Adaptation of the Gröflin-Klinkert-Bürgy Local Search to the SBB Challenge

Master thesis of

Fabio Degiacomi

Supervised by

Dr. Reinhard Bürgy & Prof. Bernhard Ries

Decision Support & Operations Research Group
University of Fribourg

June 2020

Abstract

The SBB challenge is a time table generation problem, which basically, is a blocking-job-shop scheduling problem. In the blocking-job-shop, unlike the classical job-shop, the N_1 neighbour of a given solution is generally not feasible. Gröflin, Klinkert & Bürgy developed a method, conceptually based on the insertion of a single job (a SBB train) into a schedule, which recovers the feasibility of N_1 neighbours. In the present master thesis, ways to improve the computational efficiency of that method are developed and the method is adopted to the additional constrains of the SBB challenge.

Contents

Cheatsheet

A	set of conjunctive arcs of a disjunctive graph, (V, A) acyclic
A^J	set of conjunctive arcs of the job-insertion graph G^J (inserting J)
block	'blocking' complication of a job-shop problem
$C_{\max}$	makespan objective, time span 0 until all operations terminated
d_{Jk}	due time of operation o_{Jk}
$\delta(.)$	function mapping an set of vertices to all arcs adjacent to any vertex of the set
E	set of disjunctive arcs of a disjunctive graph, $E = \bigcup_{(e,\bar{e})\in\mathcal{E}} \{e, \bar{e}\}$
$\mathcal{E}$	set of pairs of disjunctive arcs
E^J	set of disjunctive arcs of the job-insertion graph G^J (inserting J)
$\mathcal{E}^J$	set of pairs of disjunctive arcs of the job-insertion graph G^J
e	a disjunctive arc
$\bar{e}$	mate of e, ie $(e, \bar{e}) \in \mathcal{E}$, $(V, A \cup \{e, \bar{e}\})$ cyclic
f	another disjunctive arc
$\bar{f}$	mate of f
G	a disjunctive graph (directed), $G = (V, A, E, \mathcal{E}, l)$
G^J	job-insertion graph (inserting J), a disjunctive graph, $G^J = (V, A^J, E^J, \mathcal{E}^J, l)$
H_G	conflict hypergraph of a disjunctive graph G
H_{G^J}	conflict graph of a job-insertion graph G^J, short-cycle property, bipartite
$h(.)$	function mapping an arc to its head vertex
$\mathcal{J}$	set of jobs of a (blocking-)job-shop problem
J	a job of a (blocking-)job-shop problem
J_m	job-shop problem with m machines
$l(.)$	function mapping an arc to its length
$l^S(.,.)$	function mapping two vertices to the length of a longest path between them in $(V, A \cup S)$
$\mathcal{M}$	set of machines of a (blocking-)job-shop problem
m	a machine
n_J	number of operations of job J
$\mathcal{O}$	set of operations of a (blocking-)job-shop problem
o_{Jk}	k^{th} operation of job $J \cong$ a vertex in the disjunctive graph
p_o	processing time of operation o
pm	'parallel machines,' a single operation may use multiple machines
prec	'precedence constraints' between operations of different jobs
r_{Jk}	release time of operation o_{Jk}
rcrc	'recirculation,' a jobs uses a machine for multiple operations
route	'routing options', for each job a route has to be selected
S	a selection, $S \subset E$, feasible if $(V, A \cup S)$ acyclic, complete if $\lvert S\rvert = \lvert E\rvert/2$ else partial
s_i	setup time of machine i
$succ(.)$	function mapping an operation to its successor operation
T_{Jk}	tardiness (delay) of operation o_{Jk}
$T_{\max}$	maximum tardiness (delay)
$t(.)$	function mapping an arc to it's tail vertex
t_o	entry time of operation o
U	set of edges of a conflict graph $\cong$ set of cycles in a disjunctive graph
V	set of vertices of a disjunctive graph $\cong$ the operations of a (blocking-)job-shop problem
w_{Jk}	weight/priority of the operation o_{Jk}
Z, Z'	cycles in a graph
$\preceq$	$a \preceq b$, if a path from a to b exists in (V, A) $\cong$ operation a precedes b of the same job

1 Background

1.1 SBB Challenge

In 2018 the Swiss Federal Railways (de: Schweizerische Bundesbahnen (SBB), fr: Chemins de fer fédéraux suisse (CFF); subseqently SBB) organized a competition, where a participant is to solve a series of timetable generation problems [27].

Such a problem defines a set of *trains*. For each train a *route* has to be chosen. Each route consists of multiple *sections* which the train has to traverse. Each section has a *run time*, that is, the minimum time it takes a train to traverse said section. Sections may also have requirements attached: latest and/or earliest entry times. Sections have *resources* associated. The time table generation problem is then: associate each train with a route; associate each section (within a route chosen) with an entry time; such that, at no time, no two sections, which share a resource, are both being traversed by their respective train. Earliest entry times have to be respected. The delay (assigning entry times later than the 'latest entry') has to be minimized. Finally, *connections*: we have to guarantee that passengers can change form some trains to others at given sections.

The core of this problem is called *blocking-job-shop problem*, which will be introduced shortly. The blocking job shop problem is well known to be NP-hard. As the scale of problem, that researchers at SBB want to solve is much too large to solve to optimality, the hope of the SBB challenge was, that someone would come up with a ingenious heuristic that could be developed further.

Academic research, while traditionally focusing on the simpler *job-shop problem* made progress in the last 20 years on the *blocking-job-shop problem*. However, there are significant differences between the problems considered in academic literature and the SBB challenge: foremost the scale of problems studied. The problem size can be characterized by the number of *jobs* (or *trains*) n and the number of *machines* (rail segments) m. The blocking-job-shop research community has focused on a series of randomly generated problem instances, which range in size ($n \times m$) from 10×5 to 30×10. The problem instances provided by the SBB are as large as 350×450.

While it is the case that the blocking-job-shop problem is very much relevant in practical applications - in contrast to the job-shop problem - it is also true, that the scale most often studied academically is too small for practical relevance.

One of the most promising approaches to solve the blocking-job-shop problem, was developed over the last years at the University of Fribourg by Heinz Gröflin, Andreas Klinkert and Reinhard Bürgy [10, 12, 22, 23]. The SBB challenge provided us with an larger example problem for which I could adopt this existing algorithm. Hence, we considered it worthwhile to work on the SBB challenge, even though the competition formally concluded by the time it caught our attention.

1.2 Real-time train dispatching

The time table generation problem, as established by the SBB challenge, is often referred to as *real-time train dispatching* in the literature. It is not the problem of creating an official time table (off-line), but rather, as deviations to this official timetable inevitably occur, rescheduling and rerouting trains in real time (on-line) to minimize overall delay.

In practice this problem is broken down into sub problems, namely the *conflict detection and resolution* and the *speed profile generation*[11]. It is also common to solve these problems for specific areas and then integrate these partial solutions (see [11] for an overview of the employed system architectures, or [18] for a solution approach which depends on a decomposition of the problem into geographical sub units: line and station problems).

The speed profile generation problem has it's own intricacies, as when we concern ourselves with the problem on such a practical level, we have to take heterogeneous safety and operational rules, different signaling systems, as well as conflicting objectives such as punctuality, minimization of travel time and energy efficiency into account [11]. However, as the SBB challenge's API exposes essentially the real-time train dispatching, conflict detection and resolution problem. We shall focus on this aspect alone. It should be noted, that completely different approaches to the conflict detection and resolution problem exist. For an overview consult [17].

In the next paragraphs, we will first define the blocking-job-shop scheduling problem, which is an abstraction and simplification. After the paragraph on disjunctive graphs - which are a common model of the blocking job shop - I present a short literature overview of different approaches to the blocking-job-shop. Most of them depend on disjunctive graphs. Only then shall we come back to real-time train dispatching and the SBB challenge, by connecting the blocking-job-shop with the SBB challenge, thereby concluding the 'background' chapter.

1.3 (Blocking-)Job-Shop Problem

The *job-shop problem* and its variations aim to formalize different *scheduling* problems. In the classical job-shop problem, a set of *jobs* $\mathcal{J} = \{J_1, \dots J_n\}$ is given. Each job consists of a sequence of operations, ie. $J = (o_{J1}, \dots, o_{Jn_J})$, where n_J denotes the number of operations of job J. The jobs can be thought of as tasks that need to be completed, where each task decomposes into subtasks, the *operations*, which need to be done *in the given order*. Each operation occupies for its processing a *machine*, that is, there is a function $M : \mathcal{O} \to \mathcal{M}$ associating to each operation a machine. Here, $\mathcal{O}$ denotes the set of all operations and $\mathcal{M} = \{M_1, \dots, M_m\}$ is the set of all machines. The machines model resources that need to be acquired to complete the operations/subtasks, and let us encode constraints like 'o_l and o_k cannot be processed simultaneously'. (Going forward notation will be abused slightly. Vectors use as index equivalently: (1) an operation or (2) a job and an operation index.)

Each operation o has a *processing time* $p_o \in \mathbb{R}_{\geq 0}$ (or $\in \mathbb{Q}_{\geq 0}$, or $\in \mathbb{N}_0$) associated. The goal is to set the start time of each operation, that is, to define a vector $t \in \mathbb{R}_{\geq 0}^{|\mathcal{O}|}$.

Commonly, we associate a job with its completion time, $C : \mathcal{J} \to \mathbb{R}_{\geq 0}, J \mapsto t_{Jn_J} + p_{Jn_J}$. The duration form 0 to $C_{\max} := \max_{J \in \mathcal{J}} \{ \ C(J) \ \}$ is called *makespan*. The minimization the makespan is an objective common to job shop problems. A job shop problem can easily be transformed into a linear disjunctive program:

$$\begin{aligned} &\min_t C_{\max} && \\ 0 &\leq t_o && \forall o \in \mathcal{O} \\ t_{Jk} + p_{Jk} &\leq t_{Jl} && \forall J \in \mathcal{J}\ \forall k\ \forall l \mid 1 \leq k < l \leq n_J \\ t_{o_1} + p_{o_1} \leq t_{o_2} &\vee t_{o_2} + p_{o_2} \leq t_{o_1} && \forall \{ \ (o_1, o_2) \mid o_1 \in \mathcal{O}, o_2 \in \mathcal{O}, o_1 \neq o_2, M(o_1) = M(o_2) \ \} \end{aligned}$$

Variations of the job-shop problem have been classified. A specific problem is associated with a triple $\alpha \mid \beta \mid \gamma$, where α stands for the broad problem type, β is a list of 'flavors' and γ defines the objective function. $J_m \mid\mid C_{max}$ is job-shop problem introduced above. The notation introduced here is based on the textbook 'Scheduling' by Micheal L. Pinedo [20]. Elements are selected due to the relevance to the present thesis and not their importance in scheduling theory. For more in depth definitions the reader is referred to [20] page 13 et seq.

Problem types (α):

1 A scheduling problem with a single machine.

$\mathbf{J_m}$ The job shop problem introduced above, with m machines.

'Complications' (β):

$\mathbf{r_{Jk}}$ Release date: a job J's, operation o_{Jk} cannot start execution before the time r_{Jk}.

$\mathbf{d_{Jk}}$ Due date: if job J's operation o_{Jk} completes processing after d_{Jk} a penalty is applied.

prec Precedence constraints: an operation of a job may require operations of other jobs to be completed before being able to start.

prmp Preemptions: processing of an operation can be interrupted. The machine is freed. When the job in question resumes the operation, it has to be processed for the remaining time span.

$\mathbf{s_i}$ Machine dependant setup times: the machine i has to be idle for the time span s_i between operations.

pm Parallel machines usage: an operation may use multiple machines.

route For each job a *route* has to be selected. Equivalently, $\mathcal{J}$ can be seen as a *set of sets* of jobs. For each $J \in \mathcal{J}$, one $j \in J$ has to be selected and the scheduling problem is solved for only those selected jobs.

block Blocking models the absence of storage: a job needs to reside on a machine on all times. As a result, a job can only take over a machine after its occupant moved to the next machine. In the blocking environment, the last family of constraints of the disjunctive program above has to be modified:

$$t_{succ(o_1)} \leq t_{o_2} \vee t_{succ(o_2)} \leq t_{o_1} \quad \forall \{ (o_1, o_2) \mid o_1 \in O, o_2 \in O, o_1 \neq o_2, M(o_1) = M(o_2) \},$$

where $succ(o)$ denotes the successor operation of o. Two remarks are in order. Evidently the disjunctive formulation above does not work in case of the last operation. The academic literature therefore distinguishes between 'blocking' and 'ideal' (i.e. last) operations. However, as this is not a problem in practice - we can simply append a dummy operation of zero processing time, which uses no machine - I will not discuss this further. The second question which arises, asks if we are allowed to swap two jobs J_1, J_2 if J_1 uses first machine m_1 then m_2, while J_2 uses first m_2 then m_1, after both completed processing on their respective first machine. Two variants arise: blocking *with swap* and *without swap*. But the distinction vanishes if $s_i > 0$ for all $i \in \mathcal{M}$, which is the case in the SBB challenge. I will henceforth assume 'without swap' which makes the implementation slightly easier.

rcrc Recirculation: a job may need processing by the same machine multiple times.

Objective functions (γ):

$\mathbf{C}_{\max}$ *Makespan* objective, as defined above.

$\mathbf{T}_{\max}$ We define the *tardiness* of operation o_{Jk} to be $\max\{t_{Jk} - d_{Jk}, 0\}$. The objective T_{max} is to minimize the maximum tardiness.

$(\mathbf{w_{Jk}T_{Jk}})_{\max}$ Weighted maximum tardiness. As above, with operations also having priorities.

$\sum \mathbf{w_{Jk}T_{Jk}}$ Sum of weighed tardiness.

Just as some readers doubtlessly suspect, the train scheduling problem (as defined by the SBB datasets) are of the structure $J_m \mid r_{Jk}\ d_{Jk}$ prec s_i block rcrc pm route $\mid \sum w_{Jk}T_{Jk}$. $1 \in \alpha$ and prmp $\in \beta$ are introduced here, as heuristics that we will encounter in the blocking-job-shop literature review depend on such sub problems.

1.4 Disjunctive Graph

The aim of this paragraph is the introduction of a common model of the (blocking-)job-shop problem, the *disjunctive graph*. The problems we consider here are, $J_m \mid\mid \gamma$ and $J_m \mid$ block $\mid \gamma$. The disjunctive graph is a directed graph where each operation is associated with a vertex. Connecting them, arcs are labelled with a duration ($\in \mathbb{R}_{\geq 0}$). Arcs indicate that, the operation at the head of the arc cannot commence before the time when the operation at the tail started plus the duration of the arc's label. Arcs encode the idea that a operation has to precede another operation by a given time. Such graphs are also known as *precedence constraint graphs*. A given solution to a blocking-job-shop problem can be naturally represented by a precedence constraint graph. Now we aim to represent the solution space of a (blocking-)job-shop problem by the specialized type of precedence constraint graph which is the disjunctive graph. In the solution space of (blocking-)job-shop we encounter two types of arcs:

i) *Conjunctive* arcs occur in every solution. These are the arcs within the same job and encode that operation o_{Jk} has to precede o_{Jk+1}.

ii) *Disjunctive* arcs: For every two operations o_1, o_2 that use the same machine m we have a pair of disjunctive arcs. They encode: "o_1 is scheduled on m before o_2" and "o_2 is scheduled on m before o_1" respectively.

The disjunctive arcs occur in pairs, o_1 before o_2 *or* o_2 before o_1. A solution must contain one disjunctive arc for every pair, as we must make a choice which operation to schedule first. I offer some remarks before introducing a formal definition.

- In $J_m \mid\mid \gamma$, operations o_1, o_2 sharing m lead to the disjunctive arc pair

$$((o_1, o_2), (o_2, o_1)).$$

  The labels of theses arcs are the processing times of o_1 and o_2 respectively.

- In $J_m \mid$ block $\mid \gamma$, operations o_1, o_2 sharing m lead to the disjunctive arc pair

$$((o_1, succ(o_2)), (o_2, succ(o_1))).$$

  The labels of theses arcs are 0.

- If we have the model of a feasible job shop scheduling solution, a precedence constraint graph consisting of conjunctive arcs as well as the above mentioned choice of disjunctive arcs, it becomes easy to determine the start time of all operations. The start time of a operation is simply the maximum of vertices/operations that it is connected to backwardly plus the respective arc lengths. This has to be computed according to the topological order of the operations. This is described for example in the influential work of Mc Cormick et al. (1989) [3], who distinguish between 'processing arcs' (conjunctive arcs) and 'dummy arcs' (disjunctive arcs).
- The remark above already hints at the condition, when a precedence constraint graph corresponds to a feasible solution: If the graph contains a cycle (of positive length), we would have to conclude that all operations within the cycle could never be started. The respective solution is not feasible. The reverse is also true. Choosing disjunctive arcs such that the graph remains acyclic corresponds to a feasible solution.
- Given a precedence constraint graph corresponding to a feasible solution, the *makespan* objective is the length of the longest path in the graph.
- The constraints of the type $0 \leq t_o \ \forall o \in \mathcal{O}$ pose no difficulty. We add an additional vertex called σ to the graph. From σ we add arcs to the first operations of all jobs of length 0. The vertex σ is particular in the sense that it's associated time t_σ is constant 0. σ has no effect on the graph being cyclic or acyclic as its indegree is always 0.

Definition 1.1 (Disjunctive graph)**.** A tuple $(V, A, E, \mathcal{E}, l)$ defines a disjunctive graph, if

i) $(V, A \cup E)$ is a directed graph.

ii) (V, A) is an acyclic directed graph.

iii) $\mathcal{E}$ is a set of unordered pairs of arcs; E exactly contains all arcs of $\mathcal{E}$, ie.:

$$\forall (e, \bar{e}) \in \mathcal{E}.\Big(e \in E \wedge \bar{e} \in E\Big).$$

Furthermore, adding both arcs of a pair to (V, A) yields a cyclic graph. ie.:

$$\forall (e, \bar{e}) \in \mathcal{E}.\Big((V, A \cup \{e, \bar{e}\}) \text{ is cyclic}\Big).$$

For $(e, \bar{e}) \in \mathcal{E}$, we call $\bar{e}$ the mate of e and vice versa.

iv) $l : (A \cup E) \to \mathbb{R}_{>0}$, defines the length of an arc.

Note, that it is common to allow zero-length or negative length arcs, which I do not include in this definition, as no such arcs occur in the SBB-problem. It does simplify the discussion and implementation. Under this restriction, positive-length (a)cyclic is equivalent to (a)cyclic.

Directly related to the disjunctive graph is the *selection*.

Definition 1.2 (Selection)**.** S is a selection in a disjunctive graph $G = (V, A, E, \mathcal{E}, l)$ if $S \subset E$ and $\forall (e, \bar{e}) \in \mathcal{E}.\neg\Big(e \in S \wedge \bar{e} \in S\Big)$, that is S contains at most one element of every pair of $\mathcal{E}$. We call S complete (else partial) if S contains one element of every pair of $\mathcal{E}$, ie. $\forall (e, \bar{e}) \in \mathcal{E}.\Big(e \in S \vee \bar{e} \in S\Big)$, equivalently, $|S| = |E|/2$. S is called a feasible if $(V, A \cup S)$ is acyclic. A complete feasible selection corresponds to a feasible solution of the (blocking-)job-shop problem associated with the disjunctive graph.

We conclude this paragraph with definition of the critical subgraph.

Definition 1.3 (Critical subgraph)**.** For a given selection $S \subset E$, in a disjunctive graph $G = (V, A, E, \mathcal{E}, l)$ we define a subgraph of $(V, A \cup S)$. This subgraph depends on the objective function of the (blocking-)job-shop which we model with the disjunctive graph:

$\mathbf{T}_{\max}$ Let o be the operation/vertex in $(V, A \cup S)$ where the the maximum tardiness occurs. The longest path to o is called *critical path*. All disjunctive arcs along the critical path are called *critical arcs*. $T_{\max}$ is strictly more general than $C_{\max}$.

$\sum \mathbf{w_{Jk} T_{Jk}}$ Let $\mathcal{O}_{\text{cost}}$ be the set of all operation/vertices o where $T_o > 0$. The subgraph induced by all vertices along a longest path to any vertex in $\mathcal{O}_{\text{cost}}$ is called *critical subgraph*. A disjunctive arc within the critical subgraph is called *critical arc*.

Remark 1.1. *Note, that the critical arcs defined by a selection S exactly correspond to the choices leading to the cost of the solution corresponding to S. Hence, any change of the solution aiming to reduce the objective value, needs to modify choices represented by critical arcs.*

1.5 Literature review: blocking-job-shop

In the present review I am considering the development of different algorithms to tackle the blocking-job-shop problem that where developed over the last two decades. In effect, this complements older reviews such as [5]. To assign credit is a notoriously perilous task. In the present review I do not aim to do so, but rather, point out developments that are in *my opinion* important and hope this paragraph has an educational value. I proceed by introducing a construction heuristic that gave rise to multiple neighbourhood search schemes, called 'Amcc'. This, is followed by the category branch-and-bound based attempts to solve the blocking-job-shop problem. The paragraph is concluded by Xie & Mati's method which takes a different approach all together.

Amcc algorithm

A. Mascis & D. Pacciarelli adopted the disjunctive graph to the blocking-job-shop problem [6, 7], renaming the concept as 'alternative graph'. This, they motivate: a partial feasible selection of the (classical) job-shop problem can always be feasibly extended; a partial feasible selection of the *blocking*-job-shop problem cannot necessarily be feasibly extended [6, 7]. Here, we continue to use the term *disjunctive graph.*

Their work had a large impact on subsequent research, due to the proposition of a simple blocking-job-shop solution construction heuristic. I.e., a method, which builds a complete feasible selection. This construction heuristic - which is outlined below - depends on another heuristic to select a pair of disjunctive arcs, of which we did not yet include either in the solution. Of the four variants proposed, *Avoid maximum current* $C_{\max}$ *(Amcc)* had a lasting impact on research. "Amcc" was subsequently used for both the arc pair selection heuristic and the solution construction heuristic using said selection heuristic. For clarity I refer to them as the Amcc *rule* and Amcc *algorithm* respectively.

In the pseudo code below, see algorithm 1, the function $l^S : V^2 \to \mathbb{R}_{\geq 0}$ returns the length of the longest path in $(V, A \cup S)$ between the two input vertices.

Algorithm 1: Amcc algorithm

Input : A disjunctive graph $G = (V, A, E, \mathcal{E}, l)$.
Output: A complete feasible selection S or a failure.

Initialize $S := \emptyset$
Initialize a set $X := E$
while $X \neq \emptyset$ **do**
  // the AMCC rule:
  $(h, k) := \arg\max_{(u,v)\in X}\{l^S(0, u) + l(u, v) + l^S(u, n)\}$
  let (i, j) be the mate of (h, k)
  if $(V, A \cup S \cup \{(i, j)\}$ *is acyclic* **then**
    $S := S \cup \{(i, j)\}$
    $X := X \setminus \{(i, j), (h, k)\}$
  else if $(V, A \cup S \cup \{(h, k)\}$ *is acyclic* **then**
    $S := S \cup \{(h, k)\}$
    $X := X \setminus \{(i, j), (h, k)\}$
  else
    return *failure*
  end
end
return S

The algorithm chooses the pairs of $\mathcal{E}$ one-by-one, each time choosing one arc, which is inserted into the partial selection S. The pair chosen, contains the 'worst' arc, the arc which leads to the highest increase in cost. First, it is attempted to insert the other arc. If this fails, the arc which leads to the highest increase in cost is added. This may fail as well, in which case the algorithm failed.

Clearly, this is a weakness of the algorithm. This potential to fail is due to the fact, that not every partial feasible selection can be extended (mentioned above, unlike in the classical job-shop problem).

The authors try to mitigate this with the inclusion of *static implications*: an arc (i, j) is said to statically imply (u, v), if (with very limited on-line computational effort) it can be shown that (i, j) is incompatible with the mate of (u, v), given the partial selection S. Then, when adding (i, j) to S, also all static implications of (i, j) are added to S and removed from X.

Some examples of static implication rules include [7, 9]:

- If two jobs use the same sequence of machines, scheduling one on any of the machines implies the same choice - which one goes first - on the other machines of the sequence.

- A redundant arc is statically implied. (i, j) is redundant if $l(i, j) \leq l^S(i, j)$.
- If we work with a upper bound, the mate of an arc that pushes the cost above the upper bound is implied.
- The mate directly results in a cycle.

Pranzo & Pacciarelli [21] then developed a neighbourhood search based on the Amcc algorithm. A neighbour of a selection is generated by removing a subset of the selected disjunctive arcs (called *destruction phase*) followed by rerunning Amcc algorithm to again obtain a compete selection (*construction phase*). In the destruction phase 80% of the disjunctive arcs are randomly selected for removal.

This is similar to the approach of Oddi et al. [14, 15, 16] named 'iterative flattening search'; their 'relax' step corresponds to the destruction phase and their 'flatten' step corresponds to the construction phase. They however treat a slightly different problem, where machines additionally have capacities.

Dabah et al. [24, 26] refine the destruction phase: a single critical arc is selected and replaced by its mate. All cycles are then identified and disjunctive arcs along those cycles are removed. It should be noted, that the replacement of an arc by its mate is always feasible in the job-shop setting. The neighbourhood defined by those swaps is called N_1 neighbourhood (see [2], p394). This, they complemented by clever parallelisation and significant cluster based computational power. However, the claim of results being state-of-the-art is false. The results of [22] published two years prior are superior.

Branch & bound

A branch & bound algorithm is also commonly used to solve the job-shop problem and is a natural starting point. Branch and bound algorithms were used by D'Ariano et al. [9] or Dabah et al. [19]. A contribution by Sama, D'Ariano et al. [25] focuses on the incorporation of routing options (solving $J_m \mid \text{block route} \mid C_{\max}$).

A branch and bound algorithm orders the decision variables, before creating a tree of partial solutions. At the root is the partial solution without any decisions made. For every node in the tree, all possible decisions (for a single decision variable) are taken and result in child nodes. Until, at the leaf level, we are left with solutions (no more decisions left to take). When, as is commonly done, using the disjunctive arc pairs (choose between e and $\bar{e}$) as decision variables we get a binary tree.

The branch & bound algorithm requires a (heuristic) function associating with each sub tree a valid lower cost bound. I.e., given the decisions taken so far (path from the root), the partial solution cannot be extended into a complete solution with objective value lower than LB. If we already know a valid solution, of objective value UB, and $LB \geq UB$, we can conclude that it is pointless to further investigate the mentioned sub tree, as the already known solution cannot be improved. The sub tree is *pruned*.

The core questions here are:

- How do we choose the next decision variable
  D'Ariano proposes the use of the Amcc rule. Here, it should be noted that, complementing the Amcc algorithm with a static implication rule "arcs that violate an upper bound are prohibited" and backtracking when failing essentially transforms the Amcc algorithm into the branch & bond algorithm.
- What lower bound is worth computing?
  The branch & bound algorithm could prune the complete tree except for the path to the optimal solution if the LB were optimal, but then, finding this LB is the original problem.

  First, the current objective value of any partial solution is clearly a lower bound.

  Second, any blocking-job-shop solution is also a valid job-shop solution; the same solution has the same objective value in both problem settings. Hence, the optimal job-shop solution is a valid lower bound on the blocking-job-shop solution. As a result lower bounds developed for the job-shop are valid, though they are clearly not as effective.

  Pinedo [20] suggests solving $1 \mid\mid C_{\max}$, ie. the single machine job-shop problem for all machines separately before taking the best/highest of these lower bounds. Carlier [4] suggests computing $1 \mid prmp \mid C_{\max}$ instead, which is a valid (though worse) lower bound which can be calculated in polynomial time ($O(|J| \log |J|)$ by the EDD rule, see [20]).
- Implications:
  The static implication rules discussed for the Amcc algorithm can be employed here. In the context of the branch & bound algorithm they are called *dynamic* implications [9].

It is clear that, a branch and bound algorithm also profits form the cooperation with heuristic, as the better the upper bound, the more the search tree can be pruned. As an example, Dabah et al. [19] directly initialized their search with the best known objective value as an upper bound.

Geometric method

A different approach was developed by Mati & Xie in 2010 [13]. Their method is based on a efficient solution of the *two* job problem $J_{m,n=2} \mid \text{block} \mid C_{\max}$, by the *geometric method.* To find a schedule containing the jobs J_1, J_2, we have to 'stretch' or 'prolong' some of the operations of J_1 and J_2. They show how the 'stretching factors' leading to a minimal $C_{\max}$ can be found with a transformation of the problem into a shortest path problem. A 2D plot with the operations of J_1 on the x-Axis, the operations of J_2 are on the y-Axis is created. The operations follow the order given by the job ($o_{J_1 1}$, $o_{J_2 1}$ start at the origin). The length of a operation is defined by the processing time. For any given point in the plot we associate the two operations (of J_1, J_2) obtained by the projections onto the axes. A point is an 'obstacle' if and only if the two associated operations share any machines. The goal is now, to find a path from the origin to the top-right corner; moving only up, right and diagonally up-right. The path has to avoid all obstacle points. Intuitively, time moves forward as we move along the path. A shortest path corresponds to optimal 'stretching factors': moving right means J_1 is running its operation while J_2 is idle (up, vice-versa); moving diagonal means both jobs are processing an operation.

Now, we use the method of the two job two problem, to solve the blocking-job-shop problem. Notice, that a set of scheduled jobs can be merged into a single job. At a point in time, we know which machines are currently used by all jobs of the set. Whenever this changes, we create a new operation. Each operation is associated with all machines used during its respective time window.

Combining this with the geometric method, a permutation of the input jobs is required to build a solution. Mati & Xie propose the use of a taboo search to find a permutation leading to a good schedule.

1.6 The SBB challenge as a blocking-job-shop

As already stated, the academic literature mostly focuses on the $J_m \mid \text{block} \mid C_{\max}$. We hinted at the SBB problem being of the type

$$J_m \mid r_{Jk}\ d_{Jk} \text{ prec } s_i \text{ block rcrc pm route} \mid \sum w_{Jk} T_{Jk}.$$

First, we connect the SBB time table generation problem to the simple blocking job shop. Then, we shall see how the different constraints of the SBB problem translate into the various (β) modifications of the problem.

Each train that we need to schedule corresponds to a job. A train has to go through a sequence of track-segments, which naturally constitute the operations of the job-shop problem, as for each track segment we need to associate an entry time. A machine corresponds to an area in the track network, wherein at a point in time only a single train is allowed, for safety reasons (block). Multiple track sections can lie within the same 'safety are', hence multiple operations of the same job/train use the same machine (rcrc). Further, as these safety areas overlap, a single operation needs to acquire the locks for multiple areas (pm). For the transfer of a lock (on a area) an additional safety time has to pass. This is modelled by machine-dependant setup times (s_i). The earliest/latest entry times of operations are directly mapped to the variations r_{Jk} and d_{Jk}. As we can reschedule trains to use different routes, we have the 'route' variation. Another complication of the SBB challenge is the existence of connections. A train is only allowed to depart form a station/operation after another has arrived and passengers had time to change trains. This adds the further complication 'prec', that there are dependencies between different operations of different trains.

Finally, the objective function: different trains have different priorities and some stops/operations might be more important than others, hence the weight w_{Jk}. The quality of the timetable is assessed by the sum of all weighted delays ($\sum w_{Jk} T_{Jk}$). In the chapter 'implementation', we return to these complications and see how they can be incorporated into the 'method' (following chapter) which, as presented, solves the problem $J_m \mid \text{block} \mid C_{\max}$.

2 Method

2.1 Informal introduction

The method I implemented was developed by Heinz Gröflin, Andreas Klinkert & Reinhard Bürgy [12, 22, 23] based on the underlying theory developed by Heinz Gröflin & Andreas Klinkert [10]. Before going into the more technical details, I want to give an informal overview of the idea.

The method reschedules individual jobs.

The method is based around restricting the complete problem, where each job is flexible, to the problem where the schedule is fixed, except for a single job J which can be moved. This manifests itself as follows: from the disjunctive graph another disjunctive graph is derived; this derivation fixes all disjunctive arcs that are not adjacent to J, transforming them into conjunctive arcs. The derived disjunctive graph is called *job-insertion graph* As we will see, the job-insertion graph has a property we can exploit to effectively manipulate the placement of J.

What purpose does this serve? First, if we want to get rid of a critical arc and replace this arc by it's mate, we can reschedule one of the two jobs adjacent to the arc, including the mate instead, otherwise staying as close as possible to the original solution. In effect, this allows us to define a neighbourhood akin to the N_1 neighbourhood of the classical job-shop. Second, we can build up initial solutions by inserting jobs one-by-one. We can delete a job and reschedule it with a different routing, or place a job according to a heuristic.

In this chapter, I first define a conflict graph for the complete problem. This serves to illustrate why the derivation of the job-insertion graph is necessary. After the precise definition of the job-insertion graph, we will come to the property hinted at above: the *short-cycle-property*. With this, we see that the conflict graph of the job-insertion graph leads to an useful algorithm.

2.2 Conflict hypergraph

We create a conflict graph for a complete blocking-job-shop problem. As said, this is not supposed to yield a foundation for a algorithm, but rather motivate the construction of the *job-insertion graph* that follows.

We define a conflict graph for a disjunctive graph G. The conflict graph is an undirected graph $H_G = (E, U)$. We associate vertices with elements that can be selected. If selected elements *conflict* with each other, they are connected by an edge. In the case of a blocking-job-shop problem, the elements to select are the disjunctive *arcs* of E which become the *vertices* of H_G (*arcs* $\rightarrow$ *vertices*). A selection $S \subset E$ of arcs is conflicting if $(V, A \cup S)$ contains a cycle. Cycles in the blocking-job-shop conflict graph can be made of more than two arcs, and therefore, the 'edges' in the conflict graph also connect two or more vertices. The conflict graph is a hypergraph. An edge $\{e_1, \dots e_k\} \in U$ if $(V, A \cup \{e_1, \dots, e_k\})$ is cyclic.

Example 2.1 (Conflict hypergraph)**.** *In Figure 1 we have a disjunctive graph. Here, $\mathcal{E} = \{(b1, b2), (g1, g2), (o1, o2)\}$, that is the blue arc 'b1' is the mate of the blue arc 'b2', etc. We note that, as should be the case, each of these arcs conflicts with its mate, eg. $(V, A \cup \{b1, b2\})$ contains a cycle. However there are other cycles, eg. the one including $b2$, $g2$, $o1$ or $b1$, $o1$, $g1$. Figure 2 shows the conflict hypergraph. Edges between mates are represented as black lines, while other edges are represented by colored boxes around the respective vertices.*

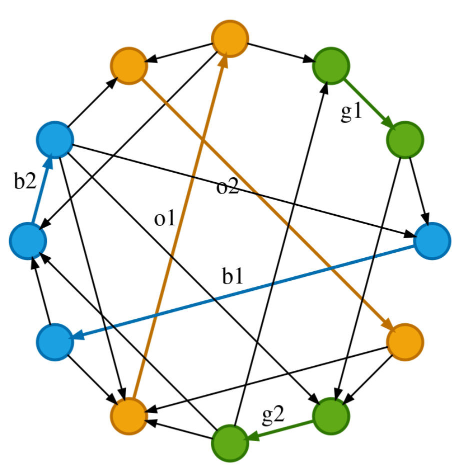


Figure 1: A disjunctive graph, where: $V =$ to all nodes, $A =$ all black arcs, $E =$ all colored arcs and $\mathcal{E} = \{(b1, b2), (g1, g2), (o1, o2)\}$. The graph illustrates the example 2.1.

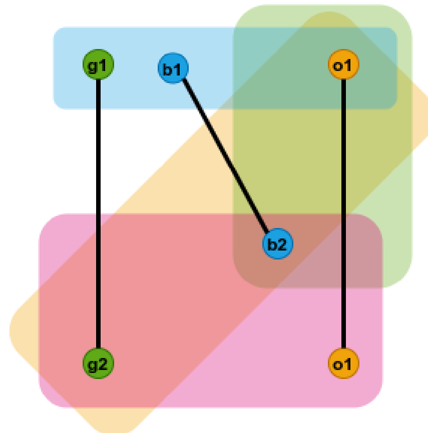


Figure 2: The conflict hypergraph of the disjunctive graph of figure 1. Nodes correspond to the arcs of figure 1. A black line represents an edge/conflict between mates. Colored boxes represent hyperedges. The graph illustrates example 2.1.

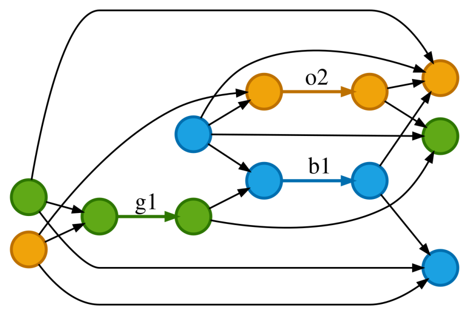


Figure 3: Continuation of figure 1 and 2, which are referenced in example 2.1 and 2.2. This figure shows the graph $G = (V, A \cup S)$ with $S = \{g1, b1, o2\}$. As expected G is acyclic.

Let us link the conflict graph to the original problem: A solution to the blocking-job-shop problem is feasible if and only it corresponds to a selection which is acyclic in the associated disjunctive graph. Furthermore, a selection corresponds to a set of vertices in the conflict graph and (by construction) we know that the selection is acyclic in the disjunctive graph if and only if it is an *independent* set (*stable* set) of the conflict graph. Vice-versa: an independent set of vertices in H_G of size $|\mathcal{E}|/2$ corresponds to a complete feasible selection, as from all pairs in $\mathcal{E}$ exactly one arc has to be chosen.

Example 2.2 (Conflict hypergraph)**.** *$S = \{g1, b1, o2\}$ is an independent set of the conflict graph of figure 2 of size $|\mathcal{E}|/2$. S corresponds to a complete feasible selection. Figure 3 shows $(V, A \cup S)$.*

Since we are only interested in the stable sets of the conflict graph, it is clear that we do not need to include '*non-elementary*' edges in the conflict graph. An example of non-elementary edge in figure 2 is the 'yellow' edge, as it contains the 'green' edge. Based on the above observations, we could define the following algorithm to solve the blocking-job-shop problem: Enumerate independent sets of the conflict graph of size $\mathcal{E}/2$, and choose the one with the best objective in the associated disjunctive graph. Return this as a solution. The number of possible cycles is expected to grow to fast with the size of the problem. This approach does not work, but I think it is important to see this and to contrast this with the method below.

2.3 Job insertion graph

The job insertion graph is also a disjunctive graph - as is the case with the 'complete' disjunctive graph which we defined for a blocking-job-shop problem. The difference is that we restrict ourselves to the following 'sub'-problem:

Given a feasible schedule, integrate an additional job.

We call this problem the *job-insertion problem.* To recall the connections between the elements introduced so far: A blocking-job-shop problem is modelled by the disjunctive graph. A solution to the blocking-job-shop problem corresponds to a selection in that disjunctive graph. The job insertion problem is a sub-problem, which we will also model with a disjunctive graph, called the job-insertion graph. A solution to the job-insertion problem corresponds to a selection in the job-insertion graph. Recall, that we defined conflict (hyper-)graphs for disjunctive graphs.

Definition 2.1 (Job-insertion graph)**.** Given is a blocking-job-shop problem with jobs $\mathcal{J}$, a specific job $J \in \mathcal{J}$ and a complete feasible selection S to the problem $\mathcal{J} \setminus J$. For S i.e.:

$\forall (e, \bar{e}) \in \mathcal{E}$ not adjacent to J, $e \in S \vee \bar{e} \in S$.

$\forall (e, \bar{e}) \in \mathcal{E}$ adjacent to J, $e \notin S \wedge \bar{e} \notin S$.

Associated with the blocking-job-shop problem we have the graph $G = (V, A, E, \mathcal{E}, l)$. The job-insertion graph $G^J = (V, A^J, E^J, \mathcal{E}^J, l)$ is constructed as follows:

$A^J := A \cup S$

$\mathcal{E}^J := \mathcal{E}$ restriced to arcs adjacent to J.

$E := E$ restriced to arcs adjacent to J.

In effect, this construction fixes the schedule of the jobs $\mathcal{J} \setminus J$. The the disjunctive arcs of those jobs are added to the conjunctive (fixed) arcs A. The disjunctive arcs adjacent to J are not restricted - we are free to insert J at any position.

In the next section, we will introduce the *short-cycle property*. We will show that the job-insertion graph does have this property. With this we will come back to the conflict graph. The conflict graph of the job-insertion problem will allow us to define a useful algorithm. This is to be contrasted with the attempt to define a conflict graph for the complete problem.

2.4 Short-cycle property

Notation: For a set of vertices X, we denote all arcs adjacent to some vertex of X by $\delta(X)$.

For every cycle visiting J, there is a 'shorter' cycle visiting J once.

The short-cycle property is a property of a disjunctive graph and an *insertion set* $N \subset V$. We impose the condition that $E \subseteq \delta(N)$, that is, all disjunctive arcs of the graph are adjacent to the insertion set. We consider the short-cycle property for job-insertion graphs. In our case, $N = J$, that is the insertion set we consider, is exactly the job to be inserted. Hence, by construction of G^J, $E^J \subseteq \delta(J)$, indeed $E = \delta(J) \setminus \{\text{conjunctive arcs of } J\}$. Furthermore, we know that the vertices of J form a path in (V, A). This path is not reachable (in (V, A)) form any vertex in $V \setminus J$.

Definition 2.2 (Short-cycle property)**.** A disjunctive graph $G = (V, A, E, \mathcal{E}, l)$, with insertion set N ($E \subseteq \delta(N)$), has the short-cycle property if for any cycle Z in $(V, A \cup E)$, there exists a cycle Z' in $(V, A \cup E)$ with $Z' \cap E \subseteq Z \cap E$ and $|Z' \cap E| = 2$.

Proposition 2.3. *The job-insertion graph $G^J = (V, A^J, E^J, \mathcal{E}^J, l)$ has the short cycle property.*

Proof. We proof the proposition by induction over the number of times a cycle visits the job J.

- $\mathbf{n = 1}$:
  In this case we enter and leave J, hence $|Z \cap E| = 2$. ✓

- $\mathbf{n = 2}$:
  The treatment of this case is not necessary for the proof, but helpful to understand the proposition. We choose arbitrarily one vertex where the cycle Z enters J and call this vertex a. Let b be the vertex where Z first leaves J, after a. Let c be the vertex where Z reenters J, after b. And, let d be the first vertex where Z leaves J, after c. Within J vertices are ordered by reachability (in (V, A)), we write $v \preceq w$ if $\{v, w\} \subseteq J$ and w is reachable from v.

  We have the following six possibilities, see figure 4 for an illustration of the cases.

  i) $a \preceq b \preceq c \preceq d \Longrightarrow$ case a).
  ii) $a \preceq c \preceq d \preceq b \Longrightarrow$ case b).
  iii) $a \preceq c \preceq b \preceq d \Longrightarrow$ case c).
  iv) The above cases, with $a \leftrightarrow c, b \leftrightarrow d$.

  Which is exhaustive, given the constraints, $a \preceq b$, $c \preceq d$ (a cycle cannot leave J before it entered it). As indicated by the colored boxes in figure 4, at least one short cycle exist in all cases. ✓

- $\mathbf{n \rightsquigarrow n + 1}$:
  Let Z be a cycle which enters J $n + 1$ times. We choose arbitrarily a vertex a where the cycle Z leaves J. Let b first vertex after a where Z reenters J. We differentiate two cases:

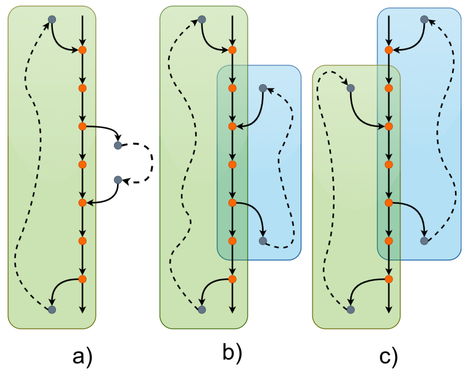


Figure 4: Illustration of the short cycle property. The figure shows a part of the graph G^J. The orange vertices are within the job J, grey vertices are of the other jobs $\mathcal{J} \setminus J$. Dashed lines indicate that "somewhere" in the graph a path exists. We show the 3 cases of a cycle Z visiting J 2 times. The colored areas indicate the short cycles Z' which satisfy the conditions $Z' \cap E \subseteq Z \cap E$ and $|Z' \cap E| = 2$.

- **a $\preceq$ b**:
  Define a cycle Z', equal to Z, with the path $a \to b$ replaced by the path $a \to b$ within J. Then Z' enters J n times. From the induction hypothesis it follows that a short cycle exists. ✓
- **a $\succ$ b**:
  $b \to a$ (within Z) followed by $a \to b$ (within J) is a valid short cycle. ✓

□

In the next section we will see the importance of the short cycle property.

2.5 Conflict graph

We recall the conflict graph $H_G = (E, U)$, defined for a disjunctive graph $G = (V, A, E, \mathcal{E}, l)$ in paragraph 2.2. The disjunctive arcs in the insertion graph become the vertices of the conflict graph.

Analogous, we now create the conflict graph of a job-insertion graph G^J, $H_{G^J} = (E^J, U)$, and restrict the edges of U to *elementary edges* ($\cong$ elementary conflicts). The short-cycle property now guarantees that each edge in U is a 'normal' edge connecting *two* vertices. If this were not the case, an edge $u \in U$ would represent a cycle in $(V, A^J \cup u)$ with $|u| > 2$, hence by the short-cycle property a short cycle exist. This short cycle would correspond to an edge $u' \subset u$, with $|u'| = 2$. Then, u is not elementary, a contradiction to the construction. The conflict graph H_{G^J} is not a hypergraph. I adopt the definition of the edge set to this insight: $U := \{\ (e, f) \mid (V, A^J \cap \{e, f\})$ contains a cycle $\}$, that is two vertices in the conflict graph are connected if adding them to (V, A^J) adds a cycle to that graph.

Next, we review the link to selections: $S \subset E^J$ is a selection. The selection is feasible if S is an independent set in H_{G^J}. S is complete if $|S| = |E^J|/2$.

Proposition 2.4. *The job-insertion conflict graph is bipartite.*

Proof. Let $(e, \bar{e}) \in \mathcal{E}^J$. Then either e or $\bar{e}$ has its head in $J \subset V$: $(h(e) \in J) \oplus (h(\bar{e}) \in J)$. And, the other arc has its tail in J: $(t(\bar{e}) \in J) \oplus (t(e) \in J)$. This is the case, as the two arcs represent the following decisions: an operation outside J is scheduled before the operation in J and vice-versa. We define $W := \{\ e \in E^J \mid h(e) \in J\ \}$, $\bar{W} := E^J \setminus W$. Then $W, \bar{W}$ are disjoint and $W \cup \bar{W} = E^J$. It remains to be shown that W and $\bar{W}$ are independent in H_{G^J}. We consider $(V, A \cup W)$. We know that the subgraphs of (V, A) induced by $V \setminus J$ and J are acyclic, and that all arcs in W go from $V \setminus J$ to J. Hence $(V, A \cup W)$ is acylic since as a cycle would require an arc from J to $V \setminus J$ and no such arc exists. We conclude that W is independent in H_{G^J}, by the definition of U. An analogous argument shows that $\bar{W}$ is independent in H_{G^J}. We conclude, H_{G^J} is bipartite. □

2.6 Closure operator

The previous definitions and properties (disjunctive graph, insertion graph with corresponding bipartite conflict graph) allow us to define a "closure operator", which forms the core of the algorithm. The closure operator takes as input a feasible solution (complete feasible selection) of a blocking-job-shop problem, as well as, a disjunctive arc that we want to "force" into the solution and returns a feasible solution - including the forced arc - that is as similar as possible to the input solution.

How does this work: Denote the input selection with S. First a job J at the head or tail of the arc to be forced in is identified. The insertion-graph is constructed.

$$G_J = (V, A^J, E^J, \mathcal{E}^J, l) = \left(V, A \cup \underbrace{\left(S \setminus (\underbrace{\delta(J) \cap E}_{*})\right)}_{**}, \underbrace{\delta(J) \cap E}_{*}, \delta(J) \cap \mathcal{E}, l\right)$$

* $\delta(J) \cap E$ are all the disjunctive arcs adjacent to the job J.

** (*) are removed from the selection S given as input.

In effect, we fix all disjunctive arcs not adjacent to J (add to A), but "reset" E to include all possibilities with respect to J. We construct the conflict graph $H_{G_J} = (E^J, U)$.

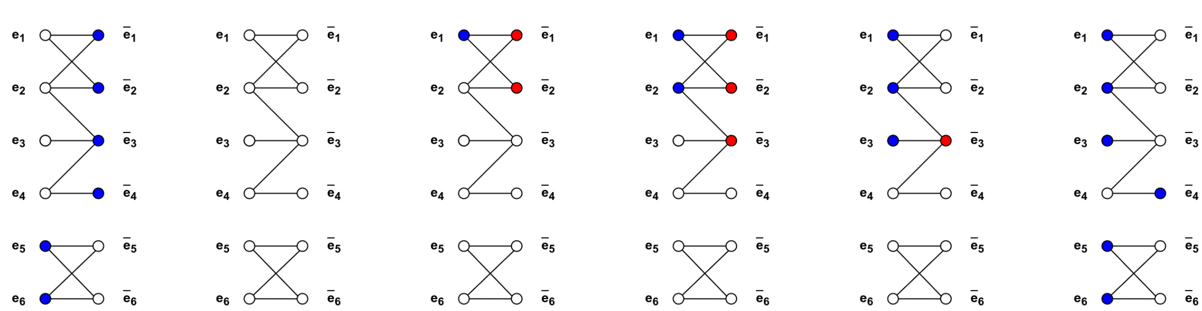
Now, we select the forced arc e in H_{G_J}. (Recall, e is an arc in G_J but a vertex in H_{G_J}.) If e is connected to f, we know that f cannot be part of the resulting solution. $\bar{f}$ has to be. Hence, $\bar{f}$ is also selected. The closure operator than takes the *closure* of this selection process, which yields a subset of E^J which is stable in H_{G_J}. This subset is then completed with the input selection S to form a new feasible solution, which is the result of the closure operator.

Example 2.5 (Closure operator). *In figure 5 a tiny example with six pairs of disjunctive arcs is introduced and a closure is derived to illustrate the description of the method above.*

Remark 2.6. *For a given critical arc e which we aim to remove, we have the choice of fixing all jobs $\mathcal{J}$ except $J_1 \mid h(e) \in J_1$ or $J_2 \mid t(e) \in J_2$. What is the difference?*

- *Choosing J_1: We remove an arc e which enters J_1 by forcing in an arc $\bar{e}$ that leaves J_1. A short cycle through $\bar{e}$ must therefore contain an arc f which enters J_1. f must be replaced with $\bar{f}$, which leaves J_1. The same argument holds for $\bar{f}$, indeed for the complete process. More and more arcs leave J_1, while fewer and fewer arcs enter J_1. This has the effect that J_1 moves backward in time through the time table, or, in a Gantt-chart, to the left. We call this version of the closure the left-closure.*

- *The exact opposite.*
  (Choosing J_2: We remove an arc e which leaves J_2 by forcing in an arc $\bar{e}$ that enters J_2. A short cycle through $\bar{e}$ must therefore contain an arc f which leaves J_2. f must be replaced with $\bar{f}$, which enters J_2. The same argument holds for $\bar{f}$, indeed for the complete process. More and more arcs enter J_2, while fewer and fewer arcs leave J_2. This has the effect that J_2 moves forward in time through the time table, or, in a Gantt-chart, to the right. We call this version of the closure the right-closure.)



(a) (b) (c) (d) (e) (f)

Figure 5: $\mathcal{E}^J = \{(e_1, \bar{e_1}), \ldots, (e_6, \bar{e_6})\}$. The input to the closure operator is the initial selection $S = \{\bar{e_1}, \bar{e_2}, \bar{e_3}, \bar{e_4}, e_5, e_6\}$, as well as the constraint, that e_1 has to be included in the selection. An example conflict graph $H_{G^J} = (E^J, U)$ is given in (a). The initial selection S is marked in blue. In a first step, (b), the selection is cleared. Next, (c), the arc to be added, e_1 is selected. Since $(e_1, \bar{e_1}) \in U$ and $(e_1, \bar{e_2}) \in U$, the arcs $\bar{e_1}$ and $\bar{e_2}$ cannot be selected. Hence, (d), the mate of $\bar{e_2}$ which is e_2 is selected. Now, as $(e_2, \bar{e_3}) \in U$, $\bar{e_3}$ cannot be selected. In (e), after selecting $\bar{e_3}$, we detect that the closure is complete. Finally, (f), the new partial feasible selection $S' = \{e_1, e_2, e_3\}$ is complemented with S to yielding the result of the operation: a new complete feasible selection $\{e_1, e_2, e_3, \bar{e_4}, e_5, e_6\}$.

2.7 Local search

With the closure-operator we can create an complete feasible selection S' from a given complete feasible selection S such that a given arc $e \in S$, $e \notin S'$. How do we use this to find a solution to the blocking-job-shop problem? Recall the definition 1.3 where critical arcs are introduced. A critical arc is corresponds to a decision that leads to some penalty. If we use the closure-operation to remove a critical arc, we find a similar, possibly better solution. This way we define a a neighbourhood. I.e., S has, as neighbours all S', such that a critical arc e in S exists with $S' = \text{closure}(S, e)$. Given the neighbourhood of feasible selections, we can employ a suitable meta-heuristic. We use a taboo-search to find a locally optimal solution to blocking-job-shop problems.

A taboo search generates a sequence of solutions $S^{(0)}, S^{(1)}, S^{(2)}, \ldots$ such that $S^{(i+1)}$ is the neighbour of $S^{(i)}$ with minimum cost, which is not taboo. Taboo arcs are accumulated when we have to accept a solution which is worse than the current one; the critical arc e being removed to create a new neighbour becomes a taboo arc. A neighbour is taboo if it's objective is worse than the best objective value in $S^{(0)}, \ldots, S^{(i)}$ and contains taboo arcs. The search can be terminated at any point. $S^{(i)}$ with minimum cost is returned as a solution.

3 Implementation

This chapter on the implementation largely decomposes into two parts:

First, we will see how the theoretical algorithm discussed in the chapter 'method' is implemented. As outlined, this algorithm solves the problem $J_m \mid \text{block} \mid C_{\max}$. The discussion here mostly focuses around the fact that, neither the job-insertion graph G^J, nor the conflict graph $H_{G^J} = (E^J, U)$ are computed or stored in memory. These optimization are mostly part of the method as implemented by [22], but are for the first time documented here in detail. In line with those optimization, I present the changes developed during this master thesis, namely: a faster way to update the entry times, a more efficient way to calculate the closure operation (which, additionally avoids the construction of the job-insertion graph) and a termination criterion which helps to further speed up the closure operation.

Second, I will describe how the algorithm is adopted to the additional complexity of the SBB problem: i.e., r_Jk, d_Jk, operation based earliest and latest entry; s_i, machine dependant setup time; $rcrc$, recirculation or usage of the same machine over multiple operations; *route*, routing possibilities; pm, parallel machine usage; *prec*, precedence constraints between jobs (connections); and the change in objective function to $\sum w_{Jk} T_{Jk}$. Theoretically, these elements can be incorporated with ease, although the proposed changes to incorporate *prec* and *route* should be improved. In any case, the implementation becomes significantly more complex.

3.1 Transitive arcs

Definition 3.1 (Transitive arc)**.** An arc (i, j) is called transitive, if a path from i to j exists and $l((i, j)) \leq l^S(i, j)$ (the arc is shorter than the path form i to j).

As we only ask if paths exist, and if so are only interested in the longest path in the graph $(V, A \cup S)$, transitive arcs can be ignored.

Example 3.1. *Assume we have 3 jobs a, b, c that use a machine m for their respective operations o_a, o_b, o_c. If the jobs use machine m in the order $a \to b \to c$, we have the following arcs is the disjunctive graph:*

- $succ(o_a) \to o_b$.
- $succ(o_b) \to o_c$.

The arc "$succ(o_a) \to o_c$" is transitive, we do not save it in the graph, as it is implicitly present.

For each machine we only store the non-transitive arcs. Assume machine m is used by n jobs. A complete selection now contains $n - 1$ arcs defining the order in which m is used instead of $\frac{n(n-1)}{2}$ arcs.

Remark 3.2. *If arc lengths are non-negative; the length of disjunctive arcs is 0, or only dependant on the machine m (as is the case in the SBB problem, see s_i), it is guaranteed that the path $succ(o_a) \to o_b \to succ(o_b) \to o_c$ is at least as long as $succ(o_a) \to o_c$. However, if the above condition is not met, we have to ensure that longest paths are always present in the disjunctive graph.*

As we have seen, a common operation is the replacement of an arc e by its mate $\bar{e}$. Figure 6 illustrates how this operation is preformed in the setting of storing transitive arcs only. On the left, some operations of four jobs are visualized. Operations circled in red share a machine. We assume that, before the operation is executed the red disjunctive arcs are present. On the right, the replacement is shown, step-by-step. Let e be the arc $succ$('blue') $\to$ 'orange' and $\bar{e}$ be the arc $succ$('orange') $\to$ 'blue'. First, e, alongside two arcs - which become transitive - are remove. Then $\bar{e}$ is inserted, alongside two arcs which become non-transitive. Note, that this operation corresponds to swapping the order in which the jobs 'blue' and 'orange' use the machine.

During the run time of the algorithm we store for each machine the sequence of jobs that use this machine, similar to the right side of figure 6. This allows for more efficient job-insertions and route-swaps.

3.2 Conflict graph

Why is it not necessary to form the conflict graph explicitly?

We use the conflict graph to query information like "enumerate the neighbours of e". In the disjunctive graph, this is equivalent to "find all disjunctive arcs $f \in E^J$ such that $(V, A^J \cup \{e, f\})$ is cyclic".

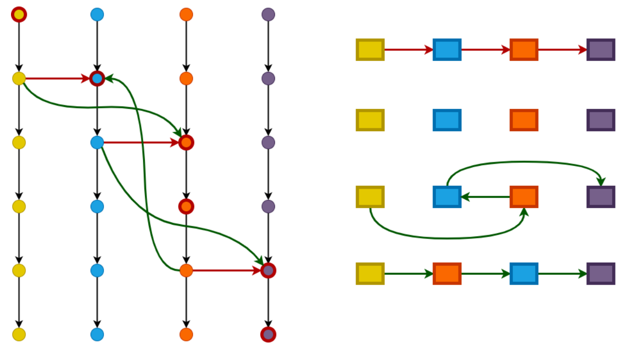


Figure 6: Illustration of the step-by-step replacement of a arc e = 'blue' $\to$ 'orange' by its mate $\bar{e}$ = 'orange' $\to$ 'blue'. Only non-transitive arcs are present.

Every edge in the conflict graph is used at most once when we transform a selection into a neighbour selection. Consider, the example of figure 5. (c), we find all cycles containing e_1. (d), we need to find all cycles containing e_2. Finally, (e), we need to find all cycles containing e_3. All the remaining edges in H_{G^J} ($\cong$ cycles in G^J) are never used.

What are the necessary cycle searches?

- In case of the left-closure: Recall, arcs such as e which *enter* J are replaced by arcs $\bar{e}$ that *leave* J. Whenever we insert $\bar{e}$, we have to check if a path exists between $h(\bar{e}) \to t(\bar{e})$ ($t(e) \in J$, $h(e) \notin J$). Of a potential cycle, only the disjunctive arc reentering J is relevant. Hence we search paths from $h(e)$ into the vertex interval $[o_{J1}, \ldots, t(e)]$. This is done efficiently with a forward path search.
- In case of the right-closure: Recall, arcs such as e which *leave* J are replaced by arcs $\bar{e}$ that *enter* J. Whenever we insert $\bar{e}$, we have to check if a path exists between $h(\bar{e}) \to t(\bar{e})$ ($t(e) \notin J$, $h(e) \in J$). Of a potential cycle, only the disjunctive arc first leaving J is relevant. Hence we search backward paths from $t(e)$ into the vertex interval $[h(t), \ldots, o_{Jn_J}]$. This is done efficiently with a backward path search.

3.3 Job-insertion graph

Why is it not necessary to form and store the job-insertion graph? We review the example of figure 5: Forming the job-insertion graph corresponds to the step (a) $\to$ (b), that is: inserting all the edges of E^J not already present into the current solution $(V, A \cup S)$, such that when we start the cycle search for the 'forced' arc e_1, we do actually detect the conflicts with $\bar{e_1}$ and $\bar{e_2}$. But, if we did not complete the graph to include all E^J we would still find the conflict with the original selection S, and those are exactly the conflict we are looking for. As a result the step (e) $\to$ (f) is also no longer necessary since, for all $(e, \bar{e})$ that were unaffected, the correct arc is already present.

However, a complication arises, if this is combined with the choice not to store transitive arcs in the disjunctive graph: a initial cycle search might not reveal a cycle as the respective conflicting arc is transitive, but, later in the process becomes non-transitive. The implementation of [22] therefore inserts all transitive arcs adjacent to J before the process is started, effectively transforming the graph $(V, A \cup S)$ into a job-insertion graph. At the end of the closure-operation the transitive arcs need to be removed.

There exist the following alternatives: The full set of path searches could be redone until no more cycles are found. Alternatively, the search can be resumed whenever a arc becomes non-transitive and another search already passed by the tail vertex (left-closure) or passed by the head vertex (right-closure).

3.4 Closure

Having seen what we actually save and compute we will now put together the closure algorithm. For simplicity, I just present the left-closure. The right-closure, as we have seen, is analogous. For a first, simple version see algorithm 2. As mentioned in the paragraph above, this only works if transitive arcs are present. The job-insertion graph still needs to be created. We shall see how this is avoided completely.

Algorithm 2: Naive left-closure

Input : A graph G corresponding to the current selection S: $G = (V, A \cup S)$, a disjunctive arc a to remove from the selection.

Output: A modified graph G, corresponding to a modified selection S, which is similar to the input S but does not contain the arc a.

arcsToRemove := $\{a\}$
while *arcsToRemove.Count* > 0 **do**
  Arc e = arcsToRemove.Pop()
  if *G.ArcExists(e)* **then**
    Arc $\bar{e}$ = G.SwapInMate(e)
    foreach *Arc* $f \in$ *G.ForwardPathSearchIntoRange*$(h(\bar{e}), [o_{J1}, t(\bar{e})])$ **do**
      arcToRemove.Add(f)
    end
  end
end

In algorithm 2 two functions are called:

- SwapInMate(e): removes e from the disjunctive graph and inserts $\bar{e}$. With respect to transitive arcs, this is done exactly as visualized in figure 6. Then, the arc $\bar{e}$ is returned.
- ForwardPathSearchIntoRange(vertex, vertexSet): searches all paths from vertex, that end in the vertexSet and returns a list of arcs, containing the last arc of each such path. The way this function is used, it returns all disjunctive arcs to which $\bar{e}$ is connected in the conflict graph H_{G^J}.

In the next paragraph we shall see how this naive version is optimized and simultaneously solves the problem, that in the present version, transitive arcs need to be present.

3.5 Closure: All at once?

The key insight for the optimization presented shortly is that the series of path searches to be done, follow a "nested" structure. Observe, that, the later within J the operation $t(\bar{e})$ occurs, the larger the target vertex interval of the path search. Indeed, $t(\bar{f}) \preceq t(\bar{e})$ implies $\{o_{J1}, \ldots, t(\bar{f})\} \subseteq \{o_{J1}, \ldots, t(\bar{e})\}$. Hence, for any vertex $v \in G$ such that a path exists form $h(\bar{f}) \to v$ and $h(\bar{e}) \to v$, the forward path search (v onward) to be completed for $\bar{e}$ renders the path search for $\bar{f}$ irrelevant. Before producing the respective algorithm 3, let us go through an example:

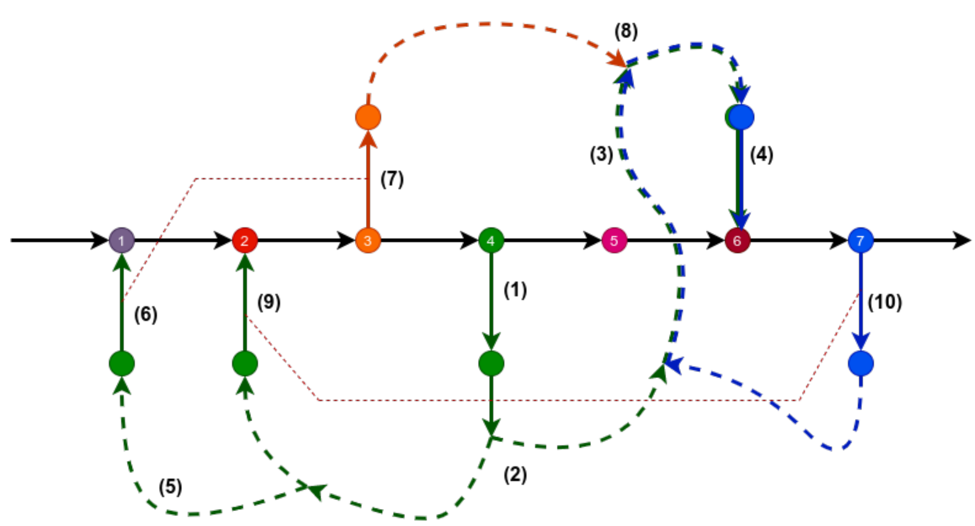


Figure 7: Illustration of example 3.3.

Example 3.3. *Closure, all in one. Below I outline step-by-step how the algorithm computes a left-closure as illustrated in figure 7. Black arcs correspond to conjunctive arcs within the selected job. Numbered, colored vertices are the vertices of the selected job. Dashed red lines match disjunctive arcs with their mates ((6) ↔ (7), (9) ↔ (10)). Dashed curves indicate that we understand this part of the graph as a black-box. We only know that a path exists. Numbers mark locations in the graph, which can be visited multiple times.*

(1) *We assume the green arc inserted at (1) is the one of which we take the left closure (ie, the mate of this arc was removed and this one inserted). We start be exploring the graph, searching for a path back into the job. While doing so, we color the visited vertices with "green (4)".*

(2,3,4) *We found a first path back into the job. But the vertex where we enter is "brown (6)". As* $6 > 4$, *there is no cycle.*

(5,6) *We found a second path back into the job. This time we hit the vertex "violet (1)". As* $1 \leq 4$ *we have a cycle. Hence, the arc at (6) has to be removed an is replaced by it's mate, the arc at (7).*

(7) *After the arc insertion at (7) we continue to explore the graph from there. Now, we color the vertices in "orange, (3)".*

(8) *We encounter a vertex that is already colored. Since the color of the vertex, is "green (4)" is larger than "orange (3)" we can stop the search here.*

(9) *We continue the search with green and find a 3rd path back into the job. Analogous to (6) we have to replace the arc at (9) with its mate, the arc at (10).*

(10) *after inserting the mate at (10) we continue out exploration of the graph, now marking vertices with "blue (7)". We encounter an already colored vertex. But since the vertex color is "green (4)" and* $4 < 7$, *we have to redo the search here. We continue and pass by (3) and (8).*

(4) *We revisit (4). But this time, the current color "blue (7)" is larger than "brown (6)". Hence, we have a cycle and the vertex at (4) has to be replaced by it's mate (which is not shown in the figure).*

Remark 3.4. *What did we save by combining the path searches as illustrated by the above example? When we met the green colored path of the graph, while preforming the path search for the vertex 3 at (7), we could skip going forward. Note, that when coloring in a higher color such as "blue (7)", we minimize the risk of having to partially redo the search. Therefore, we implement path search using a priority-queue to always first explore where we get the most "definite" results. The colors correspond to priorities. Furthermore, we know the priorities which we might encounter in advance. Those correspond to the operations within a Job. A bucket-queue (also called bucket-priority-queue or bounded-height-priority-queue) provides an efficient implementation for this use case. The use of this variant of the algorithm led to a decrease in necessary execution time of at least* 70% *(in medium sized SBB instances).*

Algorithm 3: All in one left-closure

Input : A disjunctive graph G and a selection S: $G = (V, A \cup S)$, a disjunctive arc a to remove from the selection.

Output: A modified disjunctive graph G, corresponding to a modified selection S, which is similar to the input S but does not contain the arc a.

Queue<Arc> Q = new BucketQueue()
Q.Add(G.SwapInMate(a), priority = any)
int[] P = new int[G.Count]
Initialize P: $P[o_{jk}] = \begin{cases} k, & \text{if } j = J \\ 0, & \text{otherwise} \end{cases}$
while $Q.Count > 0$ **do**
  a = Q.Pop()
  if not $G.ArcExists(a)$ **then**
    continue
  else if $h(a) \in J$ **and** $P[t(a)] \geq P[h(a)]$ **then**
    foreach $arc\ b \in G.SwapInMate(a)$ **do**
      if $P[t(b)] > 0$ **then**
        Q.Add(b, priority = P[t(b)])
      end
    end
  else if $P[t(a)] > P[h(a)]$ **then**
    foreach $arc\ b \in G.OutgoingArc(h(a))$ **do**
      Q.Add(b, priority = $P[t(b)]$)
    end
  end
end

In algorithm 3 these insights are included to form an improved version of the left-closure. Here, Q is an arc collection keeping track where in the graph $(V, A \cup S)$ we have to continue the path search. P is an array of priorities, such that we assign to each vertex in G a priority. How this works is illustrated in figure 7.

At every step a single arc a is removed from Q and treated:

- G.ArcExists(): a disjunctive arc could have been removed, due to another deteced conflict.
- The next *if* block treats the case of reaching back into the job J. If the priority we currently use is larger of equal to the priority of the operation of the job, we found a cycle. This last arc must be the disjunctive arc reaching back into J, which is removed. A key difference here is that *SwapInMate()* now has to return up to 3 arcs: the mate inserted, and the arcs that become non-transitive. If we already reached the tail of a newly non-transitive arc, then the priority is positive and we also insert this arc into Q.
- Finally, the "default" case. If we did not reach the $h(a)$ yet (priority of $h(a) = 0$) or we reach $h(a)$ again but with a higher priority, then the search continues: we insert all arcs continuing forward from $h(a)$ into Q.

A last optimization of the left-closure is presented in the next paragraph:

3.6 Termination Criterion

We developed a termination criterion for forward path search of the left closure. This also yields a significant speedup of the resulting algorithm when tested on the SBB problem instances.

The idea of the termination criterion is to partition the graph into two subgraphs: A and B. For any given time t, we can partition an acyclic graph into the vertex sets

$$A := \{v \mid t_v \leq t\} \text{ and } B := \ \{v \mid t_v > t\}.$$

Furthermore, no vertex of A is reachable from any vertex in B. Assuming that the left-closure computation effects no arcs adjacent to or in B, we can establish that A remains unreachable from B. If we then - during the forward path search of the left closure - encounter a vertex of B we can skip this branch of the path search.

We choose $t = \max_{a \in \delta(J)}\{\ t_{h(a)}\ \}$. Recall Remark 2.6: during the left-closure the job J moves to the left, or backward in time. Hence, we know that arcs in B will never be affected by the operations we preform.

Algorithm 4: All in one left-closure with termination criterion

Input : A disjunctive graph G and a selection S: $G = (V, A \cup S)$, a disjunctive arc a to remove from the selection.

Output: A modified disjunctive graph G, corresponding to a modified selection S, which is similar to the input S but does not contain the arc a.

1 Queue<Arc> Q = new BucketQueue()
2 Q.Add(G.SwapInMate(a), priority = any)
3 int[] P = new int[G.Count]
4 Initialize P: $P[o_{jk}] = \begin{cases} k, & \text{if } j = J \\ 0, & \text{otherwise} \end{cases}$
5 bool[] B = new bool[G.Count]
6 Initialize B: $B[i] = \begin{cases} true, & \text{if } t_i > \max_{b \in \delta(J)}\{\, t_{h(b)} \,\} \\ false, & \text{otherwise} \end{cases}$
7 **while** $Q.Count > 0$ **do**
8   a = Q.Pop()
9   **if not** $G.ArcExists(a)$ **then**
10   **continue**
11   **else if** $B[h(a)]$ **then**
12   **continue**
13   **else if** $h(a) \in J$ **and** $P[t(a)] \geq P[h(a)]$ **then**
14   **foreach** $arc\ b \in G.SwapInMate(a)$ **do**
15   **if** $P[t(b)] > 0$ **then**
16   Q.Add(b, priority = P[t(b)])
17   **end**
18   **end**
19   **else if** $P[t(a)] > P[h(a)]$ **then**
20   **foreach** $arc\ b \in G.OutgoingArc(h(a)$ **do**
21   Q.Add(b, priority = $P[t(b)]$)
22   **end**
23   **end**
24 **end**

The algorithm 4 shows how the termination criterion can easily be incorporated into the previous version of the left-closure, (line $4, 5, 11, 12$).

With this we conclude the discussion of the left-closure. All of the optimizations discussed here also apply to the right-closure. Here, I only remark that:

- The direction of the path search is reversed.
- The priorities of operations in J are reversed.
- The derivation criterion must be defined as a minimum of incoming arcs and the definition of A and B is reversed.

Next, I present a simple way to improve the effectiveness of the recalculation of the entry times (vector t) after we modified the graph $(V, A \cup S)$. This concludes the first half of this chapter, where we focus on improvements of the method to solve the problem $J_m \mid \text{block} \mid \gamma$. Thereafter, we will adopt the method to the SBB challenge.

3.7 Updating the entry times

We recall the standard way to compute the entry times: The entry time of the dummy vertex σ remains $t_\sigma = 0$. The rest of the graph is traversed in topological order and the entry times are updated as follows:

$$t_v = \max\{\, t_u + l((u, v)) \mid (u, v) \in (A \cup S) \,\}$$

While this is very simple to compute it has the drawback that we have to recompute entry times of the complete graph, even though most neighbours generated by the algorithm are close to the original selection and have, for a large part of the graph, identical entry times. I suggest a simple improvement: When the graph is modified we store vertices in a priority queue Q such that the following invariant holds:

- The entry times of all vertices topologically before all elements of Q remain correct.
- The entry times a vertex in Q is correct, if there is no other vertex in Q that topologically precedes the first vertex.
- Vertices topologically after a vertex of Q can have wrong entry times.

The priority with which a vertex is inserted into Q corresponds to it's (possibly wrong) entry time. The modifications necessary to the graphs methods *addArc()* and *removeArc()* are simple. Algorithm 5 shows how the entry times are updated.

Algorithm 5: Update Entry Times

Input : A graph $G = (V, A \cup S)$ and a priority queue Q. The vector t of entry times of all operations. (Precondition: invariant defined in paragraph 3.7.)

Output: A updated vector t of entry times.

while $Q.Count > 0$ **do**
    Vertex current := Q.Pop()
    foreach *Arc* $a \in G.OutgoingArcs(current)$ **do**
        if $t_{h(a)} < t_{t(a)} + l(a)$ **then**
            $t_{h(a)} := t_{t(a)} + l(a)$
            Q.Push($h(a)$)
        else if $t_{h(a)} > t_{t(a)} + l(a)$ **then**
            newTime := $\max_{b \in G.IncomingArcs(h(a))}\{\ t_{t(b)} + l(b)\ \}$
            if not $t_{h(a)} = newTime$ **then**
                $t_{h(a)}$:= newTime
                Q.Push($h(a)$)
            end
        end
    end
end

Remark 3.5. *A proof of termination and correctness of algorithm 5 is simple: Note, that the invariant defined above (precondition) is also a loop-invariant of the algorithm.*

On larger SBB instances the time necessary to recompute the entry times in the standard way made up approximately 30% of the computation time. With the modification described, this drops to 0.5%.

3.8 SBB complications

In the previous chapter we established the SBB challenge to be of the following problem type:

$$J_m \mid \text{block } r_{Jk}\ d_{Jk}\ \ s_i\ \text{ rcrc pm route prec} \mid \sum w_{Jk} T_{Jk}.$$

These additional constraints fall into one of three groups:

- Elements that can be incorporated into the method naturally and without any problems at all: $s_i, r_{Jk}, d_{Jk}, \sum w_{Jk} T_{Jk}$.
- Elements that change the data structures and implementation somewhat, but pose no theoretical challenges: rcrc, pm.
- route and prec: Here, the implementation is tricky.

I will discuss the 'complications' in the order mentioned above:

s_i, machine dependant setup times: In the paragraph 1.4 we established that the length of a disjunctive arc is 0. Instead we simply use the machine dependant setup time as the arcs label.

r_{Jk}, a operation o_{Jk} cannot start before r_{Jk}. We add arcs from σ to o_{Jk} of length r_{Jk}. The indegree of σ remains 0, hence these arc cannot be part of a cycle.

d_{Jk}, due time of operation o_{Jk}, see $\sum w_{Jk} T_{Jk}$

$\sum w_{Jk} T_{Jk}$, the weighed tardiness objective. No modification is necessary, recall how we defined critical arcs (paragraph 1.3) and local search (paragraph 2.7).

rcrc, **Recirculation and multi-operation machine use**

If a machine m is used by consecutive operations o_1, o_2 of the same job, we would have to insert the arc $(succ(o_1) \to o_2) = (o_2 \to o_2)$ of length s_m, which renders the graph immediately cyclic. Instead, we have to make the following adaption: assume e is the disjunctive arc which hands over machine m after o_2 to another job ($t(e) = succ(o_2)$), then we have to ensure that $h(\bar{e}) = o_1$.

The implementation changes: To avoid ubiquitous iteration though a job to find the start or end of a group of operations which share a machine, these groups or blocks of machine usages are made the core datatype on which we operate. In the code I call such a group a *MachineOccupation*.

pm, **Parallel machines**

In the SBB problems, an operation's use of multiple machines is ever-present and as a result, multiple machines are often transferred to other jobs after a given operation. I opted for the one-to-one correspondence between arcs and machine hand-overs, which leads to parallel arcs (disjunctive arcs are annotated with a machine). The motivation of this approach is:

- Ease of implementation: no need to query / compare the machine usages of the head and tail operations. Arcs can never get lost (ie. they merge, as a job that only uses one of the two machines is scheduled differently, but later fail to split), which could lead to bugs, that are very difficult to track down. The merging and splitting of arcs would necessitate keeping track of/computing setup times, which is otherwise "free".
- The one-to-one correspondence between arcs and the 'conceptual' nodes of the conflict graph continue to hold. If this correspondence were to break down, we would preform multiple steps in the conceptual conflict graph at once. The algorithm would become more difficult to reason about.
- Replacing one of two parallel arcs leads to a cycle which is detected immediately. Hence the other one is also directly replaced by its mate, with minimal overhead.

route, **Routing possibilities**

To incorporate routing possibilities I used a very simple and probably insufficient approach. It is easy to delete a job from a selection. We can include a job J into a solution: insert all its machines into the machine sequences: That is, pick and remove an disjunctive arc e present in the disjunctive graph approximately where we want to insert our job. Add the two arcs necessary ($t(e) \to J$, $J \to h(e)$). Then, take the left-closure of all the left arcs or take the right-closure of all the right arcs. The improved left-closure algorithm (algorithm 4) can handle this very effectively: we initialize Q to include all the "left arcs".

To choose the approximate position of the job J I use the following heuristic: each machine is positioned in the machine sequence as late as possible so that the inserted job is on time. Then a left-closure is used: The left-closure schedules J earlier. This way we guarantee that J itself does not cause any delay penalty. The "dual" of this approach would be to schedule J as early as possible and then use a right-closure. This has not been tried.

For the case when J was already present in the solution before (i.e., a route swap) I also implemented the swap such that the machines (and their disjunctive arcs) used by both routes remain unchanged.

prec, **Precedence constraints or connections**

The connections, or precedence constraints between operations of different jobs are the last complication posed by the SBB challenge. I tried to model the connection problem as follows: Assume o_1 of J_1 has to precede o_2 of J_2 by t_{conn}. A dummy machine is then added such that a disjunctive arc pair $((o_1, o_2), (\text{succ}(o_2), \text{predecessor}(o_1)))$ is added to $\mathcal{E}$. The idea was, to have the arcs length s_i being sequence dependant: t_{conn} if the arc (o_1, o_2) is present or a larger 'penalty' length otherwise. However, this leads to further complications, as the sequence dependant setup times require a modification of the termination criterion (see paragraph 3.6). A better idea might be to set the arc length for the pair to t_{conn} and add a due time $d_{o_1} = t_{o_2} - t_{\text{conn}}$ to generate both a penalty (set w_{o_1} respectively) and the critical arcs, which, when replaced decrease t_{o_1}. Furthermore, $(\text{succ}(o_2), \text{predecessor}(o_1))$ should be considered in the local search as if critical. I did not implement this, as I mostly focused on a SBB problem instance where connections are of minor concern (see paragraph 4.2).

4 Results & Conclusion

4.1 Validation experiment

The algorithm as described above was run on two sets of problem instances. In addition to the SBB challenge (see below), those are the 40 blocking-job-shop problem (J_m | block | $C_{\max}$) instances proposed by Lawrence [1]. This is done to validate the implementation of the algorithm. These results can be compared to Pranzo & Pacciarelli, 2016 [21], who also provided results for the blocking no-swap problem (as is solved by the implemented algorithm), see figure 8. I computed $40'000$ iterations of the taboo search for each problem instance. This took approximately 3 hours on a 3.2 GHz processor. The computation power is therefore roughly comparable to Pranzo & Pacciarelli ($40 \times 600\text{s} = 6\text{h}40\text{m}$, 3.0Ghz) [21]. I conclude that the validation experiment is a success (figure 8): Even though the algorithm is implemented to deal with hundreds of jobs and a number of additional constraints, the results indicate that the implementation is at least competitive and seem to have an edge when dealing with slightly less tiny (la31-la35) problem instances.

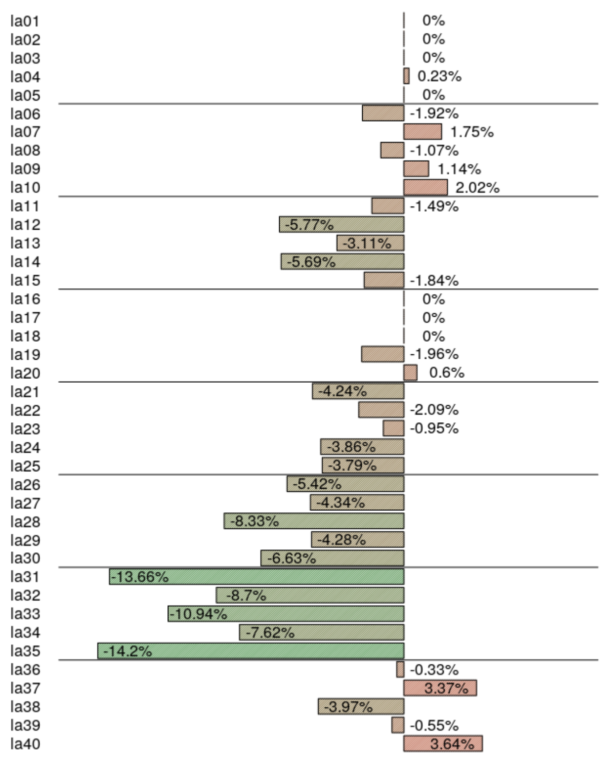


Figure 8: Results obtained by this implementation to compared to Pranzo & Pacciarelli [21]: Relative difference of the objective value ($C_{\max}$) is reported. The best result of [21] corresponds to 100%. Problem sizes ($n \times m$) are: la01 − 05, 10×5; la06 − 10, 15×5; la11 − 15, 20×5; la16 − 20, 10×10; la21 − 25, 15×10; la26 − 30, 20×10; la31 − 35, 30×10; la36 − 40, 15×15.

4.2 SBB challenge

Problem instances & Results

For the challenge the SBB [27] provided a set of 9 problem instances of the type

$$J_m \mid \text{block } r_{jk}\ d_{jk}\ \ s_i\ \ \text{rcrc pm route prec} \mid \sum w_{jk} T_{jk}.$$

These vary significantly in size and complexity (see table 1). Largely, 4 groups can be differentiated.

i) The first instances $1-4$ are relatively small (up to 148 jobs) and can be solved easily. It is worth noting that no competitor was able to find a solution with 0 cost for the forth instance, but solutions corresponding to a cumulative delay of 0.08 minutes are easily found.

ii) The 5$^{\text{th}}$ instance 'with obstruction' is the hardest to solve and resulted being the sole criterion to separate between the top 5 entries in the SBB leader board.

The other instances - with the exception of 4 - are solved to 0 cost by the best 5 algorithms submitted. To solve instance 5, the routeing seems to be important, while precedence constraints are satisfied by all better solutions. Since I focused on this instance, I did not develop the incorporation of precedence constraints further.

Instance	#Jobs	Average #Operations / Job	Average #Routes / Job	#Connections
1	4	73.25	2	0
2	58	75.05	1.10	2
3	143	61.62	1.10	22
4	148	75.24	1.46	31
5	149	74.91	1.46	31
6	365	93.60	8.38	25
7	467	96.50	8.53	25
8	133	101.69	53.27	0
9	287	125.65	208.40	734

Table 1: Key indicators of the problem instances provided by the SBB.

iii) Instances $6, 7, 8$ range in size from 133 to 467 jobs. The number of possible routes is larger, but the instances are otherwise easy to solve. Even while restricting the routing to a randomly chosen route, a solution of 0 cost can be found. The precedence constraints (connections) seem to be full filled by all solutions of acceptable quality and do not need to be enforced.

iv) The last instance 9 contains many more precedence constraints, some of which are hard to satisfy. The routing choices of many jobs are too numerous for our simplistic approach.

Figure 9 displays the best results obtained with my implementation. Note, that the solution produced for instance 9 is not feasible, as I disregarded connection constraints. Adding the respective $10'000$ points penalty the present results would be rated 9^{th} rank in the long concluded competition (including post-challenge submissions). Restricting to instance 5, the algorithm ranks 6^{th}. Though it should also be mentioned that results of quality equal to algorithms ranked 6^{th} or 7^{th} is generally available after $30 - 90$ seconds (single core, 3.2 GHz).

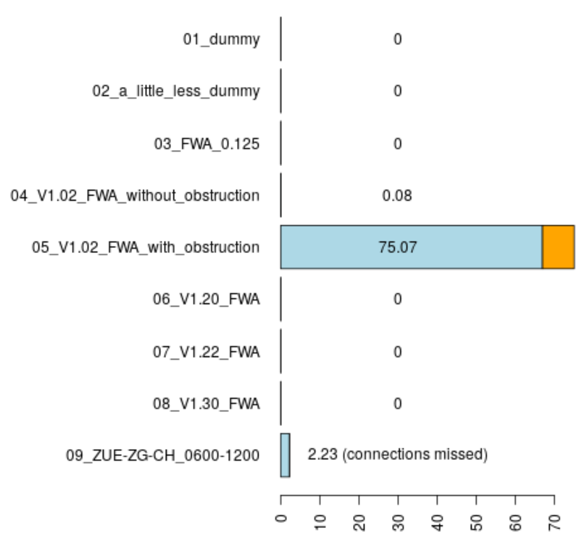


Figure 9: Penalties of the results obtained: The blue bars correspond to the delay penalty. The route-choice penalty is displayed in orange. The best results submitted to the SBB accumulate a cost of 37.45 for instance 5 and a cost of 0.1 for instance 4.

Conclusion

Given the results outlined above, I conclude that our approach was partially successful. The results are good enough to warrant the further development of our approach for problems of the given type. However, the results also show, that the method has clear shortcomings. In the next and last paragraph, I will mention two ideas to improve the method.

4.3 Future improvements

The precedence constraints of the connections and numerous routing possibilities clearly require more sophisticated treatment. However, here, I focus on the core method and the associated $J_m \mid \text{block} \mid \sum w_{Jk} T_{Jk}$ problem.

Two problems are identified:

i) The search is confined to solution relatively similar to each other. (A naive remedy is to restart algorithm several times, see eg, [21] where the authors restart a algorithm 10 times instead of allocating more computation time to the algorithm.)

ii) Even solutions close to the current solution are not explored, since multiple steps have to be taken, where the intermediary steps make the solution worse.

Diversification

Dr. Reinhard Bürgy is of the opinion that a good approach would consist the generation of a large number of initial solutions, that would then be refined with a local search. A key consideration is here, that we should guarantee, that the initial solutions are sufficiently diverse. I am convinced that the Kendall tau rank distance would be a good starting point to quantify the distance between (initial) solutions. The Kendall tau distance counts the swaps necessary to transform one permutation into another. Applying this to the sequences in which jobs use machines defines a distance, which is consistent with the algorithm. Each a move to a neighbor solution boils down to multiple calls of *SwapInMate()*, each of which corresponds to a swap in job-permutation of the respective machine. A simple implementation (based on merge-sort) computes the Kendall tau distance in $O(n \log n)$ time, where n is the length of the permutation. This would address the problem *i)*. For the problem *ii)*, I have the following suggestion:

Secondary critical arcs

The neighbours we generate swap out a single critical arc, without considering the potential gain of doing so. The potential gain, or the maximum reduction in delay of removing a critical arc, could easily be calculated. Assume job J_A is delayed significantly due to a critical arc from a job J_B. But, removing the critical arc $J_B \rightarrow J_A$ would immediately make another arc $J_C \rightarrow J_A$ critical and the decrease in delay would be marginal. Furthermore, it is likely that the sequence of removing $J_B \rightarrow J_A$, then removing $J_C \rightarrow J_A$ is not explored, as it is likely that the intermediary solution is worse (since there is not much to gain from the arc swap).

Aiming to improve a early version of the algorithm of Cesta & Oddi, Michel & Van Hentenryck[8] came to a similar conclusion: Their algorithm is based on iteratively applying ‘destruction’ and ‘construction’ phases. In a destruction phased some critical arcs are removed. Michel & Van Hentenryck predicted and observed that redoing multiple smaller destruction phases would lead to a improvement of the algorithm.

I assume, that an extension and more in-depth analysis of the critical subgraph is warranted. I.e., we could define: an arc a is *relevant* if the removal of critical arcs lead to a being critical. Relevant arcs are ordered in a hierarchy: arcs necessary to turn an arc into a critical arc are predecessors of said arc. We could then choose relevant arcs which correspond to a chain in the hierarchy such that the potential gain is maximized and the number of arcs selected is minimized (or, select at most 2, 3 arcs). A neighbour of a current solution is then defined as the removal of all arcs selected in such a way.

References

[1] Lawrence S (1984) Resouce constrained project scheduling: An experimental investigation of heuristic scheduling techniques (Supplement). *Graduate School of Industrial Administration, Carnegie-Mellon University.*

[2] Nemhauser GL, Wolsey LA (1988) Integer and combinatorial optimization. *John Wiley & Sons.*

[3] McCormick ST, Pinedo LM, Shenker S, Wolf B (1989) Sequencing in an assembly line with blocking to minimize cycle time. *Operations Research*, 37(6) 925-935.

[4] Carlier J, Pinson E (1994) Adjustment of heads and tails for the job-shop problem. *European Journal of Operational Research*, 78(2) 146-161.

[5] Hall NG, Sriskandarajah C (1996) A Survey of Machine Scheduling Problems with Blocking and No-Wait in Process. *Operations Research*, 44(3) 510-525.

[6] Mascis A, Pacciarelli D (2000) Machine Scheduling via Alternative Graphs. *RT-DIA-46-2000.*

[7] Mascis A, Pacciarelli D (2002) Job-shop scheduling with blocking and no-wait constraints. *European Journal of Operational Research*, 143(3) 498-517.

[8] Michel L, Van Hentenryck P (2004) Iterative Relaxations for Iterative Flattening in Cumulative Scheduling. *ICAPS*, 4 200-208.

[9] D'Ariano A, Pacciarelli D, Pranzo M (2007) A branch and bound algorithm for scheduling trains in a railway network. *European Journal of Operational Research*, 183(2) 643-657.

[10] Gröflin H, Klinkert A (2007) Feasible insertions in job shop scheduling, short cycles and stable sets. *European Journal of Operational Research*, 177(2) 763-785.

[11] Mazzarello M, Ottaviani E (2007) A traffic management system for real-time traffic optimisation in railways. *Transportation Research Part B: Methodological*, 41(2) 246-274.

[12] Gröflin H, Klinkert A (2009) A new neighborhood and tabu search for the blocking job shop. *Discrete Applied Mathematics*, 157(17) 3643-3655.

[13] Mati Y, Xie X (2010) Multiresource shop scheduling with resource flexibility and blocking. *IEEE transactions on automation science and engineering*, 8(1) 175-189.

[14] Oddi A, Cesta A, Policella N, Smith SF (2010) Iterative flattening search for resource constrained scheduling. *Journal of Intelligent Manufacturing*, 21(1) 17-30.

[15] Oddi A, Rasconi R, Cesta A, Smith SF (2011) Iterative flattening search for the flexible job shop scheduling problem. *In Twenty-Second International Joint Conference on Artificial Intelligence.*

[16] Oddi A, Rasconi R, Cesta A, Smith SF (2012) Iterative improvement algorithms for the blocking job shop. *In Twenty-second international conference on automated planning and scheduling.*

[17] Cacchiani V, Huisman D, Kidd M, Kroon L, Toth P, Veelenturf L, Wagenaar J (2014) An overview of recovery models and algorithms for real-time railway rescheduling. *Transportation Research Part B: Methodological*, 63 15-37.

[18] Lamorgese L, Mannino C (2015) An exact decomposition approach for the real-time train dispatching problem. *Operations Research*, 63(1) 48-64.

[19] Dabah A, Bendjoudi A, AitZai A (2016) Efficient parallel B&B method for the Blocking Job Shop Scheduling Problem. *International Conference on High Performance Computing & Simulation (HPCS)*, 784-791.

[20] Pinedo ML (2016) Scheduling: Theory, algorithms, and systems. Fifth Edition, *Springer International Publishing.*

[21] Pranzo M, Pacciarelli D (2016) An iterated greedy metaheuristic for the blocking job shop scheduling problem. *Journal of Heuristics*, 22(4) 587-611.

[22] Bürgy R (2017) A neighborhood for complex job shop scheduling problems with regular objectives. *Journal of Scheduling*, 20(4) 391-422.

[23] Bürgy R, Gröflin H. (2017) The no-wait job shop with regular objective: a method based on optimal job insertion. *Journal of Combinatorial Optimization*, 33(3) 977-1010.

[24] Dabah A, Bendjoudi A, AitZai A (2017) An efficient Tabu Search neighborhood based on reconstruction strategy to solve the blocking job shop scheduling problem. *Journal of Industrial & Management Optimization*, 13(4) 2015-2031.

[25] Samà M, D'Ariano A, Corman F, Pacciarelli D (2017) A variable neighbourhood search for fast train scheduling and routing during disturbed railway traffic situations. *Computers & Operations Research*, 78 480-499.

[26] Dabah A, Bendjoudi A, AitZai A (2019) Efficient parallel tabu search for the blocking job shop scheduling problem. *Soft Computing*, 23(24) 13283-13295.

[27] Train Schedule Optimisation Challenge. SBB CFF FFS. Crowd AI. `https://www.crowdai.org/challenges/train-schedule-optimisation-challenge` (Accessed June, 3rd, 2020)