CMU-CS-84-147

A Quantitative Study of Search Methods and the Effect of Constraint Satisfaction

Hans Berliner
Gordon Goetsch

Computer Science Department
Carnegie-Mellon University
Pittsburgh, Pa. 15213

July, 1984

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory Under Contract F33615-81-K-1539.

Abstract

While certain guidelines for selecting search methods have emerged over the years, and while the success of certain methods has been well documented, there have been few, if any, studies of a variety of search methods as the difficulty of a particular problem changes. This study of a solitaire puzzle that can be incarnated in a variety of problem sizes and hence difficulties, attempts to fill this vacuum. We select four search paradigms: A*, Best-first with a simple evaluation function (BF1), Best-first with a complex evaluation function (BF2), and Depth-First with branch and bound and iterative deepening (DF), to represent the best of a variety of searching methods for non-adversary problems. All methods except BF2 use the same knowledge; i.e. a measure of the minimum number of moves that it would take to win the current situation. DF applies this measure only at maximum depth, while the other two use it for selecting which node to expand next. Each of these methods are tested with and without a constraint satisfaction procedure.

As expected, the most informed search (BF2) does better than the less informed as the problems get progressively more difficult. One important and apparently general result is that constraint satisfaction provides the greatest gain when coupled with the most informed algorithm (BF2). We found it surprising that BF1, which uses the same evaluation function as A* but with a different coefficient, far outperformed A* in terms of work required at about a 5% reduction in the quality of the (otherwise) optimal solution. We conjecture that Best-first searches get their power from an *adventurousness coefficient* which we define in the text. Adventurousness can be thought of as a primitive form of planning.

Introduction

Any state of a domain is either solvable or not solvable. We define the set of **totally solvable** domains to be those in which all states of the domain are either solved or solvable. In such domains any operator applied to any state preserves the solvability of the new state. Typically, for each operator there exists a reverse operator that can re-establish the previous state. This type of problem is represented by puzzles such as the Tower of Hanoi, and valid instances of the 8 and 15 puzzles and Rubik's cube. An alternate condition is that the permissible operations do not allow transformation to an unsolvable state. This type is represented by problems such as the travelling salesman problem in a plane (where there are arcs between any pair of cities and it is not permitted to traverse an arc more than once).

Set against the class of totally solvable problems is the class of **partially solvable problems** in which not every state of the domain can be solved. Here the initial state may not be solvable or if solvable, a set of operations could exist which lead to a non-solvable state. For totally solvable problems the only thing of interest is the speed of the solution process and the quality of the solution. For partially solvable problems, each instance may have to be classified as solvable or unsolvable. It is interesting to note that the general bin-packing problem could be in either class depending on whether a solution into a pre-defined area is required (partially solvable), or just a best solution is desired.

Both totally solvable and partially solvable problems can use heuristic knowledge in order to speed up the search for a solution. Typically, both the preference order for applying operators and the evaluation of new states can be affected by heuristic knowledge. However, totally solvable problems have no need for the process of identifying subtrees in which no solution can exist because, *ipso facto*, such sub-trees do not exist.

Constraint satisfaction is the term used for the set of algorithms that can be applied to identify with absolute certainty that a subtree cannot contain a solution. Probably the best known constraint satisfaction algorithm is the alpha-beta procedure [2]. We wanted to study the role of constraint satisfaction on a difficult, partially solvable problem as the difficulty of the problem increased.

We also wanted to study the effect of increasing the difficulty of a problem on the effectiveness of standard search algorithms. Four search algorithms were selected as being the most useful exemplars of their class for solving non-adversary problems. These are described in a later section. The issues to be investigated were: the degree of degradation for each type of algorithm as the solution length increased, and the interaction of constraint satisfaction with each particular search

method. We also wanted to examine the effect of hash table size on the effectiveness of each method. Hash tables are used in the various search methods to:

1. Represent the tree
2. Identify nodes that have already been examined elsewhere in the tree
3. Find cycles in a branch (no progress)

The various interactions and degradations with increasing difficulty form the substance of this paper.

The Domain

The domain chosen for this study was Superpuzz, an extremely difficult solitaire puzzle developed by the first author. The rules of Superpuzz are as follows:

Superpuzz is played with 24 cards, 6 in each of 4 suits. Individual cards of a suit are numbered 0 to 5. To set up a problem deal out the cards in a raster of 6 wide by 4 deep. Then remove all the "0" denomination cards, leaving "holes". Legal moves consist of moving a card into a hole, thus creating a hole at its former location. The card that is moved into a hole must be of the same suit as the card to the left of the hole, and be one higher in denomination. Thus, if the card to the left of a hole has suit S and denomination D, the only card that can be moved to the hole is the one with suit S and denomination D + 1. The card is moved directly from its location to the new location.

No card can be moved to the right of a last card in a suit, nor to the right of a hole. If a hole is on the left edge, any 1 may be moved there, regardless of whether the 1 is already on the left edge in another row (1's can be shuffled up and down). In Figure 1A no card can be moved to Y5, D5 can be moved from W5 to W4, and any ace can be moved to X0. The game is won when all cards have been aligned suit by suit in the first 5 places of each row. There is no requirement to have particular suits in particular rows. The games is lost when there are no longer any legal moves. The above are the rules for 4x6 Superpuzz; it is also possible to play harder versions adjusting the rules for 4x7 and 4x8 formats.

```
      0   1   2   3   4   5
W     S3  H2  D2  D4  --  D5
X     --  C4  H3  C5  C1  --
Y     D3  S1  D1  S4  S5  --
Z     C2  C3  H4  H1  S2  H5
```

(A)

```
      0   1   2   3   4   5
W     D1  D2  D3  D4  D5  --
X     H1  H2  H3  H4  H5  --
Y     C1  C2  C3  C4  C5  --
Z     S1  S2  S3  S4  S5  --
```

(B)

Figure 1: An initial and terminal configuration

An initial position is shown in Figure 1A, and a terminal position in Figure 1B. A path from the initial position to the terminal one can be found at the end of this article just before the references, for the benefit of those that may wish to try to find a solution by themselves. Superpuzz can exist in a variety of widths; the simplest interesting version being the 4x6 configuration presented above. There are an average of 4 legal moves at any point in the play, and solutions may range in depth from about 22 for simple 4x6 examples to 60 or more for complicated 4x8 examples. There are approximately 10^{21} states in 4x6 Superpuzz, 10^{26} for 4x7, and 10^{32} for 4x8. This compares to approximately 10^{20} states for checkers and backgammon, and 10^{43} states for chess. Standard puzzles such as the 8-puzzle and 15-puzzle only have 10^{5} and 10^{13} states respectively. For many instances of Superpuzz very few solutions exist out of 10^{10} or more potential solution paths. Because of the combinatorial nature of the game, even moderately good searching programs will outperform well practiced humans. This makes for an interesting domain, since it is no longer simply a matter of a programmer imparting heuristic knowledge to a program in order to get it to play better.

Certain insights for playing Superpuzz well may be useful to the reader for understanding what follows. It is possible to win about 83% of the 4x6 puzzles, 78% of the 4x7 variety, and 75% of the 4x8's. Typically good human players win on the order of 70%, 55% and 35% respectively. This is because of the combinatorial nature of the problem which frequently requires non-intuitive solutions that humans have a hard time finding.

Each card denomination has a destination column. Typically, it is advantageous to have a card located to the right of its destination column; the best location being exactly one column to the right of the destination column. To see why this is so, consider that all cards are so placed. Then it is trivial to win the position by first moving all the aces to the left edge, then all 2's to their final destinations, etc. Cards that are located to the left of their destination column are badly placed, particularly if they are high denomination cards. Such cards are typically difficult to move back, and if they are in sequences it become even more difficult.

A major part of the skill in playing Superpuzz comes from managing ace moving. Once a hole opens on the left edge, it is possible to move an ace there, either from some non left-edge position or possibly from the left edge. The real skill in playing Superpuzz comes from doing this well. Well planned moves of aces along the left edge will allow the moving of sequences of cards in an advantageous manner, while meaningless shuffling of aces would cause a program to waste time and prolong the solution or the discovery that there is no solution.

The Search Paradigms

There are two obvious choices for search paradigms:

1. A best-first search with a highly informed evaluation function that would encourage the development of good "positional" formations that could hopefully be transformed into wins. This was named BF2.

2. A depth-first search using the branch and bound technique. A depth-first search must have a maximum depth else it may continue downward for a long time. For these puzzles, if one wishes to select an absolute limit, it could mean a depth of 15 to 20 greater than the depth at which the solution actually occurred. To avoid such problems a natural method is to use iterative deepening [4], which approaches a solution depth by iterating depth-first searches to ever greater depths. Iterative deepening has recently been proved to dominate simple depth-first search when the depth of the solution is not known [1]. The above paradigm was named DF. The most effective constant of iteration for this domain was found empirically to be 2. Thus the first iteration was to the depth of the number of misplaced cards in a starting configuration, and future iterations were each to depths two greater than the previous. The bound for this search is the number of misplaced cards in the current configuration. This means that if N cards are misplaced and the search is at depth D, then it is impossible to reach a solution at less than depth D + N. This results in cut-offs of branches where insufficient progress is being made.

In both these paradigms and in the others that will be added, a hash table that contains generated nodes plays a key role. In the best-first searches, it becomes the representation of the tree, since each node and its parent must be maintained. The hash encoding also detects identical states, so that identical subtrees in different parts of the tree will only be searched once. If two identical nodes occur in the same branch, a cycle has occurred. It is important to detect these if progress is to be made. In a depth-first search, the hash table only performs the two latter functions.

As the result of giving this problem as a programming task in our graduate level AI course, two other meaningful candidate searches emerged:

3. The A* search [3]. At first sight this would appear to be destined to do worse than a well-informed best-first search, because the measure of effort remaining to reach a solution must be relatively simple; i. e. the number of misplaced cards, which is not an excellent measure of the goodness of a state. However, it turned out experimentally that even though the A* search is not very smart, it does well by its thoroughness. In certain situations, it looks as if progress is being made, but it really isn't. In such cases, best-first searches keep trying their favorite sub-tree, while A* will spread out at the root to try any branches whose cost functions indicate that they have the best chance at a minimal solution. Further, it is cheap to compute the number of misplaced cards, which is an advantage.

4. Even better than A*, turned out to be a best-first search that uses the A* evalution function. This algorithm (BF1) was tried by several students with superior results. Here, the evaluation function, instead of calculating a lower bound on the number of moves

required for a solution, attempts an accurate estimate of this. To do this, the cost of misplaced cards is multiplied by a constant since it is expected that rarely will it be the case that a misplaced card goes directly to its final destination in one move. The most successful constant for this domain was 1.8, so the cost function for BF1 is H = D + 1.8N, where D = depth, and N = number of misplaced cards.

These were the four search paradigms tried in our experiment. Two evaluation functions were required. These are 1) A misplaced card counter, and 2) A position goodness function. These are described below.

The misplaced card counter uses the following two rules to determine if a card is in place:

1. If the four aces are in the four left edge slots the position is said to be frozen; e.g. no more ace moves are possible and the final destination of all cards is known. In that case, a card is misplaced if it is not at its final destination.

2. If the four aces are not yet all in place, a card is misplaced if it is not in its destination column, or if it is but does not constitute a part of the majority of properly placed cards in that row. The latter condition prevents cards of various suits all being counted as being in place in a single row.

The position goodness function evolved over a number of months. It gives points as follows:

1. One point for every non-misplaced card.

2. For every card that is to the left of its destination column, subtract the number of points that is equal to the denomination of the card; e. g. if the three of hearts is in one of the first two columns, a penalty of 3 points is assessed.

3. If a card is exactly one column behind its destination column a reward equal to the (denomination of the card)/2 is given. As explained earlier, this is the *a priori* best location for any card.

4. A penalty of MIN(11, number of misplaced cards) is assessed for every hole to which no card can currently move. The reason for the MIN function is to smooth the transition to a final position near the end of a solution where the number of useless holes must increase willy-nilly.

5. If there are no misplaced cards, the position is declared to be terminal and winning.

The misplaced card function is used in all search paradigms in one way of another, while the goodness function is used only in the pure best-first search.

The Constraint Satisfaction Method

The constraint satisfaction function that was developed for this problem is as follows: Once all the aces are in place, it is possible to determine what the final destination of every card must be. Any card that is not at its final destination will have to be moved if the puzzle is to be solved. Cards that are forward (to the left) of their destination are typically harder to move than other cards that are not forward. We examine only cards that are forward of their destination. If any such card cannot be moved, the problem instance represented by that node cannot be solved (i. e. it is deadlocked), and there is no point in searching its sub-tree. This is the basis of the procedure presented below which is able to reject about 50% of all configurations presented to it during a search as being non-solvable. We chose to apply constraint satisfaction only after the position is frozen (all aces in place). It is possible to try to identify deadlocked positions prior to this, but there is a great deal of computing required and the likelihood of finding an actual deadlock is very small.

The constraint satisfaction procedure is a recursive transitive closure procedure, presented in stylized form in Figure 2.

```
Procedure Deadlock(CardName,History,TrainLength) =
Begin

If Denomination(CardName) = ACE then return fail;        ! no moving aces
If OccursIn(History,CardName) then return fail;    ! card in hist = cycle
NewHistory = IncludeInSet(History,CardName);             ! Form new hist
If BehindPredecessor(CardName) then      ! behind pred: can't be moved
        return Deadlock(Predecessor(Cardname),NewHistory,Trainlength+1);

!!! the next section checks to see if a whole train can be moved into
!!! an area.  Two conditions are required for each slot: either it
!!! already has a hole there or the card there can be moved.
!!! After it is found that a slot can be vacated, the history is peeled
!!! back to remove the card for which slot was being prepared.

If SpaceForTrain(Locat(Predecessor(CardName)),TrainLength) then
L1: Begin
      For I=0 to Trainlength Do
      if Locat(Predecessor(CardName)) + I = occupied then
        if Deadlock(Name(Locat(Predecessor(CardName)) + I),
                    PurgeHist(History, CardName + I), TrainLength - I)
             = fail then leave L1;
       Return Success
    End;                                        ! L1
Return Deadlock(Predecessor(Cardname),HistPtr+1,Trainlength+1);

end;                                        ! Deadlock
```

Figure 2: The Deadlock Detection Algorithm

The intuition for the algorithm in Figure 2 is: any card that is too far forward could constitute a

serious block to solving the current state. The procedure Deadlock is called with the name of that card, a blank history, and a trainlength of 0. Deadlock then tries to see if the card can be moved, either immediately or as a result of clearing away other cards first, without creating a cycle; i. e. asking to move a card that is already in the history of cards that want to be moved. As each recursive call on Deadlock is made, the history is updated to include the card that most recently wanted to be moved. If a card cannot be moved, it attempts to move the predecessor cards and then move the card in question. Each time a card needs to have its predecessor moved, trainlength is increased by 1. If the predecessor is not being moved, the trainlength reverts to 0.

If the transitive closure fails on any card that must be moved, then clearly the state is not solvable. When cards are not immediately moveable, sequences of predecessor cards (trains) that have to be moved first are accumulated. The need to move trains around is what makes Superpuzz both interesting and difficult. The critical observation for the deadlock detection algorithm was noting that the history must be adjusted *backwards* as the algorithm attempts to deal with moving a train, one card at a time. Working out the present algorithm was actually a far from trivial matter. It can probably still be improved upon, as there are a number of conditions that it does not at present catch, but they occur rarely enough and are so expensive to detect that it did not seem worth including them in a pragmatic program[1] .

To further enhance the power of deadlock detection, two additional methods are invoked:

1. Once a position passes the deadlock detection, the search first attempts to make all forced moves in a cannonical order. A forced move is one that moves a card to a position behind a card that is already in its final position when the moving card is either at the right edge or has its successor already behind it (so that moving its successor cannot be affected by moving it). Intuitively, there is no motive to delay moving such a card. The moves are made in a cannonical order as the move generator always works in the same way, and the first forced move that is encountered is always selected. This "single-threading" of positions that are likely solvable, results in somewhat reduced effort for the search in those cases that are, in fact, solvable. It results in reduced effort and reduced utilization of hash table space in those cases that are not solvable. Making the moves in cannonical order tends to produce the same positions over and over again, even if they were approached through different deadlock tests. This collision of previously tested positions when checking the hash table produces notable savings in the search.

2. When the deadlock detection pronounces a position as not solvable just after moving the last ace into place, some efficiencies are possible. Clearly, if moving that ace resulted in

[1]Since this was written and the experiments described herein performed, an improved version of the Deadlock algorithm has been developed. This algorithm additionally requires that a card not be moved if another card is supposed to be moved behind it, until that card has, in fact, been moved in the deadlock testing. Also, if a card is hypothesized to move to a certain location, that location is reserved for that card until the transaction has been made. These two new facets of deadlock detection have improved the ability of the algorithm to the point where it is able to reject about 88% of all non-solvable positions

a not solvable position then moving any non-ace prior to making the ace move will not change the solvability unless the space behind the ace can be utilized to move the 2 of that suit there. In cases where this is not possible, we prune all non-ace moves because of the following additional reasoning. If the position is solvable, it must be by moving one of the aces along the left edge before moving the last ace not on the left edge. This may just as well be done immediately, since it will then permit trying the deadlock detection algorithm on a new grouping of aces along the left edge. The earlier such groupings are tested in the search the better, since the early rejection of unsolvable configurations materially speeds up the solution. Clearly, if such a configuration passed the deadlock detection, nothing has been lost.

Results

The experimental design involved testing each of the four search algorithms with and without deadlock detection. Each of these eight algorithms were tested on the same 100 randomly generated instances of 4x6, 4x7 and 4x8 puzzles. This made a total of 8 program types, each tested on 300 puzzle instances making a total of 2400 data points. For each program a 64K hash table was used. Experiments showed that, as expected, the size of the hash table significantly affects the problem solving power of the program. Unsolvable cases frequently required over 32K nodes to exhaust the space of nodes that needed to be tried to prove unsolvability. For the relatively uninformed algorithms: A*, BF1, and DF, deep solutions usually meant visiting many nodes. A 16K or 32K hash table would have meant quite a lot fewer solutions for these algorithms because of their exhaustive search nature. Therefore, a 64K hash table seemed the appropriate size to use.

All programs were implemented in a single piece of code with compile time switches that allowed compiling the 24 different programs required. Techniques such as incremental updating of data structures were used to make the code efficient, and the shared code tended to make the programs directly comparable in efficiency. The outputs required were: A statement of whether the problem instance was or was not solvable, and a satisficing (not necessarily optimal) solution. A maximum of 10 CPU minutes was allowed per problem, and all programs except the depth-first search had to halt when the hash table was full (because without hash table space no further representation of the tree would be possible). Most unsolvable problems were designated as such by all the programs. A problem that was not solved by any program was considered to be unsolvable, although it is possible that a very small percentage of these were solvable. If a program failed to solve a solvable problem, it received a default time of 10 minutes, a default number of nodes of 128K, and a default depth of 60 for 4x6, 90 for 4x7, and 120 for 4x8. All test runs were made on VAX-11/780's.

Table 1 shows the average solution length, average number of nodes per solution, average time to complete, and number intractable for the eight programs for each of the three problem widths. As expected, effort in all categories goes up with problem difficulty. It should be noted that the A*, BF1,

	Size = 6		Size = 7		Size = 8	
	Solvable	Unsolv.	Solvable	Unsolv.	Solvable	Unsolv.
BF1 (deadlock)						
Av. Sol. Len.	23	--	32	--	47	--
Av. Nodes	2800	8170	9624	41407	22611	82139
Av. Time	9.6	23.8	29.9	186.8	80.5	373.1
No. Intract.	1	0	3	3	9	9
A* (deadlock)						
Av. Sol. Len.	22	--	33	--	62	--
Av. Nodes	5489	8167	24187	41407	59162	82134
Av. Time	13.3	22.7	70.9	186.3	222.0	372.8
No. Intract.	1	0	7	3	26	9
DF (deadlock)						
Av. Sol. Len.	22	--	31	--	50	--
Av. Nodes	4618	9299	21906	41331	50636	81275
Av. Time	9.4	24.8	64.0	185.8	176.9	370.7
No. Intract.	0	0	6	1	19	5
BF2 (deadlock)						
Av. Sol. Len.	26	--	36	--	48	--
Av. Nodes	1775	8320	6600	41460	10770	82135
Av. Time	3.4	25.6	20.6	188.1	29.6	373.4
No. Intract.	0	0	2	3	2	9
BF1						
Av. Sol. Len.	23	--	32	--	48	--
Av. Nodes	2896	15760	10140	42121	24603	85665
Av. Time	9.4	55.9	28.4	189.5	84.9	390.7
No. Intract.	1	1	3	3	10	9
A*						
Av. Sol. Len.	22	--	34	--	65	--
Av. Nodes	5711	15760	25923	42121	62705	85665
Av. Time	12.6	53.0	72.7	189.0	231.9	387.0
No. Intract.	1	1	8	3	29	9
DF						
Av. Sol. Len.	22	--	33	--	54	--
Av. Nodes	4834	15760	24476	42121	54647	85665
Av. Time	7.9	46.2	62.6	185.9	178.3	376.0
No. Intract.	0	1	6	3	18	9
BF2						
Av. Sol. Len.	26	--	37	--	50	--
Av. Nodes	2912	15760	8376	42121	15145	85665
Av. Time	9.7	57.0	26.6	189.6	47.6	384.0
No. Intract.	1	1	3	3	5	9

Table 1: Performance Statistics

and DF algorithms all use the same knowledge. A* and DF use this knowledge for bounding

purposes; i. e. to determine that a solution cannot be completed in a given number of ply. BF1 however, uses this knowledge as a simple measure of goodness. In view of the above, it is startling to observe the overwhelming domination of the BF1 algorithm over A* and DF. Because of our method of punishing programs that don't solve solvable problems, the average solution length is a good indicator of quality of solutions on a given problem set. Here one can see that while the average solution length is approximately the same for all three algorithms on the 4x6 and 4x7 sets, the resources required (nodes and time) are considerably less for BF1. In the 4x6 set, the solving performance differed by at most one problem among the various methods, so the 5% degradation of solution quality for BF1 is real. However, this is a transitory effect, and by the time the 4x8 problem set is reached, BF1 clearly outperforms the other two algorithms in quality of solution (because our performance criterion also includes performance on the problems that an algorithm fails to solve). In general, the most knowledgable search (BF2) outperforms the others by all four criteria; the degree increasing as the width of the problem size increases. The performance on unsolvable problem instances shows relatively little difference among the algorithms, although it can be seen that DF has fewer intractable unsolvable problems in the 4x7 and 4x8 categories, thus reflecting the fact that depth-first searching is considerably more time-efficient than any other method.

		Nodes			Times		
	Search Technique	Size Change 6:7	7:8	6:8	Size Change 6:7	7:8	6:8
Deadlock	BF1	3.44	2.35	8.08	3.11	2.69	8.37
	A*	4.41	2.45	10.80	5.33	3.13	16.68
	DF	4.74	2.31	10.95	6.81	2.76	18.80
	BF2	3.72	1.63	6.06	6.06	1.44	8.73
No Deadlock	BF1	3.50	2.43	8.51	3.02	2.99	9.03
	A*	4.54	2.42	10.99	5.77	3.19	18.41
	DF	5.06	2.23	11.28	7.92	2.85	22.57
	BF2	2.88	1.81	5.21	2.74	1.79	4.90

Table 2: Performance Ratios

Table 2 shows the ratio of increase in number of nodes and time to solution for each algorithm type as the width of the problem increases from 6 to 7, and from 7 to 8, and overall from 6 to 8. These data show that, in general, the more informed the algorithm is, the less the degradation in performance as the problem gets more difficult. This is an expected result; the trees grown by the most informed algorithm grow at a lower exponential rate than those grown by the less informed, more brute-force-like algorithms. The BF2 search has the most information. BF1 has had its coefficient tuned so it is more responsive with its sole performance measure (misplaced cards) than either of the remaining

algorithms. Thus it is next most informed. The A* and DF have the least information. Although BF2 is only slightly superior among the 4x6 algorithms, it has achieved an overwhelming dominance by the time the 4x8 puzzles are considered.

	Nodes			Times		
Search Technique	Size 6	7	8	Size 6	7	8
BF1	0.97	0.95	0.92	1.02	1.05	0.95
A*	0.96	0.93	0.94	1.06	0.98	0.96
DF	0.96	0.89	0.93	1.19	1.02	0.99
BF2	0.61	0.79	0.71	0.35	0.77	0.62

Table 3: Ratio of Effort with and without Deadlock Detection

Table 3 shows the performance ratio of a search method with deadlock detection, to the same method without deadlock detection. If the ratio is less than 1.0, then improvement resulted; otherwise the cost of the additional work was not beneficial. As expected the deadlock detection improves performance on the nodes measure in all categories. For several categories however, doing the deadlock detection resulted in enough additional time costs to produce greater over-all effort when deadlock detection is invoked. Table 3 shows a rather surprising and interesting result: BF2 makes best use of the constraint satisfaction process. We believe this to be a general result of considerable importance, and discuss it in the next section.

Discussion and Conclusions

Superpuzz is a very difficult game; much more difficult than standard puzzles. Further, the difficulty of the game goes up as the width of the puzzle is increased. In this study we examine the 4x6, 4x7 and 4x8 games. Although the branching factor in each of these remains the same, the solution depth increases significantly with increases of width and the percent of unsolvable problem instances increases also.

The solution process can be thought of as occuring in two phases. In phase one, the combinatoric power of search attempts to see whether it is possible to obtain a position where all four aces are in place. When this has occurred, the deadlock detection algorithm is invoked, which is powerful enough to reject about one-half of all positions it encounters. When a position passes the deadlock algorithm, phase two is invoked. Here by relatively small searches, the solution is either found or rejected for the sub-tree defined by moving the fourth ace into its final position. In case no solution is found in the sub-tree, phase one again obtains control.

Given that the combination of deadlock detection and very small searches in phase two is very efficient, certain ideas emerge. If the position is solvable, then it is advantageous to reach phase two as quickly as possible. The BF2 search does this most effectively. It is not at all unusual to have the first configuration discovered by the BF2 search be solvable, whereupon the solution proceeds immediately. In those cases where this does not happen, the first dozen or so attempts do usually yield a solvable phase two. Only in cases where the solution is very contrived, or where there is no solution, is the BF2 procedure outperformed by others. In the case where there is no solution, all phase two positions must be explored and the procedure that reaches these with the minimum amount of effort is the most effective. From this it can be seen that some knowledge of what percentage of problems is solvable is instrumental in deciding on a search paradigm. In this research, the BF2 evaluation function is expensive to compute, and traversing up and down the tree is expensive also. But, while the BF2 search is only marginally superior in the width 6 puzzle, it becomes completely dominant by the time the width is increased to 8.

Let us consider why one search paradigm is better than another. A* and DF are really quite similar. They probe to new depths in a breadth-first style that takes advantage of certain efficiencies. A* knows about effort remaining and builds a permanent copy of the tree, which it continues to expand at the best leaf nodes. DF also knows about effort remaining and gets its power from great efficiency in space and time. However, neither is able to venture very far on a probe down a branch unless it is continuously having success (reducing the number of misplaced cards). Any failure to improve this measure would immediately force the A* search to try another branch. Because the DF search has an iteration constant of 2, two non-successful moves are allowed in an expansion before returning. Since the criterion for success is rather simplistic and it is very likely that any solution will require a number of non-constructive or backward-appearing steps, it is unlikely that either search will be able to make significant forward progress through such territory. Instead they plow steadily forward until the treacherous territory is overcome by *all* highly evaluated branches, and then pursue one to successful conclusion.

Now consider the BF1 search paradigm. We had originally hypothesized that the coefficient 1.8 that multiplies the number of misplaced cards in the BF1 algorithm represents an estimate of the number of times a misplaced card must be moved before it reaches its final destination. In pursuit of this notion, we then sought to improve performance based on the consideration that a misplaced card is less likely to require more than one move to reach its final destination when almost all cards are already in place, than it is when very few cards are in place. However, we were not able to produce any meaningful improvement by making the constant 1.8 into a variable.

We now believe the meaning of the 1.8 is that it is an **adventurousness coefficient.** In BF1 the value of a descendant node either decreases by .8 units in case the number of misplaced cards is reduced, or is increased by 1.0 in case this does not happen. This allows the descendant node to put some distance between it and its competitors when it is able to take a few constructive steps intermixed with some that do not appear so constructive. The essential point is that it does not have to produce "progress" on every move. The degree of such adventurousness is what the constant 1.8 controls, and for the given domain and evaluation function it appears to be best.

It appears to us that this is an important idea with potentially wide applicability in searching. It is possible to think of the adventurousness coefficient and the evaluation function as forming a primitive planning unit. The plan permits all actions, but the evalution function selects those that appear to be "good". The adventurousness coefficient terminates (at least temporarily) those plans that are not proving out.

The BF2 search is even more adventurous since it can gain numerous points in heuristic value by placing a card into what is considered an advantageous location. This allows it to penetrate deeply in certain branches that it "likes" while leaving others behind. Clearly, it lives or dies by the goodness of its evaluation function. In some cases the evaluation function will lead the search up a blind alley which would normally have to be explored extensively before it was found to be blind. Here is where the constraint satisfaction helps the most: *it can disenchant the search when it is in a blind alley causing it to look elsewhere*. This also happens for the other searches, but is not nearly as effective because these searches are not as adventuresome. In general, the number of backward steps taken upon discovering a deadlock would be proportional to the degree of adventurousness of the algorithm.

It is true for every search paradigm that once a node is known to be deadlocked, its successors will never be expanded. However, these savings can only be realized once such a sub-tree is reached. This is where adventurousness is important. If situations where constraint satisfaction procedures can be applied occur only after almost all important branches have been pushed to the same depth, then the savings will not be very great in the case where a solution is possible. When no solution is possible, the savings will be the same for all algorithms. However, when a solution exists, the most adventurous search has a big advantage, as long as it has some way of recovering when its optimism is misplaced. This is what constraint satisfaction brings to the best-first search and why the BF2 search improves the most when benefitted by constraint satisfaction (see Table 3). *Adventurousness enhances the effect of constraint satisfaction, by using its decisions to move forward rapidly (if favorable) and retreat rapidly (if not).*

* * * * * * * *

A minimal solution to Figure 1A is: (read left to right; names of moving cards only, except for aces where the row is also given).

C2 S1(Z) D4 D3 D4 C1(Y) C2 D1(X) C3 S2 H2 S4 C4 D2 S5 C5 H4 S3 D1(W) H1(X) S4 D2 H2 S5 D3 D4 D5 H5

References

[1] Korf, R. E., "The Complexity of Brute-Force Search", Technical Report, Department of Computer Science, Columbia University, 1984.

[2] Knuth, D. E., & Moore, R. W., "An Analysis of Alpha-Beta Pruning", *Artificial Intelligence*, Vol. 6, No. 4, 1975, pps. 293-326.

[3] Nilsson, N., *Problem Solving Methods in Artificial Intelligence*, McGraw-Hill, 1971.

[4] Slate, D. J., and Atkin, L. R., "CHESS 4.5 -- The Northwestern University Chess Program", in *Chess Skill in Man and Machine*, P. Frey (Ed.), Springer-Verlag, 1977.