this the `GreedyRandAccResched` algorithm.

3.3 Moving multiple jobs

Clearly, there are some limitations in the approach to consider only movements of single jobs. It is easy to construct examples where no single job movement yields an improvement, but moving two jobs at the same time does. However, the benefit of moving jobs simultaneously is only of interest if the jobs interact, in the sense of overlapping in time. We thus propose to search neighbourhoods of the form:

$$\tilde{N}_{j_0}(\boldsymbol{y}) = \left\{\boldsymbol{y'} \ : \ S_j(\boldsymbol{y'}) \neq S_j(\boldsymbol{y}) \text{ only for jobs } j \in J(j_0)\right\},$$

for some $j_0 \in J$, where $J(j_0)$ is the set of jobs whose time window (plus processing time) overlaps with that of j_0, i.e.

$$J(j_0) = \{j \in J \ : \ [r_j, d_j] \cap [r_{j_0}, d_{j_0}] \neq \varnothing\}.$$

The size of $\tilde{N}_{j_0}(\boldsymbol{y})$ is $\prod_{j \in J(j_0)}(d_j - p_j - r_j + 2)$. This is exponential in the number of jobs that have at least two possible start times and overlap with j_0. In particular, the instance used in the proof of Proposition 1 has the property that all job pairs overlap. Thus in general it is NP-hard to optimize over this neighbourhood, and we propose to explore it via a randomized approach as follows.

We consider each job in turn as the *base job*, j_0, and systematically search a selection $C(j_0) \subseteq [r_{j_0}, d_{j_0}]$ of its possible start times. Our idea is that $C(j_0)$ should start small, allowing a "rough" exploration of the alternatives, and increase as the algorithm progresses, thus refining the search. We explain this more precisely later. For each possible start time $S'_{j_0} \in C(j_0)$, we would like to know how "good" that choice of start time is, taking into account interactions of j_0 with other jobs, i.e. we would like to find the best $\boldsymbol{y}'$ such that $S_j(\boldsymbol{y}') \neq S_j(\boldsymbol{y})$ only for jobs $j \in J(j_0)$. Equivalently, we would like to simultaneously optimize the start times of all jobs in $J(j_0)$, finding a local optimum with respect to j_0. However we expect that doing so would be prohibitive in terms of computational time. Thus we sample from a restricted neighbourhood, restricting the possible start times of jobs in $J(j_0) \setminus \{j_0\}$ heuristically, using the intuition that jobs should either overlap as little, or as much, as possible to get best results. To see where this intuition comes from consider two arcs a and a' with the property that every source-sink path through a also contains a'. If these are the only two arcs with maintenance jobs it is clearly best possible to maximize the overlap between jobs on these arcs. On the other hand, if there is no path containing both arcs a and a', then the total throughput is maximized when the jobs overlap is minimized. Each $j \in J(j_0) \setminus \{j_0\}$ has a set of (up to four) possible start times $C(j)$, so that either the job's start or end aligns with the start or end of job j_0, (assuming j_0 starts at S'_{j_0}). This choice of $C(j)$ is motivated by the fact that in general, there is always an optimal solution such that for every job j one of the following is true.

- Job j starts at its earliest possible start time r_j, or
- job j starts at its latest possible start time $d_j - p_j + 1$, or
- there is a job j' that starts at the same time $S_j = S_{j'}$, or
- there is a job j' that ends at the same time $S_j + p_j - 1 = S_{j'} + p_{j'} - 1$, or
- there is a job j' such that the start of job j aligns with the end of job j', i.e. $S_j = S_{j'} + p_{j'}$, or
- there is a job j' such that the end of job j aligns with the start of job j', i.e. $S_j + p_j = S_{j'}$.

We simply sample randomly from the neighbourhood σ times, choosing the best, for σ an algorithm parameter. This randomized method for moving multiple jobs, denoted by `RandMultiJob`, is described more formally in Algorithm 4.

To implement the method the choice of the candidate start sets $C(j_0)$ has to be specified. For our experiments, $C(j_0)$ consists of k evenly spaced elements in the interval $[r_j, d_j]$, where $k \in \mathbb{N}$. k starts small (at $k = 1$), and increases by one whenever no improvement has been found for a number of consecutive iterations. In our experiments, we use $|J|$ iterations for the increment criterion. Since each job may be the base job multiple times for the same value of k, we want to avoid choosing the same subset of start times every time. Thus we include a mechanism for cycling through sets of k evenly spaced points, modulo the time window width. More precisely, in the m-th run through the outer loop of Algorithm 4, we put

- $W = d_{j_0} - p_{j_0} - r_{j_0} + 2$ (width of the time window of job j_0),
- $\theta = \lfloor (W-1)/k \rfloor$, and
- $C(j_0) = \begin{cases} \{r_{j_0} + (m + i\theta) \pmod W \ : \ i = 0, \dots, k-1\} & \text{if } \theta > 1, \\ [r_{j_0}, d_{j_0} + p_{j_0} + 1] & \text{if } \theta = 1. \end{cases}$

Note that W and θ vary with j_0, but we forgo using a j_0 subscript to improve readability. To illustrate how this works, consider the case that $W = 7$, $k = 3$ and take $r_{j_0} = 1$. Then $\theta = 2$ and when $m \equiv 0 \pmod W$ we get $C(j_0) = \{1, 3, 5\}$, when $m \equiv 1 \pmod W$ we get $C(j_0) = \{2, 4, 6\}$,

Algorithm 4 `RandMultiJob`.

```
Input: A feasible solution $(S_j)_{j\in J}$ with objective value $z$ and parameter $\sigma$
while not Stop do
    for $j_0 \in J$ do
        choose a subset $C(j_0) \subseteq [r_{j_0}, d_{j_0} - p_{j_0} + 1]$
        $J(j_0) = \{j \in J \ : \ [r_j, d_j] \cap [r_{j_0}, d_{j_0}] \neq \varnothing\}$
        Put $\boldsymbol{S} = (S_j)_{j\in J(j_0)}$
        for $S'_{j_0} \in C(j_0)$
            for $j \in J(j_0) \setminus \{j_0\}$ do
                set $C(j) = [r_j, d_j - p_j + 1] \cap \left\{S'_{j_0}, S'_{j_0} - p_j, S'_{j_0} + p_{j_0}, S'_{j_0} + p_{j_0} - p_j + 1\right\}$
            repeat
                for $j \in J(j_0) \setminus \{j_0\}$ do
                    if $C(j) \neq \varnothing$ do choose random $S'_j$ from $C(j)$   else $S'_j = S_j$
                compute the objective $z'$ for starting job $j$ at time $S'_j$ for all $j \in J(j_0)$
                if $z' > z$ then replace $\boldsymbol{S}$ by $(S'_j)_{j\in J(j_0)}$ and $z$ by $z'$
            until done $\sigma$ times

            if enough consecutive iterations with no improvement have passed then increase $k$
Output: An improved solution $(S_j)_{j\in J}$
```

when $m \equiv 2 \pmod{W}$ we get $C(j_0) = \{3, 5, 7\}$, when $m \equiv 3 \pmod{W}$ we get $C(j_0) = \{1, 4, 6\}$, etc.

In future work, we will consider allowing σ to vary during the course of the algorithm, by making it dependent on the size of the neighbourhood $\tilde{N}_{j_0}(\mathbf{y})$ at the current solution $\mathbf{y}$, so that more samples are taken from larger neighbourhoods.

4 Computational Experiments

In this section we report on the results of computational tests of our proposed algorithm variants. The first subsection is concerned with randomly generated instances, while the second subsection contains results for two instances coming from real world data.

4.1 Randomly generated instances

We first describe how our random test instances have been generated, then we present the details of the experiments that have been run, and finally, we compare the performance of the considered algorithms.

4.1.1 Instance generation

Our tests are carried out for a time horizon with $T = 1,000$. We need to generate networks and job lists. We generate eight different networks using the RMFGEN generator of Goldfarb and Grigoriadis [7]. For parameters a, b, c_1 and c_2 the generated network has a^2b nodes arranged in b frames of a^2 nodes each. The capacities between frames are randomly chosen from $[c_1, c_2]$, while all capacities inside frames are $c_2 a^2$. We generated 8 different networks for the parameter pairs

$$(a,b) \in \big\{(2,3),\ (2,4),\ (3,2),\ (3,3),\ (3,4),\ (4,2),\ (4,3),\ (4,4)\big\}$$

with $c_1 = 10$ and $c_2 = 20$.

In order to generate a job list for a given network, for each arc we first choose α, the number of jobs. Then we divide the time horizon into α equal subintervals, each of them associated with one of the jobs to be created. For each job we choose a processing time and a number of start time candidates randomly. Finally, we choose a random release date, making sure that it is compatible with the job being completed in its subinterval. This is made more precise in Algorithm 5 where the input parameters are

- X — set of possible number of jobs per arc,
- Y — set of possible processing times, and
- Z — set of possible sizes for start time windows.

Algorithm 5 Generate JobList (X, Y, Z) $(X, Y, Z \subseteq \mathbb{N})$

```
for $a \in A$ do
  Initialize $J_a = \varnothing$
  choose random $\alpha \in X$ and put $\mu = \lfloor T/\alpha \rfloor$
  for $\eta = 1$ to $\alpha$ do
    choose random $\beta \in Y$ and $\gamma \in Z$
    choose random $r \in \{1, 2, \dots, \mu - \beta - \gamma + 2\}$
    add job with processing time $\beta$, release date $r_j = (\eta - 1)\mu + r$
    and deadline $r_j + \beta + \gamma - 2$ to $J_a$
```

Let $\overline{\alpha}$, $\overline{\beta}$ and $\overline{\gamma}$ be the maximum elements of X, Y and Z, respectively. In order to guarantee feasible job lists we must have

$$(\overline{\gamma} + \overline{\beta} - 1)\overline{\alpha} \leqslant T.$$

As the number of binary variables in the MIP model (1)–(5) is determined by the sizes of the time windows we decided to focus on studying the influence of the set Z. So we fix $X = [5, 15]$, $Y = [10, 30]$ and test two variants for Z.

1. There are a variety of time window sizes: $Z_1 = [1, 35]$ (the *first* instance set).
2. All time windows are large: $Z_2 = [25, 35]$ (the *second* instance set).

For each network and each triple (X, Y, Z_i) we generated 10 instances, giving a total of 160 instances. The network sizes and the average numbers of jobs obtained in this way are shown in Table 2. In Tables 3 and 4 we report the average problem sizes for the MIP formulation (1)–(5).

4.1.2 Experimental setup

Each of the generated instances is solved by the following methods:

Algorithm CPX. CPLEX with default settings applied to the formulation (1)–(5),

Small networks				Large networks			
Network	Nodes	Arcs	Jobs	Network	Nodes	Arcs	jobs
1	12	32	303.2	5	36	123	1159.8
2	16	44	421.0	6	32	92	870.7
3	18	57	542.4	7	48	176	1674.2
4	27	90	847.5	8	64	240	2278.0

Table 2: The sizes of the random networks.

Network	# Rows	# Columns	# Nonzeros	# Binaries	Root relaxation solution time (s)
1	50,810	69,925	226,402	36,925	0.5
2	69,714	95,381	312,255	50,381	1.4
3	87,589	122,784	404,722	64,784	1.6
4	137,141	192,544	638,215	101,544	7.0
5	187,607	262,799	886,817	138,799	21.4
6	145,025	197,037	658,174	104,037	5.5
7	265,983	375,552	1,278,099	198,552	37.8
8	361,293	511,157	1,746,054	270,157	92.1

Table 3: Average problem sizes for the first instance set ($Z = [1, 35]$).

Algorithm GR. `GreedyResched` using CPLEX to solve the max flow subproblems,

Algorithm GRR. `GreedyRandResched` (Section 3.2.3) with parameters $(\kappa_1, \kappa_2) = (5, 0.15)$,

Algorithm GAR. `GreedyAccResched` (Section 3.2.3),

Algorithm GRAR. `GreedyRandAccResched` (Section 3.2.3) with the same parameter values as for GRR, and

Algorithm RMJ σ. `RandMultiJob` with parameter σ.

For ease of implementation, and so as to more readily exploit reduced cost information, and access primal and dual algorithm variants, we decided to solve the max flow subproblems in the algorithms `GreedyResched`, `GreedyRandResched`, `GreedyAccResched` and `GreedyRandAccResched` using CPLEX as LP solver instead of implementing combinatorial algorithms. The following three remarks on the implementation of Algorithm 2, which underlies the three greedy approaches, are

Network	# Rows	# Columns	# Nonzeros	# Binaries	Root relaxation solution time (s)
1	57,763	75,215	322,204	42,215	1.1
2	79,481	102,783	448,435	57,783	5.8
3	100,483	132,480	583,899	74,480	1.4
4	157,521	207,922	920,092	116,922	33.6
5	214,415	283,148	1,257,642	159,148	427.8
6	165,634	212,561	944,456	119,561	204.6
7	303,721	404,331	1,800,871	227,331	207.5
8	413,640	550,940	2,469,028	309,940	828.7

Table 4: Average problem sizes for the second instance set ($Z = [25, 35]$).

based on the observations in Section 3.2.2.

1. The value Δ^{+}_{ai} is computed only if the reduced cost of the arc a is positive in the current solution for time slice i as otherwise there is no potential for improvement.
2. For calculating the values z^{+}_{ai}, i.e. the gain obtained by adding in arc a, we use the primal simplex method starting from the current max flow for time slice i.
3. For calculating the values z^{-}_{ai}, i.e. the loss due to taking out arc a, we use the dual simplex method starting from the current max flow for time slice i.

For `GreedyResched`, we also note that among the pairs (j, S'_j) with maximal value of $\Delta_j(S'_j)$ it is always possible to choose one causing at most one time slice split. A simple way to achieve this is to choose the pair with the smallest S'_j: This ensures that S'_j or $S'_j + p_j$ is one of the breakpoints t_i in the current time slicing. We use this approach in our implementation. For `RandMultiJob` we solve the max flow problems using the implementation of the push-relabel algorithm in the Boost library [17]. Here we don't take advantage of the similarities between the networks of different time slices, but we still use the third observation in Section 3.1 that after changing the schedule we only have to reevaluate the flow for time slices that are actually affected by the change. We experimented with $\sigma = 1, 2, 4$ and 8. We found that results for $\sigma = 8$ were dominated by the other values, and that whilst $\sigma = 4$ did give better values than $\sigma = 1$ or 2 on a very small proportion of instances, it did not give the best value over all algorithms for any instance (random or real world). Thus we only present detailed results for $\sigma = 1$ and 2.

For all algorithms, we impose a time limit of 30 minutes, and all of them start with an initial solution given by

$$S_j = \left\lfloor \frac{r_j + d_j}{2} \right\rfloor .$$

All computations are done on a Dell PowerEdge 2950 with dual quad core 3.16GHz Intel Xeon X5460 processors and 64GB of RAM running Red Hat Enterprise Linux 5. CPLEX v12.1 was running in deterministic mode using a single thread.

4.1.3 Results

As a performance measure to compare algorithms we use the relative gap between the algorithm's solution value and the best known solution value over all algorithms. If the best known solution value for an instance I is z^{best} and the current algorithm returns z, its performance measure on that instance is given by

$$\mathcal{P}(I) = \frac{z^{\text{best}} - z}{z^{\text{best}}}.$$

In Figures 6 to 9, we plot for CPLEX and the `Greedy` algorithms the proportion of instances for which the solution found by the algorithm is within a factor of $1 - \tau$ of the best, for increasing values of τ, i.e. we plot

$$\frac{1}{n} \cdot \#\{I \ : \ I \text{ instance with } \mathcal{P}(I) \leqslant \tau\}$$

as a function of τ, where n is the total number of instances (in our case 80 for each instance set). Note that for the 5 minute plots (Figures 6 and 8) we take z^{best} to be the best known solution over all algorithms after 30 minutes. Tables 5 and 6 contain the average number of max flow problems solved for each of the local search algorithms and every network. For algorithms GR and GAR we also report the run times, GRR, GRAR and RMJ 2 ran for the whole 30 minutes. Tables 7 to 10

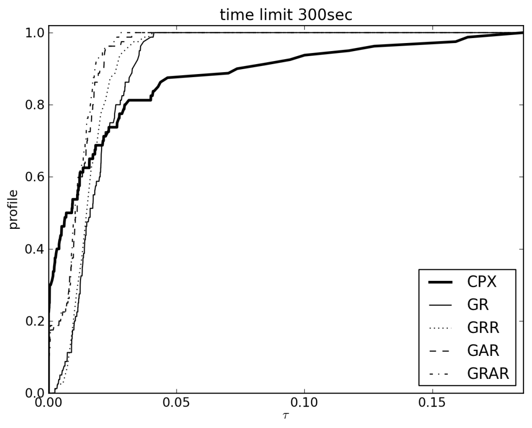


Figure 6: Performance profiles for the first instance set ($Z = [1, 35]$) with computation time limited to 5 minutes.

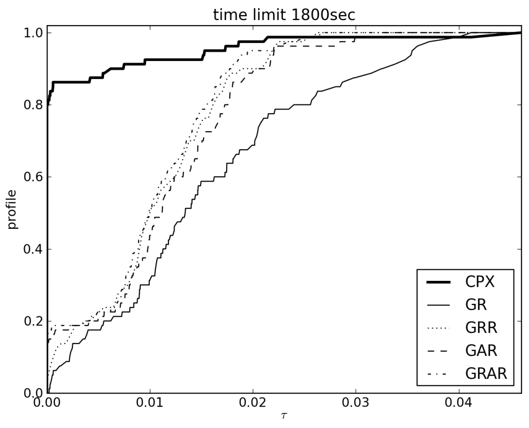


Figure 7: Performance profiles for the first instance set ($Z = [1, 35]$) with computation time limited to 30 minutes.

	GR		GRR	GAR		GRAR	RMJ 2
Network	Time(s)	mf calls	mf calls	Time(s)	mf calls	mf calls	mf calls
1	12	210	31,296	17	179	16,500	96,618
2	25	351	22,741	30	287	13,772	74,739
3	25	460	29,168	40	344	16,386	61,270
4	128	1,256	15,201	100	677	9,681	38,703
5	308	2,007	10,693	324	1,247	7,320	25,605
6	115	838	11,574	115	585	7,041	37,588
7	524	2,984	8,871	519	2,010	6,146	14,827
8	825	3,256	6,607	605	1,540	4,090	8,276

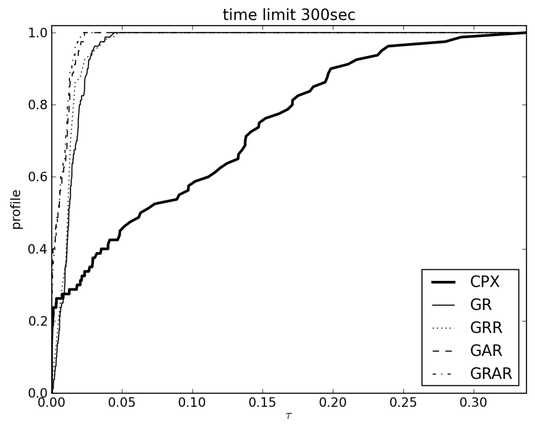
Table 5: Average numbers of max flow problems (divided by 1,000) for the first instance set ($Z = [1, 35]$).

provide information about the relative gaps (average and maximal) and the numbers of instances where each algorithm found the best solution, for all algorithms. Here the relative gap is computed as $(z' - z)/z'$ where z' is the best upper bound obtained by CPLEX in 30 minutes, and z is the objective value of the best solution found by the considered algorithm in the respective time (5 or 30 minutes).

We make the following observations.

- For the first instance set CPLEX outperforms all other algorithms, but on the large networks the heuristics, in particular `GreedyAccResched` and `GreedyRandAccResched`, arequite good in providing a reasonably good solution in a short time. The 5 minute performance profiles both show the distinct advantage of `GreedyAccResched` and `GreedyRandAccResched` over the other methods for short run times. For long run times, the 30 minute performance profiles show CPLEX to be the clear winner for the first instance set, with `GreedyRandResched` and `GreedyRandAccResched` best for the second instance set.

- For the second instance set, on the small networks CPLEX is still superior, but on the larger



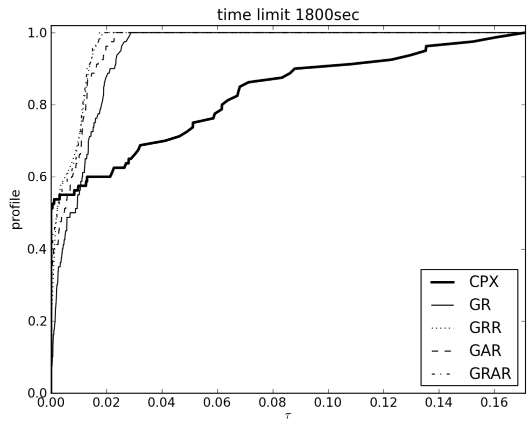


Figure 8: Performance profiles for the second instance set ($Z = [25, 35]$) with computation time limited to 5 minutes.

Figure 9: Performance profiles for the second instance set ($Z = [25, 35]$) with computation time limited to 30 minutes.

	GR		GRR	GAR		GRAR	RMJ 2
Network	time [sec]	mf calls	mf calls	time [sec]	mf calls	mf calls	mf calls
1	14	321	37,818	12	214	30,966	83,170
2	36	627	28,411	23	382	25,900	65,355
3	28	724	38,786	26	490	32,253	54,133
4	160	1,982	19,304	84	1,129	19,702	35,745
5	367	2,928	13,907	229	1,966	7,793	23,858
6	197	1,762	15,022	96	776	7,603	33,656
7	709	4,704	11,133	499	3,485	5,714	14,388
8	956	4,723	8,147	536	2,334	3,808	8,056

Table 6: Average numbers of max flow problems (divided by 1000) for the second instance set ($Z = [25, 35]$).

networks, the local search heuristics outperform CPLEX, with all heuristics giving solutions with smaller gaps on average for all (large) instances over short run times, and all `Greedy` heuristics giving smaller gaps on average over long run times – significantly smaller for 3 out of the 4 (largest) networks.

- Comparing `GreedyResched` and `GreedyAccResched` we see that in all cases it pays off to save the time for reevaluating the possible moves after each step and thus being able to make more moves in the same amount of time. A similar observation applies to `GreedyRandResched` and `GreedyRandAccResched`, but the benefits of the latter are less pronounced.
- Across the board, randomized greedy algorithms give better results than their non-random counterparts, due to the possibility to escape local minima.
- `RandMultiJob` performs better with $\sigma = 1$ than 2, particularly for larger networks in the second instance set, and for the first instance set, with the shorter run time. On the first instance set with the longer run time, the two are difficult to separate, but $\sigma = 2$ gives

		1	2	3	4	5	6	7	8
CPX	avg gap	0.0	0.4	0.3	1.1	3.0	3.7	3.9	12.9
	max gap	0.1	0.8	1.7	2.6	6.1	4.6	5.4	20.2
	# best sol	10	10	9	10	7	10	8	3
GR	avg gap	2.1	2.8	1.9	1.6	2.3	4.9	2.1	3.3
	max gap	4.1	4.0	3.5	1.9	2.5	5.5	2.7	5.3
	# best sol	0	0	0	0	0	0	0	0
GRR	avg gap	1.5	2.0	1.6	1.5	2.4	4.1	2.4	3.7
	max gap	2.2	3.1	2.4	2.0	2.7	5.2	3.1	6.1
	# best sol	0	0	0	0	0	0	0	0
GAR	avg gap	1.5	2.0	1.5	1.4	2.1	4.0	1.5	1.5
	max gap	1.9	2.4	2.2	1.6	2.5	4.8	1.9	2.2
	# best sol	0	0	1	0	2	0	2	6
GRAR	avg gap	1.4	1.8	1.4	1.4	2.0	3.8	1.5	1.5
	max gap	1.8	2.2	2.2	1.6	2.4	4.2	1.8	2.2
	# best sol	0	0	1	0	1	0	0	1
RMJ 1	avg gap	3.0	5.1	2.0	4.5	7.3	11.1	5.2	5.7
	max gap	4.9	7.1	2.8	6.9	8.5	12.5	6.5	7.6
	# best sol	0	0	0	0	0	0	0	0
RMJ 2	avg gap	2.5	4.6	1.9	4.8	7.3	11.2	5.3	5.9
	max gap	3.8	6.0	2.6	7.1	8.1	13.2	6.7	8.0
	# best sol	0	0	0	0	0	0	0	0

Table 7: Average and maximal relative gaps and number of best solutions found on the first instance set, $Z = [1, 35]$ (runtime 5 minutes).

better results on more networks, and in particular does better on the difficult case of the sixth network. As might be expected, the `RandMultiJob` algorithms show more significant improvement than the greedy heuristics when given more run time. However in no case do the `RandMultiJob` algorithms outperform the greedy heuristics. (Hence we omit their profiles from Figures 6 to 9, to avoid cluttering them.)

4.2 Instances derived from real world data

The real world maintenance scheduling problem is complicated by additional constraints imposed, for example, by daylight restrictions, availability of equipment or labour force to carry out the maintenance, incompatibility issues between jobs, or conflicts with other users of the infrastructure. All of this can be modelled in an MIP framework and taken into account in a local search, both of which are the subject of ongoing work. For the present paper, we ignore the additional constraints and conduct some experiments on pure **MaxTFFAO** instances derived from real world data. The network shown in Figure 10 is a simplified version of the real situation. We generate two instances using the maintenance job lists and the actual maintenance schedules for 2010 and 2011. These job lists contain $1,457$ and $1,234$ jobs, respectively. Based on the level of detail occurring in practice, we use a time discretization of 1 hour, leading to instances with time horizons $T = 365 \cdot 24 = 8,760$. The processing times vary between an hour and several days, while 75% of the jobs have a

		1	2	3	4	5	6	7	8
	avg gap	0.0	0.2	0.3	0.3	1.6	2.0	0.9	2.8
CPX	max gap	0.0	0.4	1.7	0.8	2.8	2.7	2.1	6.1
	# best sol	10	10	9	10	7	10	8	3
	avg gap	2.1	2.8	1.9	1.6	2.3	4.9	1.6	1.6
GR	max gap	4.1	4.0	3.5	1.9	2.5	5.5	2.1	2.3
	# best sol	0	0	0	0	0	0	0	0
	avg gap	1.4	1.7	1.5	1.4	2.0	3.7	1.4	1.5
GRR	max gap	2.2	2.2	2.3	1.9	2.3	4.9	2.0	2.2
	# best sol	0	0	0	0	0	0	0	0
	avg gap	1.5	2.0	1.5	1.4	2.1	4.0	1.5	1.5
GAR	max gap	1.9	2.4	2.2	1.6	2.5	4.8	1.9	2.2
	# best sol	0	0	1	0	2	0	2	6
	avg gap	1.3	1.7	1.4	1.3	2.0	3.7	1.4	1.5
GRAR	max gap	1.7	2.1	2.2	1.5	2.4	4.2	1.8	2.2
	# best sol	0	0	1	0	1	0	0	1
	avg gap	2.9	5.0	2.0	3.9	6.3	9.9	4.2	4.6
RMJ 1	max gap	4.9	6.9	2.8	5.7	6.8	11.9	5.7	6.2
	# best sol	0	0	0	0	0	0	0	0
	avg gap	2.5	4.5	1.8	3.9	6.3	9.5	4.2	4.6
RMJ 2	max gap	3.8	5.9	2.6	6.1	7.0	11.0	5.0	6.5
	# best sol	0	0	0	0	0	0	0	0

Table 8: Average and maximal relative gaps and number of best solutions found on the first instance set, $Z = [1, 35]$ (runtime 30 minutes).

processing time between 1 and 18 hours. For every job we assume a time window of two weeks, i.e. $d_j = r_j + p_j + 14 \cdot 24 - 2$ for all j. This model leads to really large problems as indicated in Table 11, containing the problem sizes. As a start solution we used a snapshot of the HVCCC maintenance scheduling process. We increased the time limit to 2 hours, and the results are shown in Figures 11 and 12. For clarity, the same results for CPLEX and a selection of the better algorithms is given in Figures 13 and 14.

We observe that the MIP seems to be really hard. For the 2010 data, CPLEX finds one integer solution with better objective value than the start solution, and for 2011 no improving solution can be found at all. Confirming the results for the random instances, the greedy approaches perform very well in terms of finding high quality solutions quickly. The impacts, i.e. the annual capacity reductions due to maintenance, for the start solutions were 37.6Mt (2010) and 32.5 Mt (2011). Table 12 shows the impact reductions achieved by the different algorithms. We note two features that seem to be different to the behaviour for the random instances.

1. Randomization does not always improve the greedy heuristics. Of course, looking at two instances is very limited evidence, but for the 2011 data the randomized variants give slightly worse results. `GreedyAccResched` gives the best result for this instance.

2. `RandMultiJob` keeps improving even after 2 hours, while the other local search strategies seem to get trapped in local optima comparatively early. Both $\sigma = 1$ and 2 values give better

		1	2	3	4	5	6	7	8
	avg gap	0.1	3.9	0.0	6.3	20.5	21.9	16.1	17.0
CPX	max gap	0.4	6.2	0.1	11.0	25.3	39.5	22.8	24.8
	# best sol	10	10	10	9	0	3	0	0
	avg gap	1.8	4.0	1.6	1.9	3.4	8.2	2.5	3.6
GR	max gap	2.5	4.7	2.1	2.8	4.6	10.0	3.4	5.6
	# best sol	0	0	0	0	1	0	0	2
	avg gap	1.4	3.2	1.2	1.9	3.5	7.7	2.6	4.0
GRR	max gap	2.1	4.0	1.4	2.7	4.8	10.8	3.2	5.8
	# best sol	0	0	0	0	1	2	0	1
	avg gap	1.4	3.4	1.2	1.8	2.9	7.7	1.4	1.3
GAR	max gap	2.3	4.5	1.4	2.5	3.5	9.1	1.8	1.9
	# best sol	0	0	0	0	7	3	4	4
	avg gap	1.3	3.0	1.2	1.8	2.8	7.7	1.4	1.4
GRAR	max gap	2.3	3.9	1.4	2.4	3.5	9.4	1.8	1.9
	# best sol	0	0	0	1	1	2	6	3
	avg gap	2.0	6.5	1.4	4.6	8.1	15.3	5.1	4.9
RMJ 1	max gap	3.0	8.2	1.7	8.3	9.3	18.8	5.9	6.6
	# best sol	0	0	0	0	0	0	0	0
	avg gap	1.9	6.5	1.4	4.7	8.2	15.5	5.0	5.0
RMJ 2	max gap	2.8	8.2	1.7	8.7	10.0	18.4	5.9	6.6
	# best sol	0	0	0	0	0	0	0	0

Table 9: Average and maximal relative gaps and number of best solutions found on the second instance set, $Z = [25, 35]$ (runtime 5 minutes).

results than any of the greedy heuristics or CPLEX for the 2010 instance, with $\sigma = 2$ giving the best result overall.

		1	2	3	4	5	6	7	8
	avg gap	0.0	1.9	0.0	1.6	7.4	8.9	6.6	13.3
CPX	max gap	0.1	3.3	0.1	4.9	10.8	14.2	10.1	18.3
	# best sol	10	10	10	9	0	3	0	0
	avg gap	1.8	4.0	1.6	1.9	3.1	8.2	1.5	1.3
GR	max gap	2.5	4.7	2.1	2.8	4.1	10.0	2.0	1.6
	# best sol	0	0	0	0	1	0	0	2
	avg gap	1.3	3.1	1.2	1.8	2.8	7.2	1.4	1.3
GRR	max gap	1.8	3.9	1.4	2.5	3.5	9.0	1.8	1.5
	# best sol	0	0	0	0	1	2	0	1
	avg gap	1.4	3.4	1.2	1.8	2.9	7.7	1.4	1.3
GAR	max gap	2.3	4.5	1.4	2.5	3.5	9.1	1.7	1.9
	# best sol	0	0	0	0	7	3	4	4
	avg gap	1.3	3.0	1.2	1.7	2.8	7.3	1.4	1.3
GRAR	max gap	1.8	3.8	1.4	2.3	3.5	8.5	1.8	1.9
	# best sol	0	0	0	1	1	2	6	3
	avg gap	2.0	6.4	1.4	3.9	7.0	13.9	4.1	4.0
RMJ 1	max gap	3.0	8.2	1.6	7.2	7.9	17.4	5.2	5.8
	# best sol	0	0	0	0	0	0	0	0
	avg gap	1.9	6.2	1.3	3.8	7.1	14.1	4.2	4.0
RMJ 2	max gap	2.8	8.2	1.7	7.0	8.2	17.1	4.9	5.7
	# best sol	0	0	0	0	0	0	0	0

Table 10: Average and maximal relative gaps and number of best solutions found on the second instance set, $Z = [25, 35]$ (runtime 30 minutes).

	# Rows	# Columns	# Nonzeros	# Binaries	Root relaxation solution time (s)
2010	2,741,944	3,317,433	13,500,830	1,775,673	3,823
2011	2,735,919	3,310,352	13,226,945	1,768,592	7,154

Table 11: Problem sizes for the instances derived from real world data.

	2010			2011		
	Impact (Mt)	Reduction (%)	Gap (%)	Impact (Mt)	Reduction (%)	Gap (%)
CPX	28.9	22.9	7.8	32.5	0.0	13.4
GR	26.4	29.6	6.1	19.8	39.0	4.9
GRR	25.6	31.8	5.5	20.3	37.5	5.2
GAR	25.4	32.3	5.4	19.8	39.1	4.9
GRAR	25.0	33.4	5.1	19.9	38.7	4.9
RMJ 1	24.8	33.9	4.9	20.5	36.8	5.4
RMJ 2	24.6	34.6	4.8	20.4	37.3	5.3

Table 12: Impact reduction obtained by the different local search strategies. The column labeled "Gap" contains the relative gap to the best known upper bound from CPLEX.

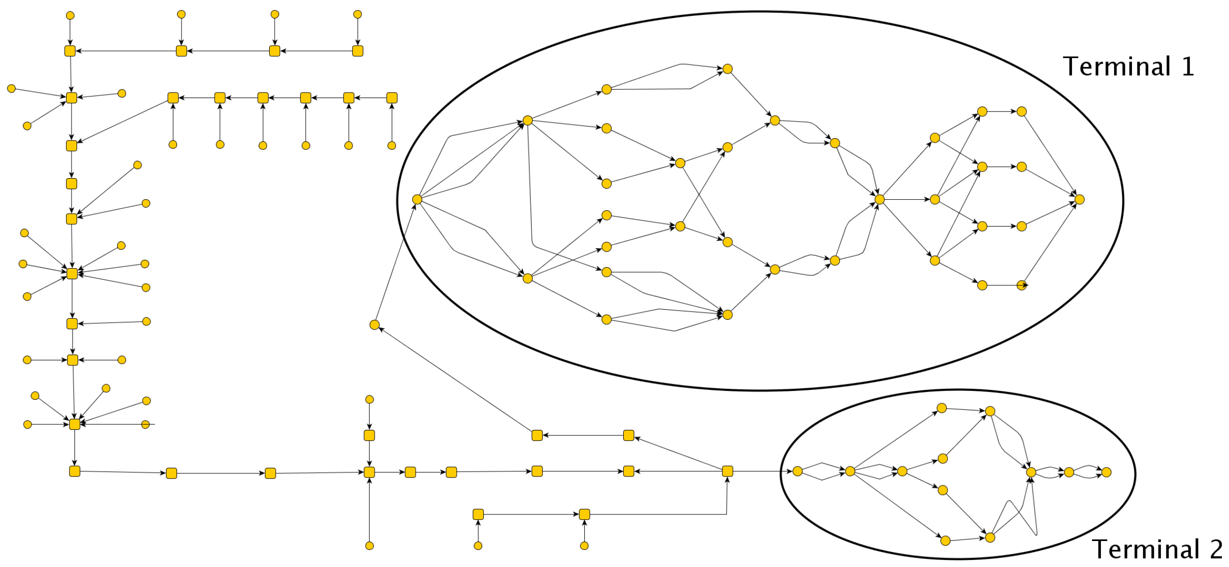


Figure 10: The HVCC network. The circled parts of the network represent the flow of coal through terminal handling equipment. The rest represents the rail network, sourcing coal from 33 coal load points.

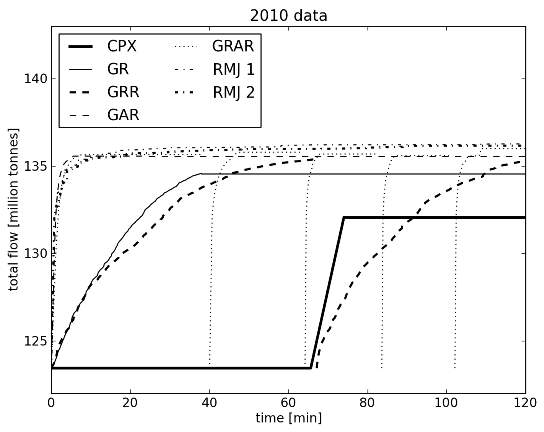


Figure 11: Progress for the 2010 data.

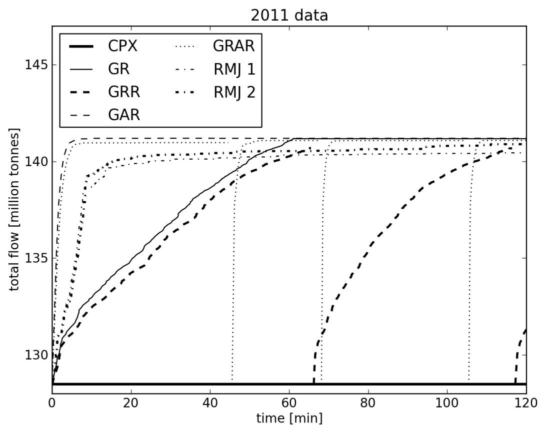


Figure 12: Progress for the 2011 data.

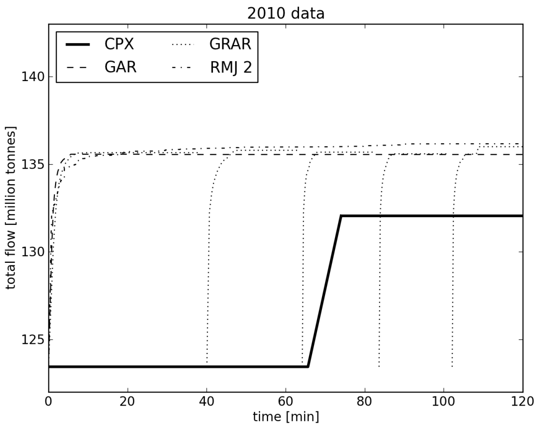


Figure 13: Progress for the 2010 data (selected algorithms).

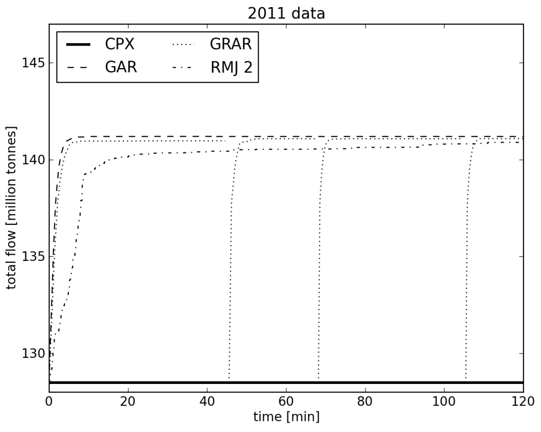


Figure 14: Progress for the 2011 data (selected algorithms).

5 Future directions

We want to point out three directions for further investigation, other than those already indicated in the paper.

1. A natural idea is to develop the local search towards a *Greedy Randomized Adaptive Search Procedure* (GRASP) [16]. That means, instead of using a fixed start solution, start solutions are constructed in a randomized greedy manner.

2. As the general problem is NP-hard, it is interesting to look for special cases (special in terms of the network structure and/or in terms of properties of the job list) that can be solved efficiently.

3. In the other direction, there might be generalizations of the problem that are worth studying, for instance allowing

    - arbitrary subsets of $[T]$ as sets of possible start times (not only intervals $[r_j, d_j]$), and
    - job processing to only reduce the arc capacity by some fraction, rather than taking it out completely.

    The former arises in the Hunter Valley coal chain application in respect of rail track maintenance, where crews must work during daylight hours of the working week. The latter obviously arises in contexts such as highway maintenance, where lane closures and slow-downs come into effect.

These examples of possible future directions illustrate what an exciting new problem we believe maximum total flow with flexible arc outages to be, with great potential for both theoretical and practical development.

Acknowledgment

We like to acknowledge the valuable contributions of Jonathon Vandervoort, Rob Oyston, Tracey Giles, and the Capacity Planning Team from the Hunter Valley Coal Chain Coordinator (HVCCC) P/L. Without their patience, support, and feedback, this research could not have occurred. We also thank the HVCCC and the Australian Research Council for their joint funding under the ARC Linkage Grant no. LP0990739. Furthermore, we are very grateful to the anonymous referees for their helpful comments and suggestions.

References

[1] N. Boland and M. Savelsbergh. "Optimizing the Hunter Valley coal chain". In: *Supply Chain Disruptions: Theory and Practice of Managing Risk.* Ed. by H. Gurnani, A. Mehrotra and S. Ray. Springer-Verlag London Ltd., 2011, pp. 275–302.

[2] B. Cavdaroglu, J.E. Mitchell, S.G. Nurre, T.C. Sharkey and W.A. Wallace. *Restoring infrastructure systems: An integrated network design and scheduling problem.* Tech. rep. `www.rpi.edu/~sharkt/RIS.pdf` (13 November 2011). Rensselaer Polytechnic Institute, 2010.

[3] L. Fleischer. "Faster algorithms for the quickest transshipment problem". In: *SIAM journal on Optimization* 12.1 (2001), pp. 18–35. DOI: `10.1137/S1052623497327295`.

[4] L. Fleischer. "Universally maximum flow with piecewise-constant capacities". In: *Networks* 38.3 (2001), pp. 115–125. DOI: `10.1002/net.1030`.

[5] L.R. Ford and D.R. Fulkerson. *Flows in Networks.* Princeton, N.J.: Princeton Univ. Press, 1962.

[6] M.R. Garey and D.S. Johnson. *Computers and intractability, a guide to the theory of NP–completeness.* W.H. Freeman, 1979.

[7] D. Goldfarb and M.D. Grigoriadis. "A computational comparison of the Dinic and network simplex methods for maximum flow". In: *Annals of Operations Research* 13.1 (1988), pp. 81–123. DOI: `10.1007/BF02288321`.

[8] B. Hajek and R.G. Ogier. "Optimal dynamic routing in communication networks with continuous traffic". In: *Networks* 14.3 (1984), pp. 457–487. DOI: `10.1002/net.3230140308`.

[9] B. Hoppe and É. Tardos. "Polynomial time algorithms for some evacuation problems". In: *Proc. 5th ACM-SIAM symposium on discrete algorithms SODA 1994.* Society for Industrial and Applied Mathematics. 1994, pp. 433–441.

[10] R. Koch, E. Nasrabadi and M. Skutella. "Continuous and discrete flows over time". In: *Mathematical Methods of Operations Research* 73.3 (2011), pp. 301–337. DOI: `10.1007/s00186-011-0357-2`.

[11] B. Kotnyek. *An annotated overview of dynamic network flows.* Tech. rep. 4936. `http://hal.inria.fr/inria-00071643/` (20 February 2013). INRIA, 2003.

[12] G.L. Nemhauser and L.A. Wolsey. *Integer and Combinatorial Optimization.* John Wiley & Sons, 1988.

[13] S.G. Nurre and T.C. Sharkey. "Restoring infrastructure systems: An integrated network design and scheduling problem". In: *Proceedings of the 2010 Industrial Engineering Research Conference.* 2010.

[14] R.G. Ogier. "Minimum-delay routing in continuous-time dynamic networks with piecewise-constant capacities". In: *Networks* 18.4 (1988), pp. 303–318. DOI: `10.1002/net.3230180405`.

[15] M.L. Pinedo. *Scheduling: theory, algorithms, and systems.* Springer, 2012.

[16] L.S. Pitsoulis and M.G.C. Resende. "Greedy randomized adaptive search procedures". In: *Handbook of applied optimization.* Ed. by P.M. Pardalos and M.G.C. Resende. Oxford University Press, 2002, pp. 168–183.

[17] J.G. Siek, L.-Q. Lee and A. Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual.* C++ In-Depth. Addison-Wesley Professional, 2001.

[18] M. Skutella. "An introduction to network flows over time". In: *Research Trends in Combinatorial Optimization* (2009), pp. 451–482. DOI: `10.1007/978-3-540-76796-1`.

[19] A. Toriello, G. Nemhauser and M. Savelsbergh. "Decomposing inventory routing problems with approximate value functions". In: *Naval Research Logistics* 57.8 (2010), pp. 718–727. DOI: `10.1002/nav.20433`.