Examining the critical path neighborhood for flexible job shop problems

Alex van Poppelen
Supervisor: Dr. J.A. Hoogeveen

January 29, 2016

Contents

1 Introduction

1.1 General introduction

The Job Shop problem looks at the problem of scheduling multi operation jobs on a set of machines, where each operation in a job needs to be processed on a certain machine. The Job Shop problem is well known to be NP-hard in the strong sense, and notoriously difficult to solve even compred to other NP-hard problems.

In this paper, the Flexible Job Shop (FJS) and the Flexible Blocking Job Shop (FBJS) problems are considered. These two problems are an extension of the Job Shop and Blocking Job Shop problems where individual machines required by each operation are replaced by a number of identical machines running in parallel. Because of the need to assign operations to machines, the FJS is more complex than the classical Job Shop problem. It is therefore also known to be strongly NP-hard [1].

In the remainder of the first section, the FJS is formally defined. The blocking constraint is also defined. In the second section, a local search approach to solve the FJS is explored. Finally, the local search approach is modified to accomodate the blocking constraints in the third section.

1.2 Formal definition

The Flexible Job Shop problem is defined as follows. We are given a set of n jobs.

$$\mathcal{J} = \{J_1, \ldots, J_n\}$$

Each job J_i consists of a fixed sequence of operations, where o_i refers to the number of operations in the job:

$$\mathcal{O}_i = \{O_{i,1}, \ldots, O_{i,o_i}\}$$

Where $\mathcal{O}$ refers to the set of all operations (in this paper, individual operations will sometimes be referred to with a single subscript; in this case it will be clear from the context whether this refers to a set of job operations or a single operation). Each operation $O_{i,j}$ must be processed on a predetermined machine type.

$$\mathcal{T} = \{T_1, \ldots, T_t\}$$

Furthermore, each operation $O_{i,j}$ has a processing time $p_{i,j} > 0$ and a delay time $w_{i,j} \geq 0$. Each machine type T_k has a given number of machines, defined as m_k. We will use $M_{k,l}$ to refer to the l-th machine of type T_k.

Machines are assumed to be continuously available from time 0, and can complete at most one operation at a time. Preemption is not allowed; the processing time $p_{i,j}$ must be uninterrupted. The delay time $w_{i,j}$ is defined as the time that must pass after operation $O_{i,j}$ is completed, before $O_{i,j+1}$ may begin. Note that the machine $O_{i,j}$ was being processed on is immediately available.

We are interested in the following objective function.

$$\min\{C_{\max}\}$$

Where $C_{\max}$ is defined as the maximum completion time, or the time at which the last machine finished. This will be referred to as the *makespan* of a schedule for the rest of this paper.

In some other literature describing the Flexible Job Shop problem, there is no concept of a machine type. Instead, each operation is defined to be processed on some subset of the set of all machines [1, 2]. This formulation is more general than the one used in this paper.

As a result, our definition of the FJS may exclude applications in settings that combine machines suited to highly specific tasks with more generalized machines, capable of processing a larger variety of operations.

1.3 Blocking constraint extension

In some situations, it may be unrealistic to model a scheduling problem as a Flexible Job Shop instance. For example, a manufacturing scenario might introduce storage space constraints. The FJS assumes that a job finished processing on one machine can be removed and placed into an intermediate buffer immediately, while waiting for the next machine to become available. This assumption may be impractical, and a factory might not have room for any intermediate buffers.

We can extend the problem definition to account for these constraints. In the Flexible Blocking Job Shop problem, a machine that is finished processing a job will be blocked until that job can be moved and processed by a successive machine. Note that this definition removes the distinction between the processing times and waiting times defined above, since the machine will be blocked for the duration of the waiting time regardless of whether subsequent machines are available or not.

This definition introduces a new questions that must be resolved. For example, what happens when machine $M_{1,1}$ is blocked, waiting for machine $M_{2,1}$ to be freed, while $M_{2,1}$ is blocked waiting for $M_{1,1}$? This depends on the problem scenario, and the mechanisms in place to transfer jobs between machines. In this paper, we will assume that these mechanisms are capable of simultaneously swapping jobs between all blocked machines. Thus, any *pure blocking cycles* will not cause a deadlock in the schedule.

2 A local search approach to the FSJ

In solving the FSJ, we are tasked with two problems. The first is to assign operations to machines. The second is to define an order on the set of operations assigned to every machine. We can choose to tackle these problems independently, or at the same time. In the independent approach, the operations would first be assigned to machines, reducing the problem to the classical Job Shop. In this paper, we will choose to tackle the problems at the same time, using a more complex schedule serialization function.

2.1 Representation

The classical Job Shop problem is often modeled as a disjunctive graph (for example, see **Figure 1**), where the directed arcs represent the precedence constraints within a job, and the undirected edges represent operations that must be processed on the same machine. By defining a direction on the edges, and making sure that these definitions result in an acyclic graph, one obtains a valid schedule for the Job Shop instance. In this case, an undirected edge represents a non-simultaneity constraint – the two operations may be processed in any order, but not at the same time. If we wish to use the same definition for the FSJ, we are forced to assign operations to machines before directing the edges. However, that splits the problem into two parts as mentioned above, and will increase the complexity of the neighborhood. We will take another approach.

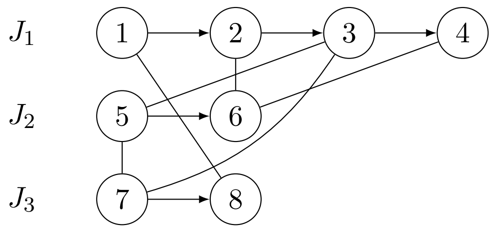

Figure 1: A disjunctive graph representing 3 jobs on 3 machines

We decide to modify the definition of an undirected edge. Instead, it will define two operations that may be processed in any order on the same machine, or independently on two different machines of the same type. Directing these edges will define the order of processing in the case the operations are on the same machine. In the case that they aren't, it only means that the start time of the second operation is equal to or greater than the start time of the first operation. This will make the serialization of a schedule more difficult, but the advantage is that generating the neighborhood will remain easy.

In our representation, we can ascertain the following information from **Figure 1**. There are three jobs. Operations 1 and 8 must be processed by the same machine type. Similarly, operations 2, 4, and 6 are processed on another, and 3, 5, and 7 on the last machine type.

2.2 Serializing the schedule

To generate a solution, all the edges in the disjunctive graph are given directions, such that the graph is acyclic. Once this has been done, it is possible to generate the corresponding schedule. The serialization function will assign operations to machines, and calculate the start time for each operation. The makespan can be determined after this process. It is important that the graph representation combined with the serialization function spans the entire set of feasible schedules. Furthermore, the serialization function should be deterministic in order to allow the local search to do its work.

To do this, we place the unscheduled operations in ordered queues for each machine type T_k. From these t queues, we take the first operation that has no unscheduled job predecessor, and schedule it greedily into the first available machine, taking care to schedule it at the same start time or after that of the previously scheduled operation for that type. This ensures the ordering on the operations is preserved in the schedule, and conforms to the requirements that were laid out earlier in the representation section. Scheduling operations before the starting time of a previously scheduled operation on the same machine type can lead to a schedule that is actually infeasible.

If there is no operation without an unscheduled job predecessor, then there exists a cycle in the graph, and the schedule serialization fails.

We could choose to apply several tricks to generate a better schedule. Instead, we will keep the serialization as simple as possible, and allow the local search do it for us by modifying the order of the operations for each machine type.

2.2.1 Schedule serialization example

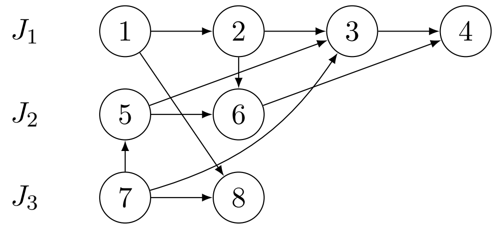

Figure 2: An acyclic directed graph representing a schedule

Consider the acyclic directed graph in **Figure 2**. The serialization function would set up three ordered queues:

$$T_1\colon O_1 \to O_8$$
$$T_2\colon O_2 \to O_6 \to O_4$$
$$T_3\colon O_7 \to O_5 \to O_3$$

Since operation O_2 has an unscheduled job predecessor, the algorithm would schedule O_1 or O_7 at time 0 on their respective machines first. After O_1 is scheduled, O_2 would become available to be scheduled at time $p_{1,1}+w_{1,1}$ or later. The algorithm continues until all the queues are empty or a cycle has been found. In the first case, the makespan is easily determined as the largest finish time for all the operations.

2.2.2 Waiting cycle

The acyclic property of the directed graph is crucial to the serialization process. It determines whether a configuration results in a feasible schedule or not. Suppose, using the above example, the arcs were modified to produce the following different queues for T_1 and T_3:

$$T_1\colon O_8 \to O_1$$
$$T_3\colon O_5 \to O_3 \to O_7$$

In this case, a cycle exists from $O_1 \to O_3 \to O_7 \to O_8$, and back to O_1. When the scheduler tries to serialize this configuration, it schedules O_5 as normal. Then it has O_8 to schedule on T_1, and O_3 on T_3. However, O_8 cannot be scheduled until O_7 is scheduled, and O_3 cannot be scheduled until O_1 (and O_2) is sceduled. This will be referred to as a *waiting cycle.*

2.3 Critical path neighborhood

For the classical Job Shop problem, a commonly used neighborhood is the set of all critical block swaps. The critical block swap is a reversal of an edge on the longest path of the disjunctive graph. This neighborhood is quite small, focusing on the operations that influence the length of the makespan. More importantly, the neighborhood is guaranteed to consist only of other acyclic graphs, or feasible solutions [3].

For the FJS, we can define the longest path in a similar manner. Whereas the longest path for the classical Job Shop problem was also chronologically sequential in terms of the generated schedule, this is no longer necessarily the case for the FJS. We must also consider operations on the same machine type, scheduled on different machines. For example, consider the operations $O_{1,1}$ and $O_{2,2}$ from different jobs that must be scheduled on machines of type T_k, connected by a directed edge from $O_{1,1}$ to $O_{2,2}$. Suppose $O_{1,1}$ has been scheduled at time t_1 on machine $M_{k,1}$ due to a precedence constraint.

Further suppose that machine $M_{k,2}$ is available from some time $t_2 < t_1$. However, the serialization function is forced to schedule $O_{2,2}$ on $M_{k,2}$ at time t_1 anyway. In this scenario, we would like to reverse the order of the arc between $O_{1,1}$ and $O_{2,2}$, so that $O_{2,2}$ may be scheduled earlier and therefore potentially decrease the length of the longest path.

Therefore, we construct our neighborhood by considering operations v and w such that:

- v and w have both been scheduled consecutively on the same machine
- v and w have been scheduled on different machines of the same type at the same time

2.3.1 Applying the swap

Let's consider the ordered queues containing the set of operations on a machine type, as defined by the acyclic directed graph corresponding to a feasible schedule. In the classical Job Shop case, a critical block swap identifies two consecutively scheduled nodes on a machine. These two nodes are necessarily consecutive in the queue, as well. In the FJS problem, this is no longer necessarily the case, since operations in between may have been scheduled on other machines of the same type. If one of these other operations is of the same job as either of the selected nodes, simply reversing the directed edge between the two nodes will lead to one or more cycles.

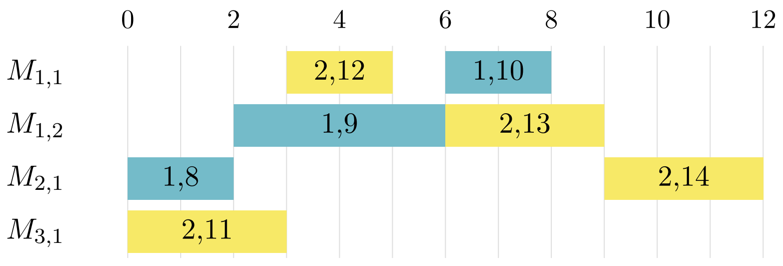

Figure 3: A flexible job shop schedule

Consider the schedule shown in **Figure 3**. The ordered queue for T_1 that generated this schedule follows:

$$O_{1,9} \rightarrow O_{2,12} \rightarrow O_{1,10} \rightarrow O_{2,13}$$

We wish to swap the order of operations $O_{1,9}$ and $O_{2,13}$, both on the critical path. Simply swapping their places leads to cycles, since the job operations will be out of order. We can resolve this situation by reversing a few more arcs, to achieve the following ordering.

$$O_{2,12} \rightarrow O_{2,13} \rightarrow O_{1,9} \rightarrow O_{1,10}$$

This can be done by incrementely swapping both target operations inwards, stopping if the swap would occur with another operation of the same job, or if the two target operations have switched order. This preserves the order among all operations exluding the two targets. Although this swap is now bounded by $O(n)$ in the worst case, in practice it will be much faster, since the two operations being swapped are chronologically sequential. It can be shown that this type of swap for two operations on the longest path can always be done without creating any cycles within the machine type.

Suppose the method above was not able to swap two operations, O_{i,a_1} and O_{j,b_2} for $i \neq j$. Then there exist operations O_{i,a_2} and O_{j,b_1} with $a_1 < a_2$ and $b_1 < b_2$ such that the ordered queue is equivalent to the following:

$$\cdots \to O_{i,a_1} \to \cdots \to O_{i,a_2} \to \ldots O_{j,b_1} \to \cdots \to O_{j,b_2} \to \ldots$$

This configation does not allow for the above swap without changing the order between the two inner operations, since it would create a cycle. However, it is easy to see that such an ordering will never create a schedule such that O_{i,a_1} and O_{j,b_2} are consecutive on the longest path, since operation O_{j,b_1} has a non-zero processing time, and cannot be scheduled until after or equal to the finish time of O_{i,a_1} due to O_{i,a_2}.

2.3.2 Feasibility

The critical path neighborhood for the classical Job Shop is guaranteed to consist of only feasible schedules (acyclic graphs), a very nice result from Van Laarhoven et al [3]. Unfortunately, this is not the case for the Flexible Job Shop. We have slightly modified the swap operation to ensure no waiting cycles within a machine type are created. However, this does not guarantee no waiting cycles in the entire graph are generated. Consider the schedule shown in **Figure 4**, where colors correspond to jobs.

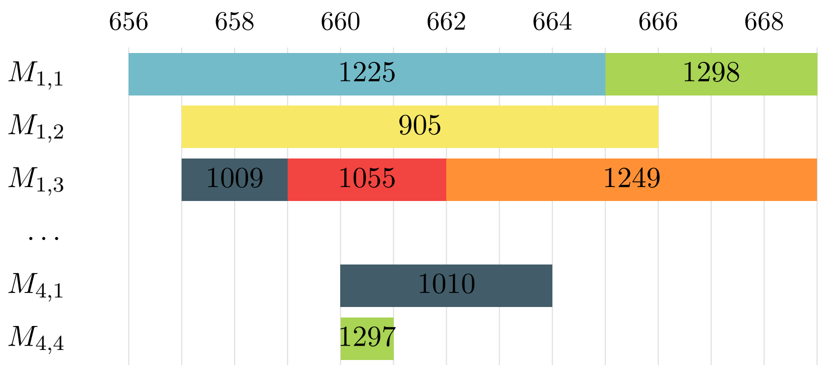

Figure 4: A flexible job shop schedule

The acyclic directed graph that generated this schedule could have the following partial ordered queue for machine type T_1:

$$\cdots \to O_{1225} \to O_{905} \to O_{1009} \to O_{1055} \to O_{1249} \to O_{1298} \to \dots$$

The ordering on T_4 could have O_{1010} placed before O_{1297}. Suppose that O_{1225} and O_{1298} were on the critical path, and have been selected for a swap. Applying the swap operation as described above would simply move O_{1298} to the front of the chain. A cycle is created, from $O_{1010} \to O_{1297} \to O_{1298} \to O_{1009}$ and back to O_{1010}.

In this case, we could perform checks and swap the two operations after the offending O_{1009}. However, correcting for one cycle could potentially lead to another, and the operation would become very expensive. In practice, these kinds of cycles rarely occur, and it is better to simply reverse the swap and allow the heuristic to try something else.

2.4 Computational results

Unfortunately, given the more restrictive formulation, there are no benchmark instances or results from the literature to compare to. Classical Job Shop instances do fit in our formulation, so a few have been included just for reference. These instances, *ft06*, *ft10*, and *ft20* were generated by Fisher and Thompson (available on the web [4]).

The next set of instances come from a previous project on exact methods for the FJS [5], therefore including the optimal values. These instances are labeled *c1 – c19.* It should be noted that in almost all of these instances, the optimal makespan is equal to the maximum job makespan. This makes it easy to prove optimality, and also much easier to find an optimum.

The rest of the instances were generated using a random instance generator built for this purpose. These are labeled *v20 – v40.* The optimal makespans for these instances is unknown. However, they do provide insight into the convergence behavior, especially when compared between instances.

The local search was implemented in C, using the GCC compiler with the -O3 flag. Each configuration was run five times, to generate an average best makespan and execution time. These results can be examined in **Table 1**.

Most of the parameters in the table are self-explanatory. The times are given in seconds, for an Intel Core i3 M370 @ 2.40 Hz CPU. α is the cooling parameter. It is responsible for lowering the tempterature of the simulated annealing process. A value closer to 1 means the temperature cools more slowly, and thus the heuristic searches longer. The restarts column indicates how many times the temperature was reset. The average best makespans are indicated by $\bar{C}$, and the average running times by $\bar{t}$.

Instance				Properties		Results				
name	$\lvert\mathcal{J}\rvert$	$\lvert\mathcal{T}\rvert$	$\lvert\mathcal{M}\rvert$	$\max C_{\text{job}}$	C_{opt}	α	restarts	$\bar{C}$	C_{best}	$\bar{t}$
ft06	6	6	6	47	55	0.9	1	55	55	0
ft10	10	10	10	655	930	0.9	5	954	941	2
						0.99	5	943	938	18
						0.999	5	933	930	182
ft20	20	5	5	387	1165	0.9	5	1215	1186	1
						0.99	5	1182	1178	13
						0.999	5	1170	1165	130
c1	4	2	4	34	34	0.9	1	34	34	0
c2	4	2	4	53	53	0.9	1	53	53	0
c3	4	2	4	133	143	0.9	1	143	143	0
c4	6	2	4	52	57	0.99	5	57	57	2
c5	6	2	4	105	113	0.99	5	113	113	2
c6	6	2	4	205	225	0.99	5	225	225	2
c7	4	3	9	39	39	0.9	1	39	39	0
c8	4	3	9	64	64	0.9	1	64	64	0
c9	4	3	9	138	138	0.9	1	138	138	0
c10	6	3	9	50	50	0.9	1	50	50	0
c11	6	3	9	103	103	0.9	1	103	103	0
c12	6	3	9	204	204	0.9	1	204	204	0
c13	8	3	9	337	337	0.9	5	337	337	0
c14	8	3	9	705	705	0.9	5	705	705	0
c15	8	3	9	1458	1458	0.9	5	1458	1458	0
c16	10	4	16	405	405	0.9	5	405	405	0
c17	10	4	16	955	955	0.9	5	955	955	0
c18	10	4	16	1621	1621	0.9	5	1621	1621	0
c19	15	5	25	2631	2631	0.9	5	2631	2631	0
v20	8	2	4	176		0.9	5	387	387	1
						0.99	5	387	387	7
						0.999	5	387	387	67
v21	8	2	8	187	187	0.9	5	204	188	0
						0.99	5	197	188	4
						0.999	5	188	187	25
v22	8	4	8	195		0.9	5	237	237	1
						0.99	5	237	237	9
						0.999	5	237	237	82
v30	16	2	4	666		0.9	5	2382	2381	10
						0.99	5	2381	2381	102
						0.999	5	2381	2381	1045
v31	16	2	8	566		0.9	5	963	961	12
						0.99	5	960	959	126
						0.999	5	958	956	1238
v32	16	4	8	655		0.9	5	1119	1116	11
						0.99	5	1112	1111	116
						0.999	5	1108	1106	1164
v33	16	4	16	626	626	0.9	5	628	626	5
						0.99	5	657	651	84

Table 1: Computational results for the FJS

2.5 Analysis

For the Job Shop instances (*ft06*, *ft10*, and *ft20*), the code performs as well as previous results in literature [3]. This gives us confidence that the code is robust.

The series of instances from the previous project (*c1* – *c19*) are largely uninteresting, due to there being too few jobs in relation to machines. For the few instances whose optimal makespans were larger than the lower bound given by the job makespans, the search had significantly more trouble finding the optimum, despite being much smaller than some of the other instances. In several cases, the search failed to find an optimal solution despite the restarts, being very slightly off. However, this was averaged out by the other results. It should be noted that in some of these cases, the algorithm actually became stuck in a small area of the solution space, with the only neighborhood solution available being infeasible. This is very rare, but it highlights the importance of random restarts when using the critical path neighborhood for the FJS.

For the other instances, we see that the algorithm converges very quickly. More searching failed to find solutions significantly below the best solutions found almost immediately. This indicates the critical path neighborhood works very well for the Flexible Job Shop problem. As the solution improves, the size of the neighborhood becomes smaller and smaller. This leads to very fast converging behavior.

It is also interesting to note that the amount of machine types and total machines have little to do with the "difficulty" of the instance. The algorithm converged equally quickly for all machine type configurations, with the amount of jobs and operations held constant.

3 A local search approach to the FBSJ

The Flexible Blocking Job Shop adds the blocking constraint to the problem. This constraint adds a new degree of complexity to the problem, and is notoriously difficult to solve. Any optimal makespan found will be at least as large as its non-blocking counterpart. Completing jobs in the shortest timespan possible suddenly becomes very important, in order to reduce the total time a machine spends being blocked.

3.1 Serialization

Intuitively, we can use the exact same representation and critical path neighborhood for the FBJS problem as for the FJS. The serialization function will have to be modified slightly to account for the blocking constraints. This is relatively simple – the greedy algorithm will simply mark a machine as blocked after scheduling an operation, and unmark it when its successor is scheduled.

As mentioned in the introduction, the blocking constraint introduces new types of cycles.

3.1.1 Pure blocking cycle

An example of a pure blocking cycle is one where operation $O_{j_1,a}$ has been scheduled on machine $M_{t_1,m}$ and $O_{j_2,b}$ has been scheduled on $M_{t_2,n}$. Both of these machines are now blocked. Suppose there are no other available machines for machine types T_{t_1} and T_{t_2}, and the scheduler now encounters $O_{j_2,b+1}$ to be scheduled on type T_{t_1} and $O_{j_1,a+1}$ for T_{t_2}. Since the only possible machines for these types are blocked, this can only be resolved if the two machines can swap jobs simultaneously, as described in the introduction. Since we allow for blocking with swaps, the scheduler must recognize this type of cycle and schedule the two (or more) operations simultaneously.

Unfortunately, this cycle does not show up in our disjunctive graph representation, so we cannot detect it using traditional graph algorithms. We can choose to extend our representation, and indeed this may be the most complete choice. However, the scheduler will encounter maximally 1 operation to be scheduled on each machine type at once. Thus these cycles are guaranteed to be small and quite simple. By running through the blocked operations in $O(n^3)$ time, the scheduler can reliably identify these cycles and resolve them.

3.1.2 Variable cycle

A variable cycle is a mix of waiting and blocking. For example, suppose operation $O_{j_1,a}$ has been scheduled on machine $M_{T_i,k}$, the only available machine of its type. It is now blocked, and cannot schedule the next operation, $O_{j_2,b}$. However, this means T_j cannot schedule its next operation, $O_{j_2,b+1}$. $O_{j_1,a+1}$ is unscheduled and ordered after $O_{j_2,b+1}$, creating a deadlock. One machine is blocked, while the other is waiting.

This is an infeasible solution and cannot be resolved without reversing some edges. In this paper, we simply reverse the operation leading to this configuration, and try something else.

3.2 Naive shift neighborhood

There is little literature on the FBJS, and as a result there are even few benchmark instances and results. In any case, these would not be applicable here due to the less generalized formulation we have used. To better assess the critical path neighborhood, we introduce a naive shift neigbhorhood for comparison.

In this neighborhood, we take a random operation of a random machine type, and shift it a random number of (permissible) places left or right. It may also take all of operations of the job this operation belongs to, and shift them all an individually random number of places left. The blocking constraint means that the operations belonging to a job must be executed as close together as possible. In this sense, it makes sense to move these operations as a unit, rather than individually.

While this neighborhood somewhat takes the nature of blocking and jobs into account, it doesn't use any information concerning the critical path, waiting and/or blocking times, or other useful properties. It is therefore referred to as a naive neighborhood.

3.3 Computational results

The same instances are used as before. The restarts were held constant at five. A new column, containing the best non-blocking solution for the instance, is added for comparison. The results can be explored in **Table 2**.

No running times are listed, but in general they are in the same order as for the FJS (a bit more, the blocking constraints do add some extra computation).

3.4 Analysis

It is immediately apparent that for almost all instances, the critical path neighborhood fails to search large portions of the solution space. It quickly converges to a certain makespan, and then fails to improve on it – even if it is very far off the optimal solution. Often, this makespan is even worse than one reached by the naive shift neighborhood. In fact, the critical path neighborhood only performed better on large instances. This is likely due to the fast convergence compared to the naive shift.

Neither of the neighborhoods seem to reach good solutions at all, except in the case of an excess of machines. The combination of the two neighborhoods performs better than either of the neighborhoods alone, but the solution quality and speed of convergence still leave much to be desired.

Analyzing the critical path neighborhood reveals a very high percentage of infeasible solutions in the neighborhood. The critical path neighborhood is quite small, which is a strength in the FJS case. However, this means it explores very little of the solution space, and the search quickly becomes stuck due to the large amounts of infeasible solutions. Combining the two neighborhoods improved the search somewhat. In any case, it appears as though moving individual operations on the critical path simply creates infeasible solutions in too many cases.

The blocking constraint means that good solutions will have jobs packed tightly together, in order to minimize the times that machines are blocked

	FJS	FBJS		Critical Path		Naive shift		Both	
name	C_{best}	C_{opt}	α	$\bar{C}$	C_{best}	$\bar{C}$	C_{best}	$\bar{C}$	C_{best}
ft06	55	63	0.99	102	95	64	63	66	63
ft10	930	1068	0.9	2183	1931	1507	1436	1464	1415
			0.99	2139	1931	1444	1353	1413	1349
			0.999	2366	2028	1380	1281	1371	1292
ft20	1165		0.9	2476	2406	1957	1866	1935	1869
			0.99	2454	2348	1840	1814	1860	1840
			0.999	2393	2209	1694	1646	1714	1632
c1	34		0.99	66	62	37	37	37	37
c2	53		0.99	88	75	65	65	65	65
c3	143		0.99	206	206	180	180	180	180
c4	57		0.99	116	109	94	92	91	90
c5	113		0.99	247	215	180	170	175	170
c6	225		0.99	502	434	365	360	360	353
c7	39	39	0.99	39	39	39	39	39	39
c8	64	64	0.99	64	64	64	64	64	64
c9	138	138	0.99	138	138	138	138	138	138
c10	50	50	0.99	50	50	50	50	50	50
c11	103		0.99	170	116	106	106	106	106
c12	204		0.99	265	210	208	208	208	208
c13	337		0.99	557	524	467	458	393	389
c14	705		0.99	1203	1036	866	833	797	788
c15	1458		0.99	2556	2258	1789	1666	1508	1493
c16	405	405	0.99	519	483	444	429	405	405
c17	955	955	0.99	1257	1009	962	955	955	955
c18	1621	1621	0.99	2314	2131	1710	1688	1621	1621
c19	2631	2631	0.99	4568	3973	4365	4049	2631	2631
v20	387		0.9	571	540	494	476	438	423
			0.99	527	517	447	421	424	418
			0.999	557	497	427	425	419	416
v21	187		0.9	331	293	277	254	218	215
			0.99	327	291	213	209	198	194
			0.999	335	313	196	194	195	194
v22	237		0.9	419	387	326	314	294	284
			0.99	393	369	293	281	273	262
			0.999	413	367	273	271	264	259
v30	2381		0.9	4210	4068	5969	5856	3546	3438
			0.99	4200	4154	5591	5411	3287	3147
			0.999	4150	4142	5339	5232	3334	3276
v31	959		0.9	1658	1573	4303	4133	1702	1635
			0.99	1688	1632	3772	3712	1646	1617
			0.999	1702	1656	3553	3444	1479	1391
v32	1111		0.9	2570	2389	5214	5054	2570	2389
			0.99	2629	2601	4481	4209	2171	1983
			0.999	2595	2451	3614	3005	1953	1907
v33	626		0.9	1296	1212	3182	2931	1098	1053
			0.99	1335	1244	2061	1903	952	894

Table 2: Computational results for the FBJS

instead of processing. Attempting to move one operation out of a tightly packed job to a new time in the schedule will almost inevitably lead to a variable cycle, and an infeasible solution. This is most likely the cause of the failure of the critical path neighborhood when applied to the Blocking Job Shop. It is clear a different approach is needed, perhaps one that looks at entire jobs instead of individual operations.

4 Conclusions and further research

We have examined the critical path neighborhood in conjunction with a local search for the Flexible Job Shop and Flexible Blocking Job Shop problems.

For the Flexible Job Shop problem, the critical path neighborhood is a very effective search mechanism, performing nearly as well as it does for the classical Job Shop. It doesn't provide the same feasibility guarantees, but in practice this is hardly an issue. With a small number of random restarts, the search will almost invariably find a good solution, and converge very quickly. The strength of this neighborhood lies in its small size and the clever use of the problem characteristics.

These strengths quickly turn into weaknesses when the blocking constraint is added to the problem. The critical path neighborhood fails to effectively search the solution space, and often performs more poorly than even the most primitive neighborhoods. Even when combined with a different neighborhood to help it explore different areas of the solution space, it fails to make a significant impact on the solution quality and the rate of convergence. A different approach is needed, one that takes important relationships between operations of the same job into account.

There are several options here. One is to adopt a job removal/insertion technique, similar to one used by Groeflin and Klinkert for the Blocking Job Shop [6]. This is expensive, but ensures feasible solutions. Another is to temporarily allow infeasible solutions. In any case, the representation needs to be extended to account for the blocking constraints, in the form of additional edges. The lack of such a structure in the implementation used in this paper made it exceedingly difficult to recognize and deal with variable or blocking cycles.

To better test new neighborhoods, it's advisable to extend the FJS formulation to the one more commonly found in literature, where operations may be processed on some subset of all machines, instead of non-overlapping machine types. This, combined with benchmark instances, will allow for better comparison and assessment.

References

[1] Klaus Jansen, Monaldo Mastrolilli, and Roberto Solis-Oba. “Approximation algorithms for flexible job shop problems”. In: *LATIN 2000: Theoretical Informatics* 1776 (2000), pp. 68–77.

[2] H. Groeflin, D. Pham, and R. Burgy. “The flexible blocking job shop with transfer and set-up times”. In: *Combinatorial Optimization* 22 (2011), pp. 121–141.

[3] Peter J.M. Van Laarhoven, Emile H.L. Aarts, and Jan Karel Lenstra. “Job shop scheduling by simulated annealing”. In: *Operations Research* 40.1 (1992), pp. 113–125.

[4] J.E. Beasley. *OR-Library: Job shop scheduling.* Jan. 2016. URL: `http://people.brunel.ac.uk/~mastjjb/jeb/orlib/jobshopinfo.html`.

[5] C.P. Cijvat. “Exact solution methods for the flexible job shop problem using column generation”. In: *Experimentation project* (2015).

[6] H. Groeflin and A. Klinkert. “A new neighborhood and tabu search for the blocking job shop”. In: *Discrete Applied Mathematics* 157 (2009), pp. 3643–3655.