Parallel Search Algorithms for Robot Motion Planning *

Daniel J. Challou **Maria Gini** **Vipin Kumar**
Department of Computer Science
University of Minnesota
Minneapolis, MN 55455

January 20, 1993

Abstract
In this paper we show that parallel search techniques derived from their sequential counterparts can enable the solution of instances of the robot motion planning problem that are computationally infeasible on sequential machines. We present a simple parallel version of a robot motion planning algorithm based on "quasi best first" search with randomized escape from local minima and random backtracking and discuss its performance on two problem instances, one of which was computationally infeasible on a single processor of an nCUBE2[1] multicomputer. We then discuss the limitations of parallel robot motion planning systems, and suggest a course for future work.

1 Introduction

Among the many skills autonomous robots require to support their activities is the ability to plan the paths they must take while conducting those activities. Motion planning enables an object to move safely through its environment while achieving specific goals. Many algorithms exist for finding a path from a robot's initial configuration to a desired goal configuration when the locations of the obstacles in the workspace are known apriori [Latombe, 1991]. In order to describe how such robot motion planning problems relate to search, a brief review of terms is necessary.

The *workspace* of a robot is the world that it is capable of moving through; this workspace usually includes a variety of objects or obstacles. A *configuration* of a robot is the specification of the position and orientation of that robot with respect to a fixed reference frame. The *configuration space* (C-Space) [Lozano-Perez, 1983] of a robot is the set of all configurations that can be assigned to it. The reason we concern ourselves with the space of possible configurations of a robot is that in such a space the robot becomes a point. The *free C-Space* of a robot is the set of configurations belonging to the robot's C-Space in which the robot is not in a state of collision with itself or any other object in its workspace. The dimensionality of this C-Space is the number of parameters required to fully specify a configuration of the robot. For example, a fixed base robot arm with six degrees of freedom (i.e. joints) would operate in a C-Space of six dimensions.

In C-Space then, planning the path of motion for a robot becomes the problem of finding a path for a point through free C-Space given a start point and a non-empty set of goal points. There may be more than one goal configuration possible because for some robots more than one configuration will place specific points on the manipulator (e.g., the grippers) at the desired position in the workspace.

Most robot motion planning algorithms decompose C-Space into discrete components called *cells*. The motion planning problem then becomes one of computing a decomposition of C-Space and searching through sequences of contiguous cells to find a path through free C-Space (i.e. a sequence of robot configurations that involves no collisions with obstacles). Search is usually performed using state-space search algorithms such as A* or its variants.

*This work was supported in part by Contract Number DAAL03-89-C-0038 between the Army Research Office and the University of Minnesota for the Army High Performance Computing Research Center. Additional support was furnished by NSF/CDA-9022509, and IST/SDIO grant No. 28408-MA-SDI.

[1]nCUBE2 is a registered trademark of the nCUBE corporation

As more degrees of freedom are added to the robot, most methods for solving the problem become computationally infeasible on sequential machines. For example, consider a robot arm with k joints (degrees of freedom), where each degree of freedom is quantized into n discrete levels. Such an arm has a C-Space consisting of n^k unique configurations. Let us assume that our planner quantizes each joint so that it can be in up to 90 unique positions, so $n = 90$. Thus an arm with four joints ($k = 4$) has over sixty five million configurations, while an arm with six joints ($k = 6$) has over five hundred billion configurations. Even though free C-Space may be much smaller than the total C-Space, most algorithms require exhaustive search of C-Space in the worst case.

Thus, for the example above, even if it is possible to compute the C-Space the amount of memory required to store it in RAM is prohibitive. Moreover, the amount of computation required to find a solution to tough problem instances renders any approach to solving the problem on sequential machines computationally infeasible as well. Finally, since motion planning is PSPACE-hard [Reif, 1979] there is strong theoretical evidence that the best possible algorithms for solving this problem have time complexity that increases exponentially with the size of the input.

Given the considerations outlined above, it is clear that robot motion planning algorithms are computationally infeasible on sequential computers for many problem instances involving robots with more than three or four degrees of freedom. Hence, it is important to investigate the applicability of parallel search algorithms to existing sequential robot motion planning methods in order to make them computationally feasible for a wider range of problems. Furthermore, the application of such techniques to robot motion planning schemes that perform adequately on existing problem instances may increase their performance significantly as well.

2 Previous Work

Research in area of motion planning can be traced back to the late sixties, but most of the work has been carried out more recently. Over the last few years the theoretical and practical understanding of the issues has increased rapidly, and a variety of solutions have been proposed. Latombe [1991] provides an extensive survey.

Recently, Lozano-Perez [1991] developed a parallel algorithm which computes the discretized C-space for the first three links of a six degree of freedom manipulator. The path for the gripper portion of the manipulator is found by computing its free C-space in parallel at each arm configuration considered by the sequential search algorithm. Although this method works well, it is limited to relatively coarse C-space discretizations (it has a maximum discretization level of 64) because of the lack of memory available in which to store the precomputed C-space.

Barraquand and Latombe [1991] utilize discrete representations of the robot, the robot's workspace, and its C-Space. Space is represented with multiscale pyramids of bitmap arrays. Artificial potential fields are used as the heuristic to guide the search of C-space. They describe two algorithms. The first algorithm is resolution complete, but becomes computationally infeasible when the dimension of the configuration space exceeds four - (less with C-space discretization levels of 128 or more). The second algorithm is probabilistically complete[2], but, in general, runs much faster than the complete approach.

Other than the parallel scheme developed by Lozano-Perez described above, we are aware of no parallel method capable of solving instances of the robot motion planning problem involving higher dimensional C-space. Fortunately a great deal of work has been done in developing parallel search algorithms capable of solving similar problems [Kumar and Rao, 1987, Kumar *et al.*, 1988]. Many of the algorithms developed have yielded linear speedup with increasing problem and processor size on various problems[Arvindam *et al.*, 1991, Ananth and Kumar, 1991]. If parallel robot motion planning methods that make use of such parallel search schemes can be developed, they may be able to deliver such performance as well. This is due to the following observation.

Amdahl's law states that if s is the serial fraction of an algorithm then, no matter how many processors are used, speedup is bounded by $1/s$ [Amdahl, 1967]. Thus, if an algorithm spends 98 percent of its time computing a certain function (say C-Space), and only that component can be computed in parallel, then the maximum speedup which can be obtained by the parallel algorithm is 50 because it must still spend 2 percent of its time in its serial component. Hence, parallelizing robot motion planning methods that rely almost entirely on a totally parallelizable search process may yield better speedup than parallelizing those approaches with even a small serial component.

[2]An algorithm is *probabilistically complete* when the probability of finding a path when one exists converges to 1 as the search time increases without bound.

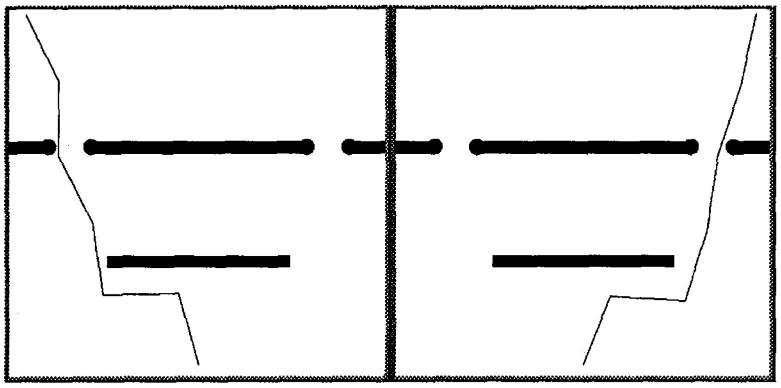

Figure 1: Start and Goal Configurations for Six Degree of Freedom Robot operating in a 256 x 256 cell workspace. Each C-space axis has 256 possible discrete positions

We discuss our initial results involving the implementation of such methods in the next section.

3 A Parallel Motion Planning Algorithm for MIMD multicomputers

In this section we describe how to parallelize the probabilistically complete method proposed by Barraquand and Latombe [1991]. We give a brief outline of the method below in order to clarify what is necessary to make it run on an multiple instruction multiple data (MIMD) multicomputer such as the nCUBE2.

Step I: Compute the heuristics used to guide the search:

1. Pick a "Control" point (or points) on the robot.
2. Pick the desired goal location in the workspace for the point(s) designated in step 1.
3. BROADCAST DESIRED GOAL LOCATION OF THE CONTROL POINT(S) TO ALL PROCESSORS.
4. For each "Control Point": Starting from the desired goal location, label each point in the workspace that is not an obstacle with the L1 (city block distance) from the goal location. For example, in a 2-D discrete workspace, the goal location gets the value 0, its four neighbors get the value 1, etc.

Step II: Search using heuristics to evaluate which new configuration to expand during iteration in the almost best first search phase.

```
Trials = 0
Path = Start Configuration

Repeat
      Path Trials = 0
      Temp_path = End of Path

      Repeat
          Quasi best first search until a local minimum is reached.
          IF TERMINATION MESSAGE RECEIVED, EXIT.
          Brownian Motion to escape local minimum.
          IF TERMINATION MESSAGE RECEIVED, EXIT.
          if Path Trials > Threshold Randomly backtrack to a previous point in Temp_path.
      Until Path Trials > Max_better_path_trials or Temp_path with new minimum found.

      If new minimum found, append Temp_path to Path.
Until Solution Found or Trials > Trial_Limit
IF SOLUTION FOUND, BROADCAST TERMINATION MESSAGE TO ALL OTHER PROCESSORS.
```

No. Processors	1	2	4	8	16	32	64	128	256	512	1024
Avg. Sol. Time	51430	66360	32864	11973	5598	6875	2596	900	380	301	178
Max. Run Time	173040*	183893*	92227	35008	26718	22847	10068	1919	576	634	258
Min. Run Time	5386	655	4006	588	356	1326	194	145	191	53	11 0
Max - Min. Time	167654	183238	88221	34420	26362	21521	9874	1774	385	581	148
Speedup ($\frac{W}{T_P}$)	-	-	4	10.98	23.48	19.12	50.63	140.0 6	345.94	436.73	738.52
Efficiency ($\frac{S}{P}$)	-	-	1.00	1.37	1.47	0.60	0.79	1.09	1.35	0.85	0.72

Figure 2: Table summarizes the data for ten runs of the Quasi Best First/ Random Planner on up to 1024 processors on an nCUBE2 multicomputer for the problem instance pictured in figure 1. The table shows the following information: Average time to find solution; Maximum run time (solution or no solution); Minimum run time to solution; Difference between the maximum and minimum run times per number of processors; Speedup and Efficiency averaged over runs that ended in a solution. Note that * indicates no solution was found on that run. All times are in seconds.

The search is "quasi best first" because all possible successors are not generated and evaluated deterministically. To save space, the successors are randomly generated and evaluated and the best of the randomly generated successors are kept. If enough successors are generated in each iteration of the quasi best first phase, then the method approximates best first search.

The capitalized statements in **Step I** and **Step II** of the algorithm outlined above highlight the additions we made to the sequential algorithm in order to enable the method to run on an MIMD multicomputer. Each processor runs the same basic program. The only interprocessor communication done is a broadcast of the desired goal location(s) of the Control point(s) in the workspace to all processors in **Step I**, and checks for a message indicating that another processor has found a solution in **Step II**.

Figure 1 shows the start and goal configurations for one of our test cases involving a six degree of freedom planar robot with one control point operating in a 256 x 256 cell workspace. Each joint has up to 256 discrete positions. Figure 2 shows the results for ten runs on up to 1024 processors.

3.1 Discussion of Results

At first one might be surprised that such a straight forward parallel algorithm fares as well as it does on such a difficult problem instance, reducing the average computation time from 51430 seconds (over 14 hours) on one processor to an average of 178 seconds (about 3 minutes) on 1024 processors. Furthermore, from the average times calculated in figure 2, it is apparent that the algorithm does not require a large number of processors to make significant reductions in the time required to solve this problem instance - just 64 processors are required to solve the problem in an average time of about 43 minutes.

In figure 2, the speedup is not calculated for the results up to and including four processors for the following reason. The planner failed to find solutions to the problem for four of the runs on one processor and three of the runs on two processors. If we had let the planner run long enough, it would have arrived at a solution since it is probabilistically complete. However it might have taken a great deal more time than allowed by the cutoff bound of about two days that we set (Trial_Limit = 800 in Step II of the algorithm outline on the previous page). Such cases would make the speedup appear a great deal better than does assuming linear speedup on four processors. For our speedup calculation then, we used four times the average time taken by four processors as the time on which speedup is based. Thus, all the speedup figures are conservative and would be much higher if the average time for one processor was used for computing speedup.

It is interesting to note that speedup oscillates between slightly sublinear and superlinear until it starts to fall off at about 512 processors. Moreover, the time variance required to solve the problem decreases as the number of processors used to solve the problem increases. For example, on sixteen processors, the maximum and minimum time to solve the problem varies by 26362 seconds (or over 7.3 hours), while on 256 processors and up the maximum time difference is 385 seconds (a little less than 6.5 minutes). Additionally, before we increased the cutoff bound, the planner was unable to formulate a solution to the problem on sixteen processors in the time it was allocated (24601 seconds or about 6.5 hours) in two of the runs we attempted. In order to clarify why these fluctuations occur, it may be helpful to first discuss what happens to parallel search algorithms operating on fixed size problems.

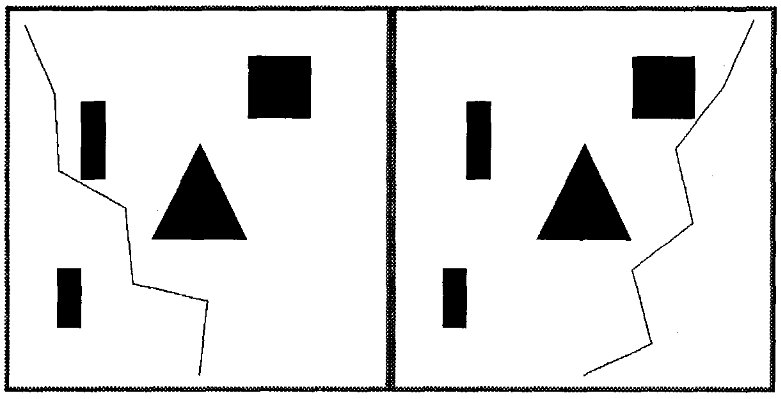

Figure 3: Start and Goal Configurations for Six Degree of Freedom Robot operating in a 256 x 256 cell workspace. Each C-space axis has 256 possible discrete positions

Let the problem size W be the time taken by the optimal (or the best known) sequential algorithm to solve the given problem on a single processor. We assume W is proportional to the size of the space searched. The execution time on P processors is defined as T_P. We define speedup S as the ratio $\frac{W}{T_P}$ and efficiency as $\frac{S}{P}$. In general, for a fixed problem size W, increasing the number of processors P causes a decrease in efficiency because parallel processing overhead will increase while the sum of time spent by all processors in meaningful computation remains the same. Since our parallel search scheme does virtually no communication during the search phase, the fall off in speedup and efficiency is due to redundant work - different processors exploring the same search states.

In this example then, the speedup and average time taken to solve the problem decreases and levels off as we increase the number of processors trying to solve the problem because we hit a point where the number of processors required to insure that one processor will find a solution in the minimum amount of time possible for the algorithm is near optimal or optimal. This is because the probability that the random component of the algorithm will ensure that different processors are exploring different parts of the search space decreases as we add more processors. When we reach that point, then adding more processors to the problem will just result in more processors doing redundant work (in the average case).

The reason for fluctuations in time taken to solve the problem from run to run on the same number of processors is as follows. On some runs the random component of the planner running on the processor solving the problem first helped it jump out of local minima more effectively than it did on other runs. Hence, on some runs the planner finds a solution very quickly using just a few processors, and on other runs, particularly with few processors, it does not find a solution at all.

Furthermore, according to our results, the probability that no solution will be found when one exists decreases as we increase the number of processors. Again, as we increase the number of processors running the planner, there is a better chance of good search space partitioning. Better search space partitioning ensures a more complete search of the space. Thus, if a solution exists, more processors are likely to find it.

Finally, in all experiments we have completed to date, our parallel algorithm delivers similar performance. For example in the problem instance depicted in figure 3, the planner managed to find a solution on each run within the designated time limit. However, even though this problem instance is not as difficult in terms of solution time as the problem depicted in figure 1, as the table in figure 4 shows, it exhibits similar behavior. Both the level off in speedup and fluctuations from run to run occur, they just occur with fewer processors.

3.1.1 Explaining Superlinear (and Sublinear) Speedup

From the superlinear speedup exhibited by our results it may appear that in some cases a better sequential algorithm can be obtained by emulating the parallel algorithm on a single processor. Such an algorithm will have an average runtime of $T_P * P$ on a single processor (where T_P is the parallel runtime on P processors defined above). In this case, the parallel algorithm will still exhibit at least linear speedups. In practice however, due

No. Processors	1	2	4	8	16	32	64	128	256
Avg. Sol. Time	6957	5447	2399	728	240	69	55	44	47
Max. Run Time	14818	12886	12886	1095	652	211	84	74	61
Min. Run Time	398	119	68	85	55	29	36	29	36
Max - Min. Time	14420	12767	12818	1010	597	182	48	45	25
Speedup ($\frac{W}{T_P}$)	-	1.28	2.90	9.56	28.99	100.83	126.49	158.11	144.94
Efficiency ($\frac{S}{P}$)	-	0.64	0.73	1.20	1.81	3.13	1.98	1.24	0.57

Figure 4: Table summarizes the data for ten runs of the Quasi Best First/ Random Planner on up to 256 processors on an nCUBE2 multicomputer for the problem instance pictured in figure 3. The table shows the following information: Average time to find solution; Maximum run time to solution; Minimum run time to solution; Difference between the maximum and minimum run times per number of processors; Average Speedup, and Average Efficiency. All times are in seconds.

to emulation overhead and limited hardware resources, the parallel algorithm will still deliver better than linear speedups.

Furthermore, as is the case with time fluctuations discussed above, some of the superlinear and sublinear speedup exhibited by our results is due to the random nature of the algorithm. Increasing the number of successful runs included in the average will eliminate the sublinear and superlinear speedup observed.

Some of the superlinear speedup observed may be due to the nature of the problem space and search itself. Recent work by [Rao and Kumar, 1992] indicates that depth first search with simple backtracking delivers at least linear speedup on the average when searching a tree in which the distribution of solutions is non-uniform. Since, for this problem instance, the heuristic that guides the search algorithm we have implemented is misleading at times, there may be cases in which the search behaves a great deal like depth first search with simple backtracking. If this is the case, then, since the distribution of solutions is non-uniform for our problem instance, some of the superlinear speedup observed in our results may be due to the effects described by Rao and Kumar [1992].

3.1.2 Is the Parallel Algorithm Really Necessary?

One might argue that given a short enough time bound, running the serial algorithm or emulating the parallel algorithm on a single processor multiple times will yield a solution with shortest path length in the shortest time. There are, however, problems with such an argument. To begin with, one must devise a method of predicting the minimal time required to devise a solution to a given instance of the problem. In some problem instances this will be relatively trivial, but for many others it may be extremely difficult.

One possible way around developing such a method is to emulate the parallel algorithm on a sequential machine and incrementally increase the time bound until the minimum time bound necessary to compute a solution is reached. As mentioned previously, such an algorithm will have an average runtime of $T_P * P$ on a single processor. For the example in figure 1, the fastest solution time we obtained with our parallel formulation was 53 seconds on 512 processors. This implies that the average runtime of the best sequential algorithm to find the same solution would be almost eight hours, if we guessed the time bound correctly on the first try. There will be cases in which the sequential system fares much better *and* cases in which it fares much worse. In any case, running the parallel algorithm on a parallel machine is clearly advantageous if solutions are required in more realistic time frames.

3.1.3 Summary

Assuming our results can be generalized, the performance delivered by this approach is fairly impressive. Only 64 processors are required to virtually guarantee a solution in a reasonable amount of time in both problem instances - significantly less time on average than a single processor requires to solve the problem (if the single processor can arrive at a solution before it hits its cutoff bound). If less variance in time to solution and more certainty that the solution the planner delivers is correct is required by the user, he or she need only to use 256 processors.

4 Future work

The randomized scheme we have implemented does have its drawbacks. In many cases the paths the planner found were clearly sub-optimal in terms of length and their ability to be executed by any real robot. As Latombe et. al. have pointed out, in many cases post processing can help optimizing such paths [Barraquand and Latombe, 1991]. However, such processing can be expensive in terms of computation and may not always yield a meaningfully improved path.

On the other hand, parallelizing search-based methods that keep track of the C-space they have visited may yield better results. Such approaches limit the amount of redundant work performed because they generate each configuration in C-space at most once. As we discussed earlier, existing implementations of such methods run into difficulty because they cannot store the C-space they require on a single processor. However, consider the new generation of massively parallel machines such as a 512 processor CM-5 with 32 megabytes of memory per processor. After allowing space for the operating system, program, and communication buffer, there should be at least 8 billion bytes which can be utilized by existing restricted and/or distributed memory schemes.

Relatively recent restricted memory schemes such as MA*[Chakrabarti *et al.*, 1989], MREC[Sen and Bagchi, 1989], and PRA*[Evett *et al.*, 1990], as well as distributed memory schemes such as A* with probabilistic state distribution [Manzini and Somalvico, 1990] appear promising. They seem to be the best methods available for MIMD multicomputers with hypercube and treelike interconnection networks because they enable the use of much more memory in which to store the C-space then afforded by other marking search algorithms.

5 Conclusion

In summary, we have devised and implemented a parallel robot motion planning algorithm based on quasi best first search with randomized escape from local minima and randomized backtracking. The method delivers excellent speedup on difficult problem instances, including one problem which, for all practical purposes, is computationally infeasible on a single processor.

6 Acknowledgements

We would like to sincerely acknowledge Jean Claude Latombe and his group at Stanford University for providing us access to implementations of the Random Path Planner, David Strip and Robert Benner at Sandia National Laboratories for providing us access to the nCUBE2, and Grama Y. Ananth and George Karypis for their many helpful suggestions and patience.

References

[Amdahl, 1967] G.M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS Conference Proceedings*, pages 483–485, April 1967.

[Ananth and Kumar, 1991] G.Y. Ananth and Vipin Kumar. Experimental evaluation of load balancing techniques for the hypercube. In *Proceedings of Parallel Computing 91*, London, September 1991.

[Arvindam *et al.*, 1991] S. Arvindam, Vipin Kumar, V. Nageshwara Rao, and Vineet Singh. Automatic test pattern generation on multiprocessors. *Parallel Computing*, 17(12):1323–1342, 1991.

[Barraquand and Latombe, 1991] J. Barraquand and J. C. Latombe. Robot motion planning: A distributed representation approach. *International Journal of Robotics Research*, 10(6):628–649, 1991.

[Chakrabarti *et al.*, 1989] P.P. Chakrabarti, S. Ghose, A. Acharya, and S.C. de Sarkar. Heuristic search in restricted memory. *Artificial Intelligence*, 41:197–221, 1989.

[Evett *et al.*, 1990] M. Evett, J. Hendler, A. Mahanti, and D. Nau. PRA*: A memory limited heuristic search procedure for the Connection machine. In *Proceedings of the Third Symposium on the Frontiers of Massively Parallel Computation (Frontiers '90)*, pages 145–149, 1990.

[Kumar and Rao, 1987] V. Kumar and V. Nageshwara Rao. Parallel depth-first search, part II, analysis. *International Journal of Parallel Programming*, 16(6):501–519, 1987.

[Kumar *et al.*, 1988] V. Kumar, K. Ramesh, and V. Nageshwara Rao. Parallel best-first search of state-space graphs: A summary of results. In *Proceedings of the 1988 National Conference on Artificial Intelligence*, pages 122–127, 1988.

[Latombe, 1991] J. C. Latombe. *Robot Motion Planning.* Kluwer Academic Publishing, Norwell, MA, 1991.

[Lozano-Perez, 1983] T. Lozano-Perez. Spatial planning: A configuration space approach. *IEEE Transactions on Computers*, C-32(2):108–120, 1983.

[Lozano-Perez, 1991] T. Lozano-Perez. Parallel robot motion planning. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 1000–1007, 1991.

[Manzini and Somalvico, 1990] G. Manzini and M. Somalvico. Probabilistic performance analysis of heuristic search using parallel hash tables. In *Proceedings of the International Symposium on Artificial Intelligence and Mathematics*, Ft. Lauderdale, FL, January 1990.

[Rao and Kumar, 1992] V. Nageshwara Rao and Vipin Kumar. On the efficicency of parallel backtracking. *IEEE Transactions on Parallel and Distributed Systems*, (to appear), 1992. Available as a technical report TR 90-55, Computer Science Department, Un iversity of Minnesota.

[Reif, 1979] J. Reif. Complexity of the mover's problem and generalizations. In *Proceedings of the 20th IEEE Symposium on Foundations of Computer Science*, pages 421–427, 1979.

[Sen and Bagchi, 1989] A.K. Sen and A. Bagchi. Fast recursive formulations for best-first search that allow controlled use of memory. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence (IJCAI-89)*, pages 297–302, 1989.