Contrasting Search Algorithms implemented on the Job-Shop Scheduling Problem

Megan Thurber
Department of Computer Science
Colorado State University
thurber@cs.colostate.edu

July 29, 2004

Abstract

Many scheduling problems are complicated and hard to solve. This has motivated the study of search algorithms. This paper will focus on the Job-Shop Scheduling Problem. Local Search, Tabu Search, GRASP and Large Neighborhood Search are the four main algorithms discussed here. Local Search was implemented with two different operators: N1 and N5. The original GRASP was later written as a Reactive GRASP implementation. Large Neighborhood Search was not completed in time for this paper, but is discussed. The purpose here is to determine which algorithms work best on different problems. The algorithms were tested on twenty-six different Job-Shop Problems ranging in size and difficulty.

Keywords: Job-Shop Problem (JSP), makespan, Local Search, Tabu Search, GRASP, Reactive GRASP, Large Neighborhood Search (LNS)

1 Introduction

Computers today process tasks quickly. Despite this, they are often unable to complete all tasks in the alloted amount of time. To amend this problem, computer scientist started to implement scheduling algorithms. In general, scheduling is placing a set number of tasks in the best order possible within problem constraints and available resources. These resources are always limited (for example types of machines, memory space, time, etc.) and the ordering of the tasks can be crucial. In real world scheduling it is common to be unsuccessful in scheduling all the tasks. Such a problem would be considered *oversubscribed* and in this case the algorithm must select the tasks it will not complete.

Since most real-world scheduling problems are detailed, there are a large number of variables. Computer scientist have found it more useful to group the problems together and extracted a set of "distilled" problems to work with.[8] These problems are more general and represent a class of real world problems. Finding a solution to one of these problems could be applicable to all the real-world problems in that class. When attempting to solve these problems the algorithms must abide by all the constraints given in the problem definition while trying to minimize the time and resource usage. At times there are constraints that can be relaxed in order to accomplish a better schedule. These are called soft constraints. Hard constraints are those which must always be observed.

2 Job-Shop Scheduling Problem

A specific and well known class of scheduling problems is the Job-Shop Problem (JSP). In these problems there are n jobs and m machines. Each job has an operation assigned to every machine that must be performed in a specific order. Every operation must wait for the previous operation on that machine and the previous operation within its job to be completed before it can start to run. A solution to this problem is a completed schedule, that is one including all the operations.

Why is this problem not considered a real world scheduling example? There are many characteristics of JSP that contribute to its impracticality. All information about the operations is defined in the problem statement. A JSP problem specifies information that may be unpredictable in a real world situation. First the order of the machines for the operations of every job are laid out, as well as the run time of each operation. Also a list of the jobs is available before the program starts running. Even looking past all the given information there are still many things limited by the problem definition. Each job runs on a machine once and only once. There is only one machine of each type. Even though the problem is concerned with timing, the beginning and ending times are not specified for any operations. Two operations from one job can not be running at the same time on different machines. All the operations are non-preemptive, meaning they all have equal priority and will never be interrupted by another operation. [8]

2.1 Makespan

The objective in writing scheduling algorithms for problems such as JSP is to obtain some optimum. In most cases with JSP the goal is to minimize the *makespan*. All operations have a start and finish time. The start time is determined by the previous operation within that job and the previous operation on that machine. Every operation will start as soon as both of the operations it depends on have been completed. The finish time of an operation is simply the start time plus the duration of that operation. A makespan is the total time it takes for all the operations to be completed based on the start and finish times. This is minimized by changing the order of the operations on any given machine without violating any of the predefined constraints. [4]

2.2 Critical Path

Changing the order of operations on a machine is called a swap. Only operations along a critical path can be swapped. A critical path is a set of the operations that create a connected sequence from the start time of zero to the finish time of the makespan. Once a critical path is found it is broken down into critical blocks, which are the contiguous sections of the path that lie on the same machine. Any operation within a critical block is then called a critical operation. A move is the swapping of two adjacent critical operations on a machine. Any critical block containing more than two operations will consist of at least one move.

3 Search Algorithms

There are many different algorithms applied to scheduling problems. The purpose of a search algorithm is to find the best solution possible. It may be limited in many ways. Frequently the main goal is efficiency and speed. There are times when one search may guarantee the correct answer, but uses too many resources or performs too slowly. In most cases simpler algorithms run very quickly, but may not give the best results. The search algorithms considered here have been limited to thirty minutes of run time. This does not effect the result of the first two algorithms at all; however, as the algorithms become more complicated they may not complete the maximum number of iterations, which is 10,000 for these problems except for GRASP since the construction phase takes such a large amount of time.

3.1 Local Search

One of the simpler search algorithms is Local Search. The main idea of this algorithm is to form a local neighborhood around the current solution. After evaluating all the neighbors it will choose the best direction for the search to proceed. The algorithm starts by creating an initial solution. This implementation uses a random permutation of the operations on the machines. Then the Local Search begins an iterative step where it looks for the first improving neighbor to replace the current solution. A neighbor of the current solution in defined as a permutation within one move of the current solution. If a move will decrease the makespan it is considered an improving move. The move that decreases the makespan the most will be selected as the best move in that iteration. The success of the search is most often dependent on how "good" this initial solution is. A permutation is considered to be a "good solution" if it is relatively close to the optimal solution. [6]

```
//Read in the problem and return the best solution
function LocalSearch(Problem)
solution ← initialState()
while ( k ← 0 to maxIterations)
    bestMove ← findNeighbors()
    solution ← swap(bestMove)
return solution
```

Figure 1: Pseudocode for the LocalSearch

3.1.1 N1 Operator

This local search was implemented using two operators, the first of which is called N1. N1 considers all possible permutations in the neighborhood. The moves that are considered will consist of every possible pair of adjacent operations on each critical block. The operator does not look at moves outside of a critical block since swapping them will have no effect on the makespan. This makes the search space completely connected. That means that from any possible solution there is a list of moves that will find the optimal solution.

3.1.2 N5 Operator

The second operator is N5 which considers fewer moves. It will only swap the first pair and last pair of operations on each critical block. For smaller problems this will often yield the same solutions. Critical blocks must contain four operations or more in order for there to exist moves considered by N1, but omitted by N5. There are also two other cases where N5 will omit moves: this operator will not swap the first move on the first critical block or the last move on the last critical block. This search space is not completely connected. It may be impossible to find the optimum from the initial random state. Examples of this can be seen even in small problems. The JPW43 is a problem created by Jean-Paul Watson that consists of four jobs and three machines.[8] All of the

Problem	Best Known	N1	Percent	N5	Percent	N1-N5
abz05	1,234	1,408	14	1,349	9	59
abz06	943	1,083	15	1,071	14	12
abz07	656	850	30	804	23	46
abz08	669	865	29	837	25	28
abz09	679	983	45	901	33	82
ft06	55	59	7	58	5	1
ft10	930	1,164	25	1,152	24	12
ft20	1,165	1,688	45	1,613	38	75
jpw43	32	32	0	32	0	0
la01	666	741	11	715	7	26
la02	655	762	16	736	12	26
la06	926	926	0	927	0	-1
la07	890	973	9	940	6	33
la15	1,207	1,415	17	1,341	11	74
la21	1,046	1,422	36	1,313	26	109
la24	935	1,178	26	1,144	22	34
la25	977	1,240	27	1,303	33	-63
la26	1,218	1,388	14	1,446	19	-58
la27	1,235	1,707	38	1,702	38	5
la29	1,153	1,549	34	1,484	29	65
la31	1,784	1,968	10	1,949	9	19
la36	1,268	1,617	28	1,561	23	56
la38	1,196	1,504	26	1,523	27	-19
la40	1,222	1,702	39	1,566	28	136
orb08	899	1,200	33	1,229	37	-29
orb09	934	1,205	29	1,147	23	58

Table 1: Comparing Local Search Operators

search algorithms can find the optimal solution for this problem, however it it not uncommon for the local search implemented with the N5 operator to return a makespan of 33 (one more than the optimum.) This occurs when the first move of the first critical block needs to be swapped to find the optimal solution. Since N5 does not consider this move any initial state with the first two operations of a critical path out of order will not be able to find the optimal solution. This is one reason it is disconnected.

Table 1 shows the results for the Local Search algorithm implemented with both operators. The Percent column represents 1 - (search best solution over the best know solution), which returns the percentage the algorithm's solution is off from the best known. The last column takes the difference of the best solution of the N1 operator and the best solution of the N5 operator. The N5 operator performs better in all but six cases. Despite the fact the N5 does better than N1, the solutions it finds are still off by more than 10 percent of the optimal makespan in most cases.

3.2 Tabu Search

Like local search, Tabu Search begins with a random permutation and runs through each iteration looking at the local neighbors and choosing the best one to follow. It differs by tracking the recent moves of the search in a Tabu List. When the algorithm is searching for the neighbors of the current solution, it will not consider moves that are on the Tabu List. This will allow the algorithm to keep itself from back tracking. The size of the Tabu List is a very important factor in algorithm. If the number is too small, having the list will have little or no effect on the algorithm's path. However, if the number of elements in the Tabu List is too large the search will be restricted greatly and in turn find worse solutions.[3] In some implementations the algorithm will allow moves to be taken even if they are on the Tabu List. If this occurs the algorithm must have a set *aspiration* value. If the current *aspirationlevel* is less than the aspiration value and the move being considered will create a solution that improves on the current best solution then it will be accepted even if it is on the Tabu List.

```
//Read in the problem and returns the best solution
function TabuSearch(Problem)
solution ← initialState()
while (k ← 0 to maxIterations)
    bestMove ← findNeighbors()
    solution ← swap(bestMove)
    updateList()
    //Performs random restart ever 20 iterations
    if (k mod 20 = 0)
        if(makespan(solution) < globalMakespan)
            globalMakespan ← makespan(solution)
            bestSolution ← solution
        solution ← initialState()
return bestSolution
```

Figure 2: Pseudocode for the TabuSearch

There are different ways to implement the Tabu Search that may lead to smaller makespans. One possibility is to allow the algorithm to accept non-improving moves. This may seem counterintuitive, however it will permit the algorithm to expand further into the search space. Another possibility is to have the search randomly restart after a set number of iterations. The idea behind this is the same, in that it will allow the search to investigate a larger number of the solutions. This implementation using the later of the two alterations. Every twenty iterations the Tabu Search will start over from another random location. This improves the search results drastically. Random restarts allows more of the search space to be viewed by the algorithm than merely tolerating non-improving moves. This is not surprising since non-improving moves will still keep the solution close to the start point. Since random restarts changing the starting point it will broaden the search space much more.

3.3 GRASP Search

GRASP stands for Greedy Randomized Adaptive Search Procedure. This search is different from the previous two since it does not start with a random solution. Rather GRASP has two phases: first it builds a solution, then it performs local search to improve the solution. Every iteration of the search the algorithm will create a solution. The construction phase is iterative as well; in each of these iterations an operation is added to the partial solution.

In order to construct a solution the algorithm makes a Restricted Candidate List (RCL). This list contains a set number of elements which are considered the "best" operations to schedule. The number of elements in the RCL is dependent on the α value. The "best" operations are the ones that will increase the makespan the least when added to the current schedule. Once the RCL is full of the "best" operations, one of them is randomly selected to be added to the schedule. Then the RCL is reevaluated. This process continues until all the operations have been added to create a complete solution. At that point local search is performed on the solution in an attempt to improve the result. This occurs every iteration of the search and the best solution is taken at the end. In this case every run of the search will have a different level of greediness and will therefore create a large range of solutions. [2]

3.3.1 The α Parameter

The greediness of this solution is based upon a value α which lies between zero and one. Zero is a completely greedy construction; as α increases the construction becomes more random. The α value determines the greediness by altering the size of the RCL, which is calculated by multiplying α by m and n. In the primary GRASP algorithm this value is chosen randomly for each run. This gives the algorithm a different level of greediness each run. When the RCL size is set to a small number, it will include less operations. Since it always includes the "best" operations having less to choose from makes it greedier. In the case when the RCL size is let to a large number, the largest possible number being $m * n$ (for α equal to 1), it will contain more operations. This makes the algorithm much more random since it could schedule the operations in a variety of ways.

3.3.2 Reactive GRASP

In Reactive GRASP, the α value changes for each iteration of the search. In this implementation there are $m + 1$ values for α stored in a list $\psi = \{0, 1/m, 2/m, ..., 1\}$. The alpha value for the first iteration is still random, but in each iteration thereafter, α is chosen intelligently. In the first m iterations all the values of ψ are selected to ensure that each α has been used at least once. An average is computed for each α by adding all the makespans that α value has created and dividing it by the number of times that α has been selected. So after the first m iterations the average for each α will simply the makespan created by that α in its first run. Then a new α will be randomly selected; however, the α values that have lower averages will have higher probability of being selected in the following iterations. [5]

```
//Read in the problem and returns the best solution
function ReactiveGRASP(Problem)
for(i ← 0 to m + 1)
    α ← i / m
    ψ[i] ← α
α ← ψ[index]
while ( k ← 0 to maxIterations or time ← 0 to maxTime)
    solution ← constructSolution(α)
    localSearch(solution)
    if(k < m)
      α ← ψ[index + +]
    else
        bestAlpha ← createBestAlpha()
    α ← bestAlpha
return solution
```

Figure 3: Pseudocode for the ReactiveGRASP

3.4 Large Neighborhood Search

By the name of this algorithm it is clear that the search will no longer be looking on a local level, but rather at the entire problem. Once again this Large Neighborhood Search will start with a random permutation as its initial solution. Instead of swapping operation this search removes them from the solution completely and then re-inserts them into a different location. The operations that are removed are not random, rather they are chosen by their relation to the other operations. Determining how closely two operation are related can be computed in numerous ways. [7]

In JSP relatedness is determined by which machine an operation is on. The algorithm selects a value t, which represents the number of machine that will keep their current schedule. After those t machines have been selected the operations on the rest of the machines are removed from the permutation. These operations are re-inserted in a specific order. The algorithm determines which operation will increase the makespan the least and adds that operation to the schedule. In the next iteration a new set of t machines will be selected to hold their schedule. [1]

4 Results

The tables will read the same as the one above. That is the percent columns represent the distance the search algorithm is off from the best know solution. In the Tabu Table the Difference column shows the difference between the algorithm's solution and the best known. However, in the GRASP table the Difference column represents the difference between the best solution of the primary GRASP algorithm and the Reactive GRASP implementation.

Problem	Best Known	Tabu Search	Percent	Difference
abz05	1,234	1,281	4	47
abz06	943	1,015	8	72
abz07	656	797	21	141
abz08	669	817	22	148
abz09	679	912	34	233
ft06	55	55	0	0
ft10	930	1,105	19	175
ft20	1,165	1,544	33	379
jpw43	32	32	0	0
la01	666	682	2	16
la02	655	698	7	43
la06	926	926	0	0
la07	890	916	3	26
la15	1,207	1,317	9	110
la21	1,046	1,250	2	204
la24	935	1,107	18	172
la25	977	1,162	19	185
la26	1,218	1,407	16	189
la27	1,235	1,534	24	299
la29	1,153	1,408	22	255
la31	1,784	1,891	6	107
la36	1,268	1,527	20	259
la38	1,196	1,457	22	261
la40	1,222	1,539	26	317
orb08	899	1,164	29	256
orb09	934	1,045	12	111

Table 2: Tabu Search Results

The Tabu Search performed the best out of all the implemented search algorithms. As shown in Table 2 this algorithm found the optimal solution on three of the twenty-six problems and was within 10 percent for five other problems. Even though that still leaves eighteen problems where the best Tabu Search solution is more than 10 percent away from the best known solution the other algorithms do not come close to these numbers. The Tabu Search also has the smallest range of solutions. The worst solution it found was within 27 percent of the best solution. Compare that to Local Search with either operator and it will be clear Tabu Search is much better. Local Search solutions were as far away as 44 percent from the best known.

Problem	Best Known	GRASP	Percent	Reactive GRASP	Percent	Difference
abz05	1,234	1,289	4	1,287	4	2
abz06	943	1,019	8	1,026	9	-7
abz07	656	809	23	810	23	-1
abz08	669	839	25	827	24	12
abz09	679	911	34	901	33	10
ft06	55	58	5	56	2	2
ft10	930	1,150	24	1,171	26	-21
ft20	1,165	1,554	33	1,588	36	-34
jpw43	32	32	0	32	0	0
la01	666	678	2	681	2	-3
la02	655	714	9	709	8	5
la06	926	926	0	926	0	0
la07	890	926	4	937	5	-11
la15	1,207	1,333	10	1,338	11	-5
la21	1,046	1,269	21	1,239	18	30
la24	935	1,124	20	1,104	18	20
la25	977	1,245	27	1,187	21	58
la26	1,218	1,450	19	1,471	21	-21
la27	1,235	1,672	35	1,601	30	71
la29	1,153	1,451	26	1,426	24	25
la31	1,784	1,953	9	1,930	8	23
la36	1,268	1,519	20	1,507	19	12
la38	1,196	1,516	27	1,510	26	6
la40	1,222	1,584	30	1,538	26	46
orb08	899	1,187	32	1,214	35	-27
orb09	934	1,105	18	1,112	19	-7

Table 3: Comparing GRASP Searches

Reactive GRASP did perform better than the primary GRASP on most of the problems. The Reactive GRASP uses all the α values at least once in each run, allowing it to search much more of the search space. For example in the primary GRASP algorithm when the random α is low, it will have almost the same solution every iteration since the RCL is small. However, the Reactive GRASP will go through each level of greediness at least once given it an advantage.

5 Conclusions

Tabu Search tended to perform better than other algorithms when it used random restarts. After running the algorithm many times it was clear that merely increasing the number of iterations was not improving the results greatly. However, Allowing the algorithm to start from a new location every twenty interactions changed the output. It created a much larger area for the algorithm to search. The Reactive GRASP algorithm is comparable to Tabu Search. On smaller problems the Tabu Search was at most 2 percent to the optimum. Although, on larger problems the Tabu Search performed up to 7 percent better than the Reactive GRASP.

The primary GRASP algorithm did almost as well as the Reactive GRASP. For most problems it was within 3 percent. Reactive GRASP did perform better by up 6 percent on some of the hardest problems. This is due to the range of α values it uses. Changing α ever iteration of the search gives it much more flexibility since it using so many different levels of greediness. In order to get these results from the primary GRASP algorithm, it would have to be run at least m times, most likely more.

6 Acknowledgments

I would like to thank all the members of the Artificial Intelligence group at Colorado State University, especially my mentor, Professor Adele Howe. Your knowledge and guidance through this project has been invaluable to my education. Thank you for your encouragement and immense kindness.

Also I would like to thank the Distributed Mentor Project for funding this research. Nancy Amato's and Jessica Hodgins' active roles in encouraging undergraduate women in computer science is honorable. Their hard work allowed me to have this great experience.

7 References

[1] Applegate, David and Cook, William. A Computational Study of th Job-Shop Scheduling Problem. ORSA Journal of Computing. Vol. 3 No. 2, Spring 1991. pg. 149-156.

[2] Binato, Hery, Loewenstern, and Resende. A GRASP for Job Shop Scheduling.

[3] Herzt, Taillard and de Werra. A Tutorial on Tabu Search.

[4] Pinedo. Scheduling: Theory, Algorithms, and Systems Second Ed. Publisher, 2002. pg. 156-185, 345-352.

[5] Resende and Ribeiro. Greedy Randomized Adaptive Search Procedures.

[6] Russell and Norvig. Artificial Intelligence: A Modern Approach Second Ed. Publisher, 2003 pg. 94-136.

[7] Shaw, Paul. Using Constraint Programming and Local Search Methods to Solve Vehicle Routing Problems.

[8] Watson, J.P. The Job-Shop Scheduling Problem. Fall 2003. Chapter 2 pg. 1-13