Scheduling as Heuristic Search with State Space Reduction

Ramiro Varela y Elena Soto

Centro de Inteligencia Artificial.
Universidad de Oviedo. Campus deViesques. E-33271 Gijón. Spain.
Tel. +34-8-5182032. FAX +34-8-5182125.
e-mail: ramiro@aic.uniovi.es
http:\\www.aic.uniovi.es

Abstract. In this paper we confront the Job Shop Scheduling problem by means of an A* algorithm for heuristic state space searching. This algorithm can guarantee optimal solutions, i.e. it is admissible, under certain conditions, but in this case it requires an amount of memory that grows linearly as the search progresses. We hence start by focusing on techniques that enable us to reduce the size of the search space while maintaining the ability of reaching optimal schedules. We then relax some of the conditions that guarantee optimality in order to achieve a further reduction in the number of states visited. We report results from an experimental study showing the extent to which this reduction is worth carrying out in practice.

1 Introduction

State space searching is a classic artificial intelligence technique suited to problems involving deterministic actions and complete information. It has a number of interesting properties, such as the ability to guarantee optimal solutions and the possibility of exploiting domain knowledge to guide the search. Unfortunately, even when a great amount of knowledge is available at a reasonable computational cost, the total cost of a search process is prohibitive, since the number of explored nodes grows linearly with the size of the search space, even for small problem instances. For this reason, a number of techniques are usually employed to reduce the effective search space with the subsequent loss of optimality.

In this paper we confront the Job Shop Scheduling (JSS) problem by means of an A* heuristic search algorithm [6,7]. Firstly, we use a technique that enables us to restrict the search space to the set of active schedules. This is a subset of feasible schedules to a given problem that contains at least one optimal solution. In order to do so, we exploit the strategy of the well-known G&T algorithm [4]. As we will see in the reported experiments, this technique combined with a classic heuristic can solve small problem instances to optimality. We then exploit two more methods aimed at further reducing the number of states expanded during the search. The first one consists in a reduction of the search space that limits the search to a subset of the active schedules. The second is a weighted heuristic method that assigns more

reliance to the heuristic estimation during the first stage of the search. It becomes clear that neither of the aforementioned methods maintains admissibility, i.e. the guarantee of reaching optimal solutions. However, the effect can be controlled in both cases by means of parameters. We report results from an experimental study in which we calculate the value of the parameters that produce the optimal solution at the lowest cost of the search procedure.

The rest of the paper is organized as follows. In Section 2, we formally describe the JSS problem. In Section 3, we present a version the G&T algorithm, the so-called hybrid G&T, and show how this algorithm can be adapted by means of a parameter to define a search space representing either the whole set or a subset of active schedules. In Section 4, we summarize the main characteristics of the A* algorithm, as well as the heuristic strategies that we used in the experiments. In Section 5, we report the results from our experimental study. Finally, the main conclusions are summarized in Section 6, where we also propose a number of ideas for further work.

2 The Job Shop Scheduling Problem

JSS requires scheduling a set of jobs $\{J_1,...,J_n\}$ on a set of physical resources or machines $\{R_1,...,R_q\}$. Each job Ji consists of a set of tasks or operations $\{t_{i1},...,t_{imi}\}$ to be sequentially scheduled. Each task has a single resource requirement and a fixed duration or processing time du_{il} and a start time st_{il} whose value must be determined. We assume that there is a release date and a due date between which all the tasks have to be performed.

Furthermore, the problem presents two non-unary constraints: *precedence constraints* and *capacity constraints*. Precedence constraints, defined by the sequential routings of the tasks within a job, translate into linear inequalities of the type: $st_{il} + du_{il} \leq st_{il+1}$ (i.e. st_{il} before st_{il+1}). Capacity constraints, which restrict the use of each resource to only one task at a time, translate into disjunctive constraints of the form: $st_{il} + du_{il} \leq st_{jk} \vee st_{jk} + du_{jk} \leq st_{il}$ (two tasks that use the same resource cannot overlap). The most widely used goal is to come up with a feasible schedule such that

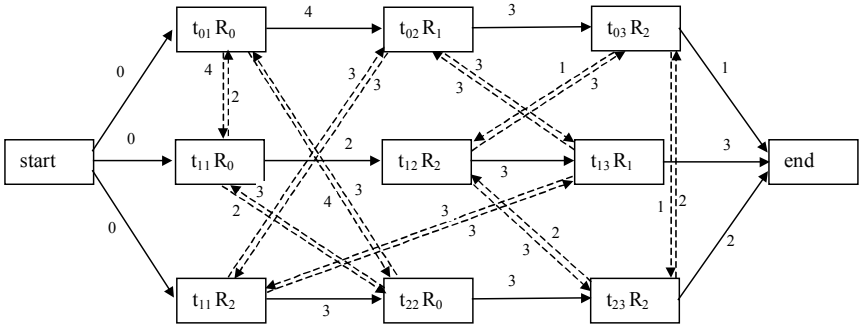

Fig. 1. A directed graph representation of a JSS problem instance with three jobs. The release date is 0 and the due date is 15. The resource requirement of each task is indicated within the boxes. Arcs are weighted with the processing time of the task at the outcoming node.

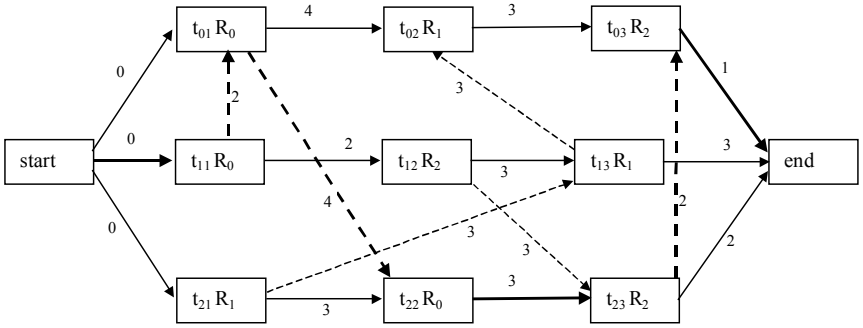

Fig. 2. A feasible schedule for the problem of Figure 1. The boldface arcs show the critical path whose length, i.e. the makespan, is 12. Hence it is actually a solution to the problem, because this value is less that 15, the due date.

the completion time of the whole set of tasks, i.e. the makespan, is minimized.

In the following, a problem instance will be represented by a directed graph $G = (V, A \cup E)$. Each node of the set V represents a task of the problem, with the exception of the dummy nodes *start* and *end*, which represent tasks with processing time 0. The set of arcs A represents the precedence constraints and the set of arcs E represents the capacity constraints. The set E is decomposed into subsets E_i with $E=\cup_{i=1..m}E_i$, such that there is one E_i for each resource R_i. The subset E_i includes an arc (v,w) for each pair of tasks requiring the resource R_i. Figure 1 depicts an example with three jobs $\{J_0,J_1,J_2\}$ and three physical resources $\{R_0,R_1,R_2\}$. Solid arcs represent the elements of the set A, whereas dotted arcs represent the elements of the set E. The arcs are weighted with the processing time of the task at the source node. The dummy task *start* is connected to the first task of each job; and the last operation of each job is connected to the node *end*.

A feasible schedule is represented by an acyclic subgraph Gs of G, $Gs=(V,A\cup H)$, where $H=\cup_{i=1..m}H_i$, H_i being a Hamiltonian selection of E_i. The makespan of the schedule is the cost of a critical path. A critical path is a longest path from node *start* to node *end*. When this value is less than or equal to the due date, the schedule is a solution to the problem. Therefore, finding a solution can be reduced to discovering compatible Hamiltonian selections, i.e. orderings for the tasks requiring the same resource or partial schedules, that translate into a solution graph Gs without cycles whose critical path does not exceed the due date. Figure 2 shows a graph representing a feasible solution to the problem of Figure 1.

2 The G&T Algorithm and the State Space for the JSS Problem

This is the well-known algorithm proposed by Giffler and Thomson in [4]. Here we present a variant called *hybrid G&T* that is based on a chromosome-decoding schema proposed by Bierwirth and Mattfeld in [2] within the framework of a Genetic Algorithm. This schema is in turn inspired by a proposal made by Storer, Wu and Vaccari in [11] for state space searching. In principle, the G&T is a greedy algorithm

that builds up a schedule for a given problem by scheduling one task at a time. In each iteration, a subset of tasks B is determined such that no matter how a task is selected from B to be scheduled next, the resulting schedule is *active*. Active schedules are a subset of feasible schedules in which no operation could be started earlier without delaying some other operation. It is easy to prove that the set of active schedules contains at least one optimal schedule. Thus, a search constrained to the whole set of active schedules is exhaustive, while keeping the size of the search space much smaller than a search over the whole set of feasible schedules.

Figure 3 shows the hybrid G&T algorithm. This is a variant of the genuine G&T in which a narrowing mechanism is introduced that enables a reduction of set B in each iteration. This is controlled by a reduction parameter $\delta \in [0,1]$; when $\delta=1$, sentence 6 has a null narrowing effect and hence we have the original G&T algorithm. In this case, it is possible to envisage a non-deterministic sequence of operation selections that gives rise to an optimal schedule. Otherwise, i.e. if $\delta<1$, the search is restricted to a subset of the active schedules; thus guaranteeing the existence of such a sequence is no longer possible. As pointed out in [2], the algorithm produces a *non-delay* schedule at the extreme $\delta=0$. Non-delay scheduling means that no resource is kept idle when it could start processing some operation.

Given its non-deterministic nature, the G&T algorithm can be particularized in many ways by fixing a selection criterion at step 7. For example, it has been widely used in the context of Genetic Algorithms as a decoding schema and is usually combined with some dispatching rule to design a greedy, non-optimal algorithm.

Algortithm G&T hybrid

1. *A = set containing the first task of each job;*

while $A \neq \varnothing$ **do**

2. *Determine the task* $\theta' \in A$ *with the shortest completion time if scheduled in the current state, that is* $t\theta' + du\theta' \leq t\theta + du\theta,\ \forall \theta \in A$;
3. *Let M' be the resource required by* θ'*, and B the subset of A whose tasks require M';*
4. *Delete from B every task that cannot start at a time lower than* $t\theta' + du\theta'$;
5. *Determine task* $\theta'' \in B$ *with the lowest possible start time, let* $t\theta''$ *be this time;*

/* *the least start time of every operation in B, if it is selected next, is a value of the interval* $[t\theta'', t\theta' + du\theta']$ */

6. *Reduce the set B such that*

$$B = \{ \theta \in B \,/\, t\theta < t\theta'' + \delta((t\theta' + du\theta') - t\theta'') \},\ \delta \in [0,1]\};$$

/* *now the interval is reduced to* $[t\theta'', t\theta'' + \delta((t\theta' + du\theta') - t\theta'']$ */

7. *Select* $\theta^* \in B$ *and schedule it at its lowest possible start time to build a partial schedule corresponding to the next state;*
8. *Delete* θ^* *from A and insert the next task of* θ^* *in this set if* θ^* *is not the last task of its job;*

endwhile;

Fig. 3. The hybrid G&T algorithm; $t\theta$ stands for the start time of operation θ and $du\theta$ for its processing time.

In this paper, we use the G&T algorithm to define a search space that can be explored by a conventional state space search algorithm such as backtracking or best first (BF). This idea was used for instance in [5] and consists in the following two steps, bearing in mind that the initial state s represents a null partial schedule $PS_{\varnothing}$:

1. At a state n representing a partial schedule PS_T for a set of tasks T, calculate a set B as done by the G&T algorithm in an analogous situation.
2. For each task $t \in B$, generate a successor n' of n with a partial schedule $PS_{T'}$, T' being $T \cup \{t\}$, by scheduling t at its lowest start time from the partial schedule PS_T.

Hence the branching factor of the state space is given by the average value of the cardinality of set B over the whole set of states. This branching factor is expected to be much lower than the one obtained with other approaches like, for example, those proposed in [9] and [1], which might be found to be impractical for an admissible search, even for a small problem instance. On the other hand, the cost $c(n,n')$ is given by the difference $C_{max}(PS_{T'}) - C_{max}(PS_T)$, where $C_{max}(PS_T)$ is the maximum completion time of a task in the partial schedule PS_T. It is easy to prove [10] that the state space generated in this way is a tree.

The state space proposed in [9] consists in selecting an unscheduled operation at a given state and then considering all the possible starting times compatible with the current partial schedule. For each one of these start times, a new partial schedule is built by scheduling the current task at this time. This strategy requires the establishment of a due date for each job so that the current set of possible start times could be restricted to a finite set. This set is initially determined from the release and due dates and the precedence constraints and is further reduced as long as the search continues.

On the other hand, the state space proposed in [1] consists in reductions of the constraint graph of the problem by means of various types of commitments that can be asserted and retracted; for example, fixing a start time of an operation or posting a precedence constraint between activities. In this case, the approach generalizes the JSS problem to other situations where, for example, one goal might be to minimize the number of resources used. In both cases, good experimental results are achieved using a backtracking search guided by means of smart, powerful heuristics. However, it becomes clear that both of the cited schemas are not suitable for an admissible search like A*, due to the fact that the branching factor is so high that it is impractical to store all the states required.

4 The A* Algorithm

The A* algorithm is a general BF heuristic search for graphs [6,7], though the algorithm is much simpler if the state space is a tree. It starts from an initial state s, a set of goal nodes and a transition operator SCS such that for each node n of the search space, $SCS(n)$ provides the set of successor states of n. Each transition from a node n to a successor n' has a positive cost $c(n,n')$. The algorithm searches for a solution path from the node s to one of the goal states. At any one time, there is a set of candidate

nodes to be expanded which are maintained in an ordered list *OPEN*; this list is initialized with the node *s*. Then in each iteration, the node to be expanded is always the one in *OPEN* with the lowest value of the evaluation function *f*, defined as $f(n)=g(n)+h(n)$; where $g(n)$ is the minimal cost known to date from the node *s* to the node *n*, (of course if the search space is a tree, the value of $g(n)$ does not change, otherwise this value has to be updated as long as the search progresses) and $h(n)$ is a heuristic positive estimation of the minimal distance from *n* to the nearest goal.

The A* algorithm has a number of interesting properties, most of which depend on the heuristic function *h*. First of all, the algorithm is complete. Moreover, if the heuristic function underestimates the actual minimal cost, $h^*(n)$, from *n* to the goals, i.e. $h(n) \leq h^*(n)$, for all nodes, the algorithm is admissible, i.e. the return of an optimal solution is guaranteed. The heuristic function $h(n)$ represents knowledge about a specific problem domain, therefore if complete knowledge were available at an assumable computational cost (at most polynomial on the problem size), i.e. $h(n)=h^*(n)$, the best solution could be found by expanding the lowest number of nodes possible. Unfortunately, this is rarely the case, and in practical cases we have to look for the best underestimation whose computational cost is assumable. This is because another interesting property of the algorithm is that if we have two admissible heuristics *h1* and *h2*, such that $h1(n)<h2(n)$, *h2* is said to be more informed than *h1* and it can be proved that in this case every node expanded by *h2* is also expanded by *h1*. It is then said that the algorithm using *h2* dominates the one using *h1*.

The most common technique for discovering admissible heuristics is problem relaxation [7]. This consists in relaxing some of the problem constraints so as to obtain a relaxed problem that can be solved to optimality, or at least a good underestimation can be made, in polynomial time. Then the solution cost of the relaxed problem is taken as the real problem cost estimation. In the case of the JSS problem, the constraints that can be relaxed are precedence and capacity constraints. Two problem relaxations are common: in the first, every capacity constraint is relaxed; whereas in the second, every precedence constraint is relaxed. It is easy to see that the cost of the optimal solution in the first case can be calculated in linear time. In the second case, however, the relaxed problem has a similar complexity to that of the original problem, although a reasonably good underestimation of its optimal cost can also be calculated in linear time. It is clear that the two aforementioned relaxations are quite strong, hence the relaxed problem is actually much simpler than the original one and the resulting heuristic is thus not too informed. Unfortunately, as far as we know, lower relaxed versions of the problem cannot be solved in an acceptable amount of time. Even though the heuristics obtained from the two relaxations mentioned above cannot be strictly compared, they are both expected to perform similarly in square problems, i.e. problems with an equal number of jobs and resources. However, the second heuristic will probably perform better for problem instances with much more jobs than resources, whereas in the opposite case, the first heuristic will probably do so.

In the experimental study described in the next section we consider the first of the aforementioned heuristics, which is calculated for a state *n* corresponding to a partial schedule PS_T of a subset of tasks *T*, as follows:

$$h_p(n) = MAX_{J \in JOBS} \left\{ C^J_{max}(PS_T) + \sum_{\theta \in US(J)} du_\theta \right\} - C_{max}(PS_T), \quad (1)$$

where *JOBS* is the set of jobs of the problem instance, $C^J_{max}(PS_T)$ is the completion time of the last scheduled task of job *J* at node *n*, $US(J,n)$ is the set of unscheduled tasks of job *J* at node *n* and du_θ is the processing time of task θ.

Although the heuristic h_p is clearly an underestimation of the optimal cost, the difference $h^*(n)-h_p(n)$ is not expected to be uniform for all nodes along the search process. Nevertheless, this difference is expected to be quite large at the beginning of the search, subsequently decreasing as the search progress and tends to be null at the end. This leads us to use the following dynamic weighted heuristic technique: a weighting factor *P(n)* is introduced into the *h* component of the evaluation function *f*, so that this function is $f(n)=g(n)+P(n)h(n)$. The value of *P(n)* depends on the depth of the node *n*. In this paper, we propose the following value:

$$P(n) = K - \left(K * \frac{depth(n)}{N * M} \right) + 1, \quad (2)$$

where $K \geq 0$ is the weighting parameter, *depth(n)* is the number of tasks scheduled at state *n*, *N* is the number of jobs and *M* is the number of resources. It is clear that for values of $K>0$, the aforementioned function *f* is not guaranteed to be admissible, since the component $P(n)h(n)$ might overstimate the value of $h^*(n)$. However, for some value of *K*, this component is expected to be a good aproximation of $h^*(n)$. Weighted heuristic search is widely used; for instance, a dynamic weighted method is proposed in [8] that guarantees that the cost of the solution does not worsen more that a factor ε with respect to the optimal. In [3], a static method is proposed that uses a constant weighting factor *W* for planning problems; this method guarantees a worsening factor not higher than *W*. And a dynamic method similar to ours is used in [5] for the JSS problem in conjuntion with a limited memory schema and a non-admissible heuristic obtained from a classic dispatching rule that overestimates the optimal cost.

5 Experimental Results

In this section we report results from an experimental study on a set of JSS problems instances. We used the prototype implementation proposed in [10], which is coded in C++ language and developed in Builder C++ 5.0 for Windows. The hardware platform was a Pentium III at 900 Mhz. and 125 Mbytes of RAM. The first problem of the test bed was problem FT06; this is a small problem obtained from the OR library (http://www.ms.ic.ac.uk/info.html). The remaining problem instances were generated by us, as the problem instances from the OR library and other conventional repositories are too large to be solved to optimality by our implementation in an acceptable amount of time.

Table 1 shows the results obtained with problem FT06, running the algorithm with the heuristics h_0 and h_p and considering values of the parameter δ ranging in the interval [0.0,1.0]. As expected, heuristic h_0 is unable to solve the problem. On the

Table 1. Experimental results from the problem instance FT06 with heuristics h_0 and h_p.

Heuristic	Reduction parameter δ	Number of generated nodes	Number of expanded nodes	Branching Factor	Cost of the reached solution	Run time (hh:mm:ss)
h_0	0.0	89,129	66,141	1.26	57	09:03:34
	0.1	89,129	66,141	1.26	57	09:03:34
	0.2	160,575	127,765	1.26	57	57:23:32
	0.3	-	-	-	-	>125:00:00
h_p	0.0	4,662	3,651	1.27	57	4
	0.1	4,662	3,651	1.27	57	4
	0.2	9,170	6,797	1.35	57	10
	0.3	**5,395**	**4,007**	**1.35**	**55**	**5**
	0.4	8,061	5,702	1.41	55	8
	0.5	11,340	7,589	1.49	55	10
	0.6	14,163	9,225	1.54	55	13
	0.7	21,841	13,571	1.61	55	19
	0.8	26,056	15,399	1.69	55	23
	0.9	28,127	16,134	1.74	55	25
	1.0	**28,127**	**16,134**	**1.74**	**55**	**25**

other hand, heuristic h_p can solve the problem to optimality, even with a space reduction given by δ=*0.3*. In this case, we can observe a significant reduction in the number of expanded and generated nodes, the branching factor and the running time, as long as the parameter δ decreases from *1.0* to *0.3*. For values of δ under *0.3*, the number of expanded nodes is higher in some cases, as the search space does not contain an optimal solution.

Figure 4 shows the evolution of the branching factor over the depth of the search space for 2 values of the parameter δ. As we can see, a value of δ=*0.3* produces a slightly lower branching factor over the first levels than δ=*1.0*, which translates into a lower overall number of both generated and expanded nodes. Here, the branching factor at depth *N* is calculated as the quotient of the number of states generated at depth *N+1* and *N* respectively.

Table 2 shows the reduction in the number of nodes generated and expanded for increasing values of *K*. Here we consider the limit values of δ that generate an exhaustive search space (one containing an optimal solution). As we can see, the number of generated and expanded nodes clearly decreases while the optimal solution is still reached, with the exception of value *K=0.3*, for which a suboptimal solution is reached.

Table 3 shows the results of an experimental study on a set of 10 problem instances of size 6×6 with random job sequences and processing times generated at random in the interval [5,95] from a uniform probability distribution. For each one of the problems, we report the number of nodes expanded for the limit values of δ and *K*. First for values δ=*1* and *K=0*, which are the only values that guarantee optimality. Then for every instance, we kept *K=0* and decreased the parameter δ down to the lowest value (δ*min*) that produces an optimal solution. After that, maintaining the value of δ*min*, the value of *K* is augmented to the largest value, *Kmax*(δ*min*), that

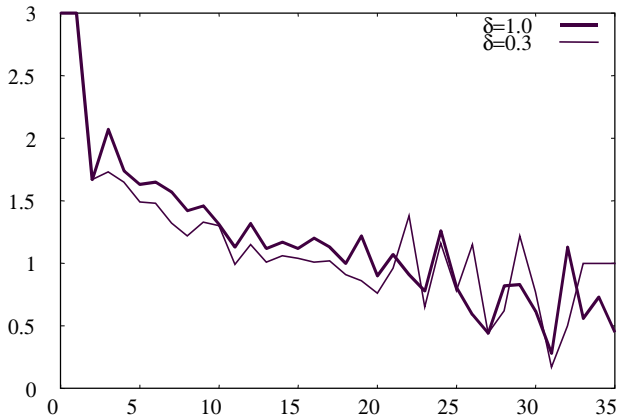

Fig. 4. Evolution of the branching factor over the depth of the search tree for two values of the parameter δ.

produced an optimal solution. And finally, keeping $\delta=1$, the value of K is also augmented to the largest value, $Kmax(\delta=1)$, that produced an optimal solution. As we can see, the variations of δ and K allowed without loss of optimality are not the same for all problems and the actual computational cost of obtaining the optimal schedule varies significantly from one problem to another. In fact, problem instance number 6 could not be solved to optimality. The best solution we were able to obtain was by running the algorithm with $\delta=0.9$ and $K=0.5$, after expanding 62328 states.

6 Conclusions

The main conclusion of the experiments reported above is that in real problems it is possible to reduce the size of the search space as well as to weight the heuristic

Table 2. Experimental results from the problem instance FT06 with heuristic h_p and different values of δ and K. In every case is the optimal solution reached, except with $K=0.3$. In this case, a solution with cost *58* is reached with either value *1.0* or *0.3* of δ.

K	$\delta = 0.3$		$\delta = 1.0$	
	Number of Generated Nodes	Number of Expanded Nodes	Number of Generated Nodes	Number of Expanded Nodes
0.00	**5,395**	**4,007**	**28,127**	**16,134**
0.05	1,715	1,279	9,397	5,472
0.10	1,092	806	5,257	3,013
0.15	812	598	2,545	1,466
0.20	313	237	599	333
0.25	**156**	**109**	**563**	**313**
0.30	210	165	574	339

Table 3. Experimental results from the set of 10 problem instances

Prob. Inst.	Number of Expanded Nodes				δmin	*Kmax*	
	$\delta=1, K=0$	$\delta min, K=0$	$\delta=1, Kmax$	$\delta min, Kmax$		$\delta=1$	δmin
0	14,018	3,373	14,018	3,373	0.4	0.0	0.0
1	1,843	717	1,843	717	0.4	0.0	0.0
2	60,276	32,621	56,953	31,674	0.7	0.1	0.1
3	3,235	280	457	102	0.0	0.2	0.2
4	2,535	543	139	64	0.2	0.3	0.5
5	1,337	219	175	48	0.2	0.2	0.2
6	-	-	-	-	-	-	-
7	11,627	118	11,627	118	0.8	0.0	0.0
8	11,120	481	1776	44	0.0	0.1	0.6
9	55,939	27,268	7,141	5,172	0.7	0.5	0.5

evaluation function to some extent up to a certain point given by the values *(δmin, Kmax(δmin))* in such a way that the optimal solution is obtained by expanding a number of states much lower than the number of states expanded with values *(1,0)*. Unfortunately, determining the limit *values (δmin, Kmax(δmin))* for a given problem instance is not trivial; it is a new search problem in the two-dimensional space *(δ,K)*. As future work, we plan to study a systematic way to calculate these values and experiment with other techniques that reduce the computational cost of the search procedure.

References

1. Beck, J. Ch., and Fox, M. S. Dynamic problem structure analysis as a basis for constraint-directed scheduling heuristics. Artificial Intelligence 117, 31-81 (2000).
2. Bierwirth, Ch., and Mattfeld D. C., Production Scheduling and Rescheduling with Genetic Algoritms. Evolutionary Computation 7 (1), 1-17 (1999).
3. Bonet, B., Geffner, H., Planning as Heuristic Search, Artificial Intelligence 129, 5-33, (2001).
4. Giffler, B. Thomson, G. L. Algorithms for Solving Production Scheduling Problems. Operations Reseach 8, 487-503 (1960).
5. Hatzikonstantis, L. and Besant, C. B., Job-Shop Scheduling Using Certain Heuristic Search Algorithms. Int. J. Adv. Manuf. Tecnol. 7, 251-261 (1992).
6. Nilsson, N., Principles of Artificial Intelligence, Tioga, Palo Alto, CA, 1980.
7. Pearl, J., Heuristics, Morgan Kauffman, San Francisco, CA, 1983.
8. Pohl, I., Practical and theoretical considerations in heuristic search algorithms, Machine Intelligence 8, Ed. E. W. Elcock and D. Michie, Ellis H. Ltd., Chich., Great Britain, 1977.
9. Sadeh, N., Fox, M.S., Variable and Value Ordering Heuristics for the Job Shop Scheduling Constraint Satisfaction Problem. Artificial Intelligence 86, 1-41 (1996).
10. Soto, Elena, Resolución de problemas de Satisfacción de Restriciones con el Algoritmo A*. Proyecto fin de carrera nro. 1012076. ETSII e II de Gijón. Universidad de Oviedo. (2002).
11. Storer, R., and Talbot, F. New search spaces for sequencing problems with application to job shop scheduling. Management Science 38, 1494-1509 (1992).