

Generating Effective Admissible Heuristics by Abstraction and Reconstitution¹

Armand Prieditis
 Department of Computer Science
 University of California
 Davis, CA 95616
 prieditis@cs.ucdavis.edu

Bhaskar Janakiraman
 Silicon Graphics Inc.
 Mountain View, CA 94039
 bhaskar@mti.sgi.com

Abstract

Admissible heuristics are worth discovering because they have desirable properties in various search algorithms. Unfortunately, effective ones—ones that are accurate and efficiently computable—are difficult for humans to discover. One source of admissible heuristics is from abstractions of a problem: the length of a shortest path solution to an abstracted problem is an admissible heuristic for the original problem because the abstraction has certain details removed. However, often too many details have to be abstracted to yield an efficiently computable heuristic, resulting in inaccurate heuristics. This paper describes a method to *reconstitute* the abstracted details back into the solution to the abstracted problem, thereby boosting accuracy while maintaining admissibility. Our empirical results of applying this paradigm to project scheduling suggest that reconstitution can make a good admissible heuristic even better.

1 Introduction

Admissible (lower-bound) heuristics are worth discovering because they have desirable properties in various search algorithms. For example, they guarantee shortest path solutions in the A^* [24] and IDA^* [19] algorithms and less expensively produced, but boundedly longer solutions in the *dynamic weighting* [26] and A_c^* [25] algorithms. Moreover, multiples of them can reduce an exponential average time complexity to a polynomial one with A^* [4]. Unfortunately, effective (accurate and efficiently computable) admissible heuristics are difficult for people to discover.

Several researchers have shown that admissible heuristics can be generated from abstractions (transformations that drop certain details) of a problem [12, 10, 25, 16, 23, 27, 28, 29]. As Figure 1 shows, the length of a shortest path solution in the abstracted problem is the admissible heuristic. For such heuristics to be effective, the abstracted problem that generates them should be efficiently solvable and yet close to the original problem [32, 23, 13, 29]. This technique typically yields efficiently computable, but

inaccurate heuristics because efficiently solvable abstracted problems often ignore precisely those details that are central to solving the original problem.

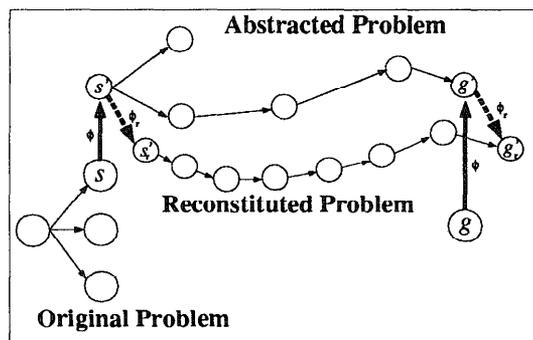


Figure 1 The Length of a Shortest Path Solution to the Abstracted Problem = Admissible Heuristic for the Original Problem; Reconstitution Increases that Length, Thereby Boosting Accuracy of the Heuristic

This paper describes a method called *reconstitution* that adds back such ignored details to the abstracted problem's solution, thereby boosting accuracy heuristic while maintaining admissibility. As Figure 1 shows, the length of a shortest path solution in the abstracted problem is increased with reconstitution to yield a more accurate admissible heuristic. The ultimate goal of this research is to develop an automatic reconstitution system to shift some of the burden of discovery from humans to machines.

2 Project Scheduling

As a vehicle for exploring reconstitution, we investigated *project scheduling* problems because they are of practical importance and are difficult to solve without effective heuristics. A project scheduling problem consists of a finite set of jobs, each with fixed integer duration, requiring one or more resources such as personnel or equipment, and each subject to a set of precedence relations, which specify allowable job orderings, and a set of mutual exclusion constraints, which specify jobs that cannot overlap. No job can be interrupted once started. The objective is to minimize

¹ This work is supported by the National Science Foundation grant number IRI-9109796.

project duration. Since this objective arises in nearly every large construction project—from software to hardware to buildings—efficient algorithms that obtain that objective are desirable.

Integer linear programming methods have been used to solve project scheduling problems for years [1, 3, 21, 11]. However, these methods are computationally expensive, unreliable, and applicable only to problems of small size. The underlying reason for the computational expense and limited problem size is that such project scheduling problems are NP-hard [9]. As a result, scheduling problems are typically solved by branch-and-bound algorithms with lower-bound duration estimates (admissible heuristics) to improve efficiency [31, 7, 2].

The only published attempt at discovering admissible heuristics a scheduling domain yielded poor heuristics when abstraction alone was applied [22, 28]. Moreover, the particular scheduling problem (uniprocessor scheduling) to which it was applied did not allow concurrency, which is the *essence* of scheduling.

3 Key Definitions

As shown in Figure 2, a scheduling problem can be represented as graph with jobs as vertices, precedences as single-arrowed edges, and mutual exclusions as double-arrowed edges. For example, the figure shows that job *I* must be completed before job *J* can start and that jobs *J* and *K* cannot overlap. The single number above each job represents the job's duration. For example, job *J* takes 10 units of time to complete. The letter to the left of each job represents the resource that the job requires; one job's use of a resource cannot overlap with another job's use of that same resource. For example, jobs *I* and *E*, which both require resource *s*, cannot overlap with each other.

A *precedence graph* is a directed acyclic graph consisting only of the precedence relations and no resource constraints. An *early schedule graph* is derived from the precedence graph, where each job is scheduled as early as possible. The numbers within the square brackets near each job in the figure represent the *earliest start time* and the *earliest completion time* of each job. The *critical path* is the longest path in the early schedule graph; it shows the earliest time by which all jobs can be completed.

No job on the critical path can be delayed, although other jobs on the same early schedule can be delayed as long as they do not increase the critical path length. For example, if job *J*, which is on the critical path, starts later than 33 units of time, the entire project will be delayed. These jobs may have to be delayed in order to satisfy mutual exclusion constraints. The total completion time of an early schedule is therefore equal to the critical path length, which in our case is 43. An *optimal schedule* is an early schedule which takes the least total time among all possible schedules.

Given only precedence constraints, finding an early schedule reduces to a topological sort of the precedence graph, which can be done in linear time of the number of precedence constraints [15]. Finding the critical path in an early schedule takes linear time of the number of jobs. Therefore, if all other constraints such as mutual exclusion constraints and resource constraints can be recast as precedence constraints, the problem is easily solvable. For example, the mutual exclusion constraint between jobs *J* and *K* can be recast in two ways: either *J* is completed before *K* or vice versa. Similarly, for resource constraints each pair of jobs sharing the same resource can be recast as a mutual exclusion constraint between the two jobs. Each mutual exclusion constraint can then be recast as one of two precedence constraints as previously described. Henceforth, we assume that all resource constraints have been recast as mutual exclusion constraints.

4 Branch and Bound Project Scheduling

The idea of recasting mutual exclusion and resource constraints as precedence constraints suggests the following simple combinatorial algorithm. Explore all recastings, one at a time, that do not create a cycle and find early schedules for all of these recastings; the early schedule with the minimum critical path length is the optimal one. Unfortunately, this brute-force algorithm is combinatorially explosive: n mutual exclusion constraints results in 2^n possible recastings, which is clearly too large a space to explore exhaustively for large n . One way to reduce this combinatorial explosion is to use a branch-and-bound algorithm with lower-bound estimates to prune certain recastings earlier. If the current duration + the lower-bound estimate exceeds a user-supplied upper-bound, then that schedule can be pruned.

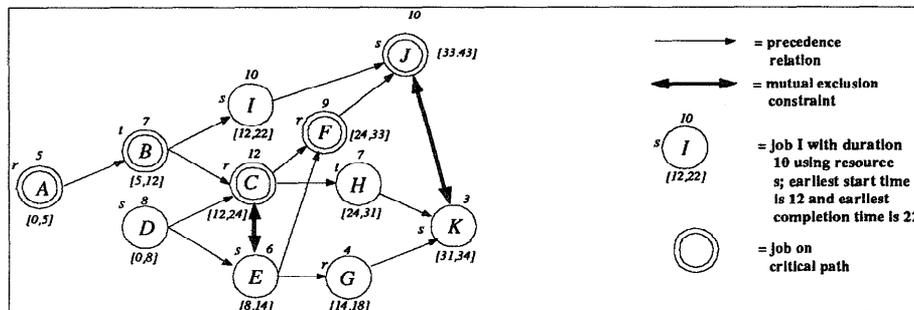


Figure 2 A Project Scheduling Problem

1. Calculate the critical path (CP).
2. Traverse the CP backwards—from late to early jobs. Assume jobs encountered are j_k, j_i, \dots
3. As each job j_k on CP is encountered, look for unsatisfied mutual exclusion constraints between j_k , and some job j_r , where j_r is not on CP. Notice that traversing the graph backwards as in step 2 is more efficient than forwards because we have to reschedule only one job.
4. If in the given schedule, execution of j_k overlaps with j_r , then push j_r to a later time so that there is no overlap.
5. Push other jobs later in time if necessary. This is done by listing all jobs j_p such that j_r must come before j_p according to the precedence relation, and rescheduling j_p so that j_r completes before j_p . Repeat this step until all the precedence relations are satisfied.
6. If the length of CP in the new schedule is greater than the original CP, then terminate the algorithm and return the new bound as original CP length + amount of overlap between j_k and j_r .
7. Else, repeat till a mutual exclusion constraint is found which increases CP length.
8. If no such constraint is found, return the CP length as the new bound.

Figure 3 An Algorithm to Compute the RCP Heuristic

The critical path estimate of an early schedule, which is efficiently computable, is clearly a lower-bound since any early schedule that satisfies *part* of the constraints is a lower bound on the completion time for any optimal schedule satisfying *all* constraints. Moreover, any additional constraint will not result in a decrease in the critical path length. Notice that the critical path (CP) heuristic results from an abstraction of the original problem: all mutual exclusion and resource constraints are ignored.

Although the CP heuristic is admissible and easily computable and has proved to be valuable in evaluating overall project performance and identifying bottlenecks, it can be far from the actual project duration. In the worst case, it can underestimate the actual project duration by a factor of n , where n is the total number of jobs to be scheduled. This case arises when the only possible schedule is a serial schedule. For example, if a scheduling problem has no precedence constraints and has mutual exclusion constraints between every pair of jobs, then the only possible schedule will be a serial one. For this case, the CP heuristic will return length of the longest job, which underestimates the optimal duration by a factor of n . Also, since the critical path estimate ignores the resource constraints, certain sequencing decisions may be required in the actual schedule that increase the project duration well beyond the critical path estimate.

5 Reconstitution-based Heuristics in Project Scheduling

What we would like is an admissible heuristic that is as easily computable as the critical path estimate, but that takes into account the resource and mutual exclusion constraints, which the critical path estimate ignores. We would like to reconstitute these ignored constraints back into the critical path somehow. The RCP (Reconstituted Critical Path) heuristic described below does exactly that.

The basic idea behind the RCP heuristic is to extend the critical path by analyzing all unsatisfied mutual exclusion constraints between jobs in critical path and jobs not in critical path. When possible, all jobs with such unsatisfied constraints are rescheduled at a later time while still

preserving critical path length. If that is not possible, then the critical path length is increased by a time overlap underestimate between the jobs of each type. For example, consider the project scheduling problem in Figure 2, which has a critical path of J, F, C, B, A . First, we examine job J and check for any mutual exclusion constraints involving it. The only such constraint is the one with job K . Next, we check if J overlaps with K , which in fact it does. The object now is to try to delay job K beyond the completion time of job J , which is at 43 time units. Delaying job K will necessarily increase the length of the critical path by 1 time unit. If the rest of the jobs were ignored, the RCP heuristic would return 44, which is the length of critical path (43) plus the overlap of the earliest start time of job J and the earliest completion time of K ($34 - 33 = 1$). The general algorithm is shown in Figure 3 and a pictorial definition of overlap is shown in Figure 4.

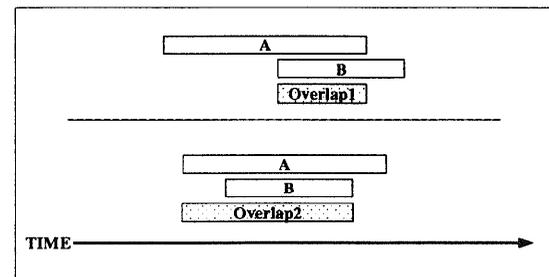


Figure 4 Overlap is the Minimum of Both Overlaps; Jobs A and B are Mutually Exclusive

To see that the RCP heuristic is admissible, consider a job j_l on the critical path which has a mutual exclusion constraint with job j_m . In the final schedule, either j_l will be scheduled before j_m or vice versa. Note that neither of the two jobs can be scheduled any earlier since the schedule is already an early schedule. If job j_m cannot be scheduled after j_l without increasing the critical path length in the current schedule by pushing jobs ahead which depend on j_m , then neither can it be scheduled after j_l in the final schedule. The reason is that precedence constraints are

always added and never removed at each iteration of the search algorithm and adding more precedence constraints cannot invert an existing scheduling order. If j_l is scheduled after j_m , then the critical path length will be increased by *at least* the minimum of the overlap between the earliest start time of j_l and the earliest completion time of j_m or the earliest start time of j_m and the earliest completion time of j_l . The reason is because job j_l is on the critical path: starting it later affects the entire project duration.

Although the RCP heuristic takes slightly longer to compute than the CP heuristic, it prunes more of the space than the CP heuristic. As we will see in the next section, the extra time taken in computing the heuristic is more than compensated by the time saved from pruning the search space. If the current critical path length is optimal, then computation of the RCP heuristic takes longer than that of the CP heuristic, since the algorithm has to examine all jobs on the critical path. The worst case complexity of computing the RCP heuristic is $O(n^2)$ for n jobs, since at most $O(n)$ jobs will be on the critical path and $O(n)$ work will be required to process a mutual exclusion constraint involving a job on the critical path. An analysis of the *average* computational complexity is, however, difficult since the heuristic depends on specific mutual exclusion constraints. The degree of complexity can be controlled by reconstituting fewer mutual exclusion constraints if desired.

The complexity of the RCP heuristic can be further reduced by computing it incrementally. That is, the RCP on a successor state (one with a new precedence constraint added) can be computed more efficiently by reusing a portion of the RCP on the original state. Since new precedence constraints are added and never removed at each iteration of the search algorithm, the critical path up to the point in the graph where the new precedence constraint is added remains the same and the critical path need only be recomputed from that point on.

6 Empirical Results

To get some idea of the effectiveness of the RCP and CP heuristics, we implemented the IDA* algorithm [19], which is a standard branch-and-bound algorithm in which to evaluate admissible heuristics, in Quintus Prolog on a Sun Sparstation 1+ and ran it on a set of random solvable (i.e. no cycles) problem instances with various numbers of jobs, mutual exclusion constraints, and precedence constraints. The algorithm works as follows. All partial schedules whose duration exceeds a certain threshold are pruned. Initially, the threshold is set to the value of the admissible heuristic on the initial state. If no solution is found within that threshold, then the algorithm repeats. On the next iteration, the threshold is set to the minimum of duration plus heuristic estimate over all the previously generated partial schedules whose duration exceeds the threshold. One important property of IDA* is that it guarantees minimal duration solutions with admissible heuristics [19].

A state consists of three items:

1. A precedence graph which includes original precedence constraints and a set of precedence constraints

originating from mutual exclusion constraints which have so far been recast as one of two precedence constraints.

2. An early schedule satisfying the precedence constraints.
3. A set of unsatisfied mutual exclusion constraints.

The goal state is characterized by an empty mutual exclusion constraint set. A state transition is a recasting of a mutual exclusion constraint into one of two precedence constraints followed by the generation of a new early schedule. Search proceeds from an initial schedule satisfying only the original precedence constraints.

We ran two sets of experiments, each with a fixed the number of jobs and precedence constraints and a variable number of mutual exclusion constraints since problem complexity grows as the number of mutual exclusion constraints increases: one with 30 jobs with 112 precedence constraints and the other with 40 jobs with 128 precedence constraints. For the first set, we varied the number of mutual exclusion constraints between 0 and 25; for the second, between 10 and 40. We chose these problems because they were the largest ones we could generate that still could be solved in a reasonable amount of time on our machine.

Table 1 summarizes the results of running IDA* on these two problem sets. For each problem set, the table lists the number of mutual exclusion constraints, the number of states expanded, the CPU time, and the amount of run-time memory used. As the table shows, for problems with few mutual exclusion constraints, the number of states expanded in both cases remain the same and CP consistently takes less time than RCP, since RCP does more work each time. However, for all problems where RCP resulted in a saving in terms of states expanded, RCP always takes less CPU time. RCP also uses slightly more run-time memory in all examples, but always within a factor of 4 when compared to CP. The breakdown between 15 and 20 mutual exclusions in the first data set and 20 and 30 in the second data sets may be sudden because a particular "hard problem" region threshold is cross in those ranges. We have yet to understand what aspect of a scheduling problem determines actual running time, but we suspect that there is some threshold of the average number of mutual exclusions per job that separates the hard from the easy problems. In summary, RCP works better than CP in all cases where the critical path length is not optimal, which is typically the case in real-world (non-artificial) problems, where it is highly probably that constraints other than precedence constraints play a major role in dictating the total project duration. Therefore, RCP will result in better performance in most real-world cases.

We are currently running more extensive experiments. One major problem in this domain is that there is a lack of good benchmark hard problems and that "easy" problems might skew randomly generated data sets unless the easy ones are filtered out. We hope to produce a set of such good benchmark problems.

7 General Reconstitution

This paper has described one instance of reconstitution: use

an efficient algorithm to generate a optimal (shortest duration) solution to an abstract problem and then calculate how adding back certain constraints increases the duration of this solution. We have identified one other type of reconstitution, this one involving abstracted problems that are decomposable into independent subproblems. This type of reconstitution involves calculating how the sum of shortest path lengths for each of the independent subproblems increases when abstracted dependencies are added back. For example, the Manhattan distance heuristic for sliding block puzzles is derivable by ignoring (abstracting) the location of the blank—a shortest path solution to the abstracted problem is the Manhattan Distance. Since the rectilinear tile distance to each tile's final destination can be independently computed, the abstracted problem that generates Manhattan Distance can be decomposed into a set of independently solvable subproblems, one for each tile. If the blank is added back to each subproblem then dependencies such as linear conflicts (i.e. two tiles that must pass through each other to their goal destination) can be efficiently detected and the solution path length can be boosted by two for such conflicts. We are currently implementing a general purpose reconstitution algorithm for decomposable abstracted problems.

8 Related Work

The idea that abstraction-derived heuristics can sometimes be made more effective by taking into account certain details ignored by the abstracted problem was first expressed by Hansson, Mayer, and Yung [14]. In particular, they hand-derived a new effective admissible sliding block puzzle heuristic (the LC heuristic) by taking into account those linear tile conflicts (same row or column) ignored by the Manhattan Distance heuristic. We have extended this idea to a problem involving time (project scheduling) rather than solution path length.

Instead of using an abstract solution as a heuristic measure, others (such as [17]) use it as a skeleton for producing a solution; the remaining details are then filled in by re-

finement. One drawback of this method is that the abstract solution often cannot be refined—backtracking between the original and abstract level must continue until a refinable solution is found. As a result researchers have tried to find specific types of abstractions that can always be refined, or refined with little backtracking [18]. However, guaranteed refinable abstractions are difficult to find.

In general, heuristics are efficient approximations of lookahead searches [20]. A heuristic approximates the search by either ignoring certain paths or adding additional paths to reduce search complexity. If a full-width lookahead search is used a heuristic for solving the original problem in an algorithm such as A^* , then the complexity of A^* will be worse than that of A^* without any heuristics. What the heuristic does is trade off efficiency for accuracy in approximating this full-width search, thus making it worthwhile for algorithms such as A^* to use the heuristic.

9 Conclusions and Future Work

This paper has described an instance of a general three step problem-solving paradigm: abstract, solve, reconstitute. Certain details of the original problem are removed by abstraction. Next, the abstracted problem is efficiently solved. Finally, the abstracted details are reconstituted back into this solution. This reconstituted solution is then used as a guide for solving the original problem. Our results of applying this paradigm to project scheduling, where reconstitution was used to generate a novel effective admissible heuristic (RCP), suggest that reconstitution can make good admissible heuristics even better.

This paradigm as applied to project scheduling has several shortcomings. First, complex project scheduling problems often involve resource constraints with fixed limits for each job, typically specifying the *number* of fixed resource units that cannot be exceeded, rather than the absolute resource constraints as in our model; it is not clear to us how to recast such resource constraints as mutual exclusion constraints. However, Davis and Heidorn [5] show a branch-and-bound solution to the problem. They describe

Jobs	Precedences	Mutual Exclusions	CP Heuristic			RCP Heuristic		
			States Expanded	CPU Seconds	Bytes	States Expanded	CPU Seconds	Bytes
30	112	0	0	.25	30820	0	.25	30808
		10	14	5.43	41024	14	8.52	58108
		15	19	5.98	43212	19	9.18	70256
		20	6237	1644.17	45796	49	30.58	109584
		25	6242	1651.53	47188	54	30.67	116068
40	128	10	12	5.15	52928	12	5.63	72228
		20	24	9.73	56988	24	18.52	101936
		30	1084	718.20	71024	431	494.33	194220
		40	1096	727.83	65104	521	521.80	224112

Table 1 Comparative Performance Analysis of the CP and RCP Heuristics with IDA*

a preprocessor algorithm that expands a job with duration k into a sequence of k unit duration jobs each successively linked with a “must immediately precede” precedence relation. After this expansion, a standard branch-and-bound project scheduling algorithm can be run. Unfortunately, such expansion can result in enormous project networks in projects with long duration jobs.

A second shortcoming is that not all scheduling constraints can be recast as precedence constraints. For example, a constraint that a particular job must start only after a certain time cannot be recast as a precedence constraint. Effective admissible heuristics that reflect such general constraints would be an important contribution to scheduling.

How does the amount reconstitution quantitatively relate to the accuracy of the resulting heuristics? How much reconstitution is enough? This paper shows one data point at which reconstitution pays off. Since reconstitution is the inverse of abstraction, results that quantitatively link abstractness to the accuracy of the resulting heuristics should be applicable [6, 30].

Finally, although this paper has described a method for generating better admissible heuristics from existing ones, the process of discovering heuristics such as the RCP heuristic is far from automatic. We are currently extending this method to job-shop scheduling problems of the sort described in [8]. In a job-shop problem, n jobs are to be scheduled on m machines with varying durations per job per machine. We hope to develop a set of general principles that practitioners in the scheduling field can follow to derive effective heuristics and eventually to automate the discovery process.

References

- [1] M. L. Balinski. Integer programming: Methods, uses, computation. *Management Science*, November 1965.
- [2] C. E. Bell and K. Park. Solving resource-constrained project scheduling problems by A* search. *Naval Research Logistics*, 37:61–84, 1990.
- [3] J. D. Brand, W. L. Meyer, and L. R. Schaffer. The resource scheduling problem in construction. Technical Report 5, Dept. of Civil Engineering, University of Illinois, Urbana, 1964.
- [4] S. Chenoweth and H. Davis. High-performance A* search with rapidly growing heuristics. In *Proceedings IJCAI-12*, Sydney, Australia, August 1991. International Joint Conferences on Artificial Intelligence.
- [5] E. W. Davis and G. E. Heidorn. An algorithm for optimal project scheduling under multiple resource constraints. *Management Science*, 17(12), August 1971.
- [6] R. Davis and A. Prieditis. The expected length of a shortest path. *Information Processing Letters*, (To appear), 1993.
- [7] M. Dinçbas, H. Simonis, and P. V. Hentenryck. Solving large combinatorial problems in logic programming. *Journal of Logic Programming*, 8(1 and 2):75–93, 1990.
- [8] M. S. Fox. *Constraint-Directed Search: A Case Study of Job-Shop Scheduling*. Pitman, 1984.
- [9] M. Garey and D. Johnson. *Computers and Intractability*. W. H. Freeman, San Francisco, 1979.
- [10] J. Gaschnig. A problem-similarity approach to devising heuristics. In *Proceedings IJCAI-6*, pages 301–307, Tokyo, Japan, 1979. International Joint Conferences on Artificial Intelligence.
- [11] D. Graham and H. Nuttle. A comparison of heuristics for a school bus scheduling problem. *Transportation*, 20(2):175–182, 1986.
- [12] G. Guida and M. Somalvico. A method for computing heuristics in problem solving. *Information Sciences*, 19:251–259, 1979.
- [13] O. Hansson, A. Mayer, and M. Valtorta. A new result on the complexity of heuristic estimates for the A* algorithm. *Artificial Intelligence*, 55(1), 1992.
- [14] O. Hansson, A. Mayer, and M. Yung. Criticizing solutions to relaxed models yields powerful admissible heuristics, 1992. To appear in *Information Sciences*.
- [15] E. Horowitz and S. Sahni. *Fundamentals of Data Structures*. Computer Science Press, Inc, Rockville, Maryland, 1978.
- [16] D. Kibler. Natural generation of heuristics by transforming the problem representation. Technical Report TR-85-20, Computer Science Department, UC-Irvine, 1985.
- [17] C. Knoblock. Learning abstraction hierarchies for problem solving. In *Proceedings AAAI-90*, Boston, MA, 1990. American Association for Artificial Intelligence.
- [18] C. Knoblock. Search reduction in hierarchical problem-solving. In *Proceedings AAAI-91*, Anaheim, CA, 1991. American Association for Artificial Intelligence.
- [19] R. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(2):97–109, 1985.
- [20] R. Korf. Real-time heuristic search: New results. In *Proceedings AAAI-88*, St. Paul, MN, 1988. American Association for Artificial Intelligence.
- [21] C. L. Moodie and D. E. Mandeville. Project resource balancing by assembly line balancing techniques. *Journal of Industrial Engineering*, July 1966.
- [22] J. Mostow, T. Ellman, and A. Prieditis. A unified transformational model for discovering heuristics by idealizing intractable problems. In *AAAI90 Workshop on Automatic Generation of Approximations and Abstractions*, pages 290–301, July 1990.
- [23] J. Mostow and A. Prieditis. Discovering admissible heuristics by abstracting and optimizing. In *Proceedings IJCAI-11*, Detroit, MI, August 1989. International Joint Conferences on Artificial Intelligence.
- [24] N. J. Nilsson. *Principles of Artificial Intelligence*. Morgan Kaufmann, Palo Alto, CA, 1980.
- [25] J. Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem-Solving*. Addison-Wesley, Reading, MA, 1984.
- [26] I. Pohl. The avoidance of (relative) catastrophe, heuristic competence, genuine dynamic weighting and computational issues in heuristic problem solving. In *Proceedings IJCAI-3*, pages 20–23, Stanford, CA, August 1973. International Joint Conferences on Artificial Intelligence.
- [27] A. Prieditis. *Discovering Effective Admissible Heuristics by Abstraction and Speedup: A Transformational Approach*. PhD thesis, Rutgers University, 1990.
- [28] A. Prieditis. Machine discovery of effective admissible heuristics. In *Proceedings IJCAI-12*, Sydney, Australia, August 1991. International Joint Conferences on Artificial Intelligence.
- [29] A. Prieditis. Machine discovery of effective admissible heuristics. *Machine Learning*, 1993. To appear.
- [30] A. Prieditis and R. Davis. Quantitatively relating accuracy to abstractness of abstraction-derived heuristics. *Artificial Intelligence*, (Submitted), 1993.
- [31] F. J. Radermacher. Scheduling of project networks. *Journal of Operations Research*, 4(1):227–252, 1985.
- [32] M. Valtorta. A result on the computational complexity of heuristic estimates for the A* algorithm. *Information Sciences*, 34:47–59, 1984.