Comparing Domain-Specific Knowledge Used to Solve Sokoban

Shay Palachy

Inbar Naor

Abstract—Previous work done on the subject of finding optimal solutions to Sokoban problems has both demonstrated the incredible complexity of the problem and made some substantial progress towards solving it. Since the major obstacle all attempts to solve Sokoban must face is the enormous resulting search space, this work attempts to compare and evaluate both some of the techniques commonly used to reduce the search space - namely deadlock detection and move macros - and some of those used to better navigate it - heuristics. Our results show that while the classic heuristic used for admissible informed search of Sokoban problems is indeed the dominant one, combining it with other, less effective, heuristics can result in an inadmissible heuristic function that can enable informed search algorithms to find non-optimal solutions while expanding significantly less search nodes. Also, we present a novel use of the classic perfect matching between boxes and goals on the Sokoban board - up until now used to (time and space) costly achieve a tighter admissible heuristic - to achieve a significant reduction of the number of nodes expanded with almost no computation cost, while making the heuristics used with it inadmissible.

I. INTRODUCTION

Sokoban is a single player computer game created in 1981 by Hiroyuki Imabayashi. The game features a board, on which walls, boxes, goal squares and an agent are located. The goal of the game is to use the agent to push all the boxes, through the maze created by the walls, onto goal squares. The number of boxes can vary from puzzle to puzzle. A box can only be pushed (not pulled) and only one box can be pushed at a time. In addition, the agent cannot walk through boxes or walls. Getting the boxes to the goal is in itself a difficult task at some of the game stages, while doing it with the minimum number of moves is much harder.

Solving Sokoban is not only interesting, but also relevant to some real-life uses of Artificial Intelligence, as it simulates to a good degree a robot having to move crates about in a warehouse, and place them in given target locations. Former research on the subject has shown Sokoban to be PSPACE-Complete [2] and NP-Hard [3].

The difficulty of the problem is caused by three major properties when analyzing it as a search problem: First, the high depth of the solution in the search graph (some of the puzzles take hundreds of moves to solve); Second, the existence of unsolvable game states, called deadlocks; And third, the large branching factor of the search graph.

When looking at the problem from a single agent perspective, trying to find the optimal set of moves for the warehouse keeper that would lead to a solution, then the branching factor is 4, as the agent can move up, down, left or right at each turn. In fact, since in some of the states the agent cannot perform one of the moves and we rarely want to perform a step back, the average branching factor is somewhere between 2 and 3. However, the problem of moving the warehouse keeper around is a standard path-finding problem, and does not compromise the difficult part of the Sokoban problem, which is finding a correct sequence of boxes movements that solves the puzzle. Therefore, it is much more common to consider only these movements, ignoring the relatively easy problem of moving the warehouse keeper around the maze. Using this approach, we may move any of the boxes to any of four directions in each state, which leads us to a branching factor of $4 \cdot B$, where $B$ is the number of boxes. This number can grow over 100 in some puzzles.

II. PREVIOUS WORK

While a fair amount of research has already been made on the subject, the game of Sokoban is far from being solved. A common benchmark for Sokoban solvers is the standard 90-problem XSokoban suite (available at http://xsokoban.lcs.mit.edu/xsokoban.html). To date, no solver has yet been able to find an optimal solution for all 90 problems. In fact, the best solver created so far, Rolling box, has been able to find
an optimal solution for only 57 of these problems [1].

The basic search algorithm used in the Rolling Stone solver is IDA*. However, due to the large search graph, IDA* cannot solve a large amount of problems even with a good heuristic. In order to get better results, several domain specific enhancements were needed. The researchers used hash tables to avoid revisiting positions that can be reached via different paths and to eliminate loops in the search graph. Deadlock tables were used to detect deadlocks quickly on the fly. These tables were an instance of pattern databases, containing all possible configurations of boxes, walls and empty squares in a 5X4 region, and its deadlock value (true or false). In order to reduce the depth of the tree the researchers used Tunnel Macros treating long tunnels as one square, and Goal Macros pushing boxes directly to the correct goal when it enters the goal area. "Rolling Stone" also uses a move ordering techniques, preferring to move the same box in subsequent moves rather than different boxes, and periodically backing up the search path to allow a different order of moving boxes.

A different approach to the problem is a multi-agent approach introduced by Van Lishout. While Sokoban as a game is definitely a single agent game, it is possible to treat it as a multi-agent game by looking at each box as an active semi-independent agent in a group of agents cooperating in order to find a solution. The warehouse keeper is treated as a tool used by them. Using this approach, the researchers discovered a subset of problems that can be solved box-by-box (i.e., moving each box in its turn to the goal). In order to be possible to solve a puzzle that way, it has to satisfy three conditions: First, it must be possible to determine the order in which the goals should be filled, regardless of the position of the boxes; Second, it must be possible to move at least one box to a goal position without moving any other box; And third, after moving a box that satisfied the second condition to the goal, the maze still satisfies the first two conditions. While only 2 problems of the XSokoban set hold the Van Lishout property, having puzzles holding this property after relatively few moves is surprisingly common. Solving a puzzle box-by-box can be mimicked by running a BFS and examining if each encountered node contains a box that can be pushed to the goal or not. When such a node is discovered, it can be farther examined to see if the recursive condition also applies. If not, the search is continued. This gives solutions similar to the Rolling Stone solution generated by the move ordering scheme but in a much shorter time.

Some researchers argue that solving Sokoban completely using search techniques might be impossible given today's computing power. Therefore, several attempts to solve the problem as a planning problem have been made. The naive approach, translating all properties of the domain to a planning representation (i.e., every move of the warehouse keeper is an action) has led to poor results. [1] a different attempt to solve Sokoban using planning used abstraction, so the planner needs only to solve a simpler abstract problem. The abstract problem was generated by a preprocessing phase, decomposing the puzzle into two types of objects - rooms and tunnels. Thus, the maze is reduced to a graph of rooms linked by tunnels. The actions refer to moving a box from one object to another. In effect, the search problem has been split into local components (moving withing a room) and a global component (moving between rooms and tunnels). The planning problem was solved using the standard planner TLPlan. The researchers tested the abstract planner on ten of the XSokoben set, comparing the length of the solution the the length of the solution produced by Rolling Stone. The length of the abstract planner solution was indeed much smaller than that of Rolling Stone. [3]

III. OUR WORK

As mentioned, one of the things that make the Sokoban problem so hard to solve is the large branching factor and the significant length of the average solution, resulting in a search space too big even for an informed search agent with good heuristics to solve in a practical amount of time. As a result, much of the significant work done on the subject has focused on reducing the search space through domain-specific techniques. One by product of this approach, is that not much work went into developing and comparing different heuristics, which can be used with informed search techniques.

Therefore, we chose to focus on two different tasks; First, to compare different search space reduction techniques used in precious work, by measuring their effect on the search graph. Second, to suggest and examine different heuristics for informed search algorithms, and compare their performance.

A. Implemented Search Algorithms

We chose to compare the performance of five different search algorithms:

1) Breadth First Search.
B. Search Space Reduction Techniques

Since the search space of Sokoban is so large, a considerable amount of work has to be done in order to reduce it before even considering the use of heuristics to further bolster the search agent. One possible improvement is to never consider the reversed action of the last action performed, i.e. never move a box to the position it just came from in two consecutive moves. It is possible to think of configurations where this improvement might actually hinder the solution or even cause potential deadlocks; However, since it is a rare enough case, and not evident in the puzzles we tested our solver on, we chose to implement it in our solver. For a general solver it would be best to keep it as an option rather than a default. This improvement reduces the search space a little, but not to a very significant degree, considering its size. Two more effective techniques are Macro Moves and Deadlock Detection.

1) Macro Moves: A very significant improvement we implemented was the Move Box To Goal macro, which is very much inspired by the "box by box" solving approach explained earlier in this report. When using this macro, if it is possible to move a box directly to a goal without moving any other box in the way, then this move is performed. This means that when searching for such a possible move, the problem is reduced to a simpler problem of finding the shortest path between the box and the goal closest to it (in our implementation). This problem is solved using an A* search agent with a simple Manhattan distance heuristic (measuring the distance from the specified box to the target goal), when all other boxes in the maze are treated as static walls, The movement of the box from its original position to the goal is done without expending any farther nodes in the original search graph, effectively encapsulating the solution of the sub-problems into a single node in the original problem.

This approach is actually an implementation of a common human heuristic approach, of focusing on a single box each time, attempting to either move to a goal tile, if possible, or to manipulate it and the other boxes on the board so as to make some goal tile reachable by it.

When using this macro, some of the problems (3, 5 and 6 in our benchmark set) were solved using a simple BFS, even though they are unsolvable by an unenhanced BFS. This demonstrates the importance of the reduction of the search space size. For some other problems, however, the use of the macro made the solution much harder to find, sometimes putting it out of the reach of our solver (such as in the case of problem 2 in our benchmark set). This happened when the closest goal to one of the boxes in the maze was a goal that was not supposed to be its destination in the final solution.

2) Matching Boxes To Goals: In order to solve the problem mentioned above, we used a heuristic (in the classic sense of the word, rather than the informed search sense of it) that tries to match each box to its optimal goal. We treated this problem as a problem of finding the lowest cost perfect matching in a bipartite graph, where group A of the graph was the boxes, group B was the goals and the costs was the Manhattan distance between each box to goal pair. This problem was solved by the Hungarian Algorithm, first presented in 1955 by Harold W. Kuhn. [4] Our solver uses an implementation of the Hungarian Algorithm to find the lowest cost perfect matching of boxes to goals in the initial state, and from then on uses this matching both to determine onto which goal tile to move a box when using the aforementioned macro, and when computing the score of any heuristic considering the distance between a box and its target goal.

Unlike the way it is used in the solver "Rolling Stone", we chose not to hold a matching for each state - which requires running an update algorithm for the parent state's matching in $O(n^2)$ - as it would have burdened our already limited memory resources, and would have effectively limited to solving even smaller level, and would have also lengthened run time considerably (this update algorithm is the dominant factor in the run-time of "Rolling Stone"). Instead, we run it once for the initial state, and all other states in the search refer to that computed matching. This means that in some cases the matching might become sub-optimal, after
a series of moves change the board’s position in a certain way. As a result, the *Sum Manhattan Distance* heuristic (and it alone) becomes inadmissible when using this option, although for many problems it will be effectively admissible.

3) **Deadlock Detection:** Another essential part of our solver is deadlock detection. As mentioned, the existence of deadlocks is one of the things making Sokoban difficult to solve, in comparison with other similar problems. In order to avoid them, a deadlock detection mechanism is required. We consider three possible deadlock types, composed of three or less boxes, as shown in figure 1:

1) A box in a corner.
2) Two adjacent boxes next to a wall.
3) Two boxes at the two sides of a corner.

![Figure 1: The three types of deadlocks detected by our solver](image)

These deadlock types are detected in any rotation and location, and in any variation of the surroundings (meaning it is detected even in a more complex position, that might include other boxes or walls).

During the graph search, before making an action, the solver checks if executing the action from a given state will result in one of these deadlocks. If so, this action is not considered further. Most of the Sokoban solvers implement deadlock detection by a pre-computation of a deadlock map, which is done by enumerating all possible box/wall placements in a specific region (4X5 in Rolling Stone) and checking if a deadlock is present. Later on, when considering a move, the solver queries this map to check if the move results in a deadlock. While this implementation is obviously efficient as almost no computation is needed to be done on-line, it is more space-consuming and harder to implement. Our solver implements an on-the-fly deadlock detection by computing the deadlock value of each state only when it is considered. Because we only dealt with relatively small problems when benchmarking different configurations of our solver, the reduction in performance is insignificant in our scale. More efficient implementations are possible, but are out of the scope of our work, as we have decided to check a large variety of domain-specific knowledge.

4) **Transposition Tables:** Since A* is optimal with an admissible heuristic only when searching a search tree, and not when searching a search graph, a transposition table eliminating equivalent nodes from the tree is needed. In order to do so, the search algorithm holds a hash set that contains all the states it already visited. In our implementation, the function determining equality of states is box-ID-insensitive, meaning that if two states have boxes is the same locations and only the IDs of the boxes are different, they are considered identical. In addition, if two states have the same box arrangement and the location of the warehouse keeper is different, but it is possible to move him from his location in the first state to his location in the second state without moving any box, it is also considered the same state. During the graph search, when a node is extracted from the fringe, its neighbors are added to the fringe only if an equivalent state was not considered before, effectively transforming the search graph into a search tree.

C. **Heuristics**

We examined five different heuristics and different weighted combinations of them, and compared the performance of the two informed search algorithms when using the different alternatives.

1) **Max Manhattan Distance:** This heuristic function scores a search state by computing the Manhattan distance of each box from the goal closest to it, and returns the maximal distance found. This is obviously an admissible heuristic function. However, when using the "Match box to goal" option, a better estimation is achieved by finding the lowest cost matching in the bipartite graph of boxes and goals, and returning the maximal distance between a matched pair in the optimal matching.

2) **Min Manhattan Distance:** This heuristic function scores a search state by computing the Manhattan distance of each box from the goal closest to it, and returns the minimal distance found. Like in the Max Manhattan Distance heuristic, the "Match box to goal" option can be used to achieve a tighter admissible heuristic function.

3) **Sum of Manhattan Distances:** This heuristic function scores a search state by computing the Manhattan distance of each box from the goal closest to it, and returns the sum of distances computed. This is of course an admissible heuristic, but, as mentioned at Junghanns and Schaffers article,[1] the gap between this lower bound value and the actual length of the solution is too large, making
it impossible to solve any large problem using this heuristic function alone. We use the "Match box to goal" option to get a tighter lower bound.

4) Number Of Empty Goals: As the name suggests, this heuristic function scores a search state by the number of empty goals on the board. It is not effected by the "Match box to goal" option.

5) Consecutive Moves: Inspired by the "initial" move ordering of the "Rolling Stone" solver, which first considers boxes that were the last to move. This heuristic returns a lower value if the box being moved is the same box as the last one that was moved, thus "encouraging" the solver to repeatedly move the same box when otherwise actions moving different boxes have the same score.

IV. IMPLEMENTATION

In order to compare the solving abilities of a search agents we such varied configurations of the techniques we mentioned, we created a very configurable Sokoban solver. This solver can be configured to use each of the five search algorithms mentioned, any combinations of the heuristics we detailed (with configurable weights), to detect any combinations of the deadlocks mentioned, and to use or not to use Move Macros and Box-To-Goal matching. To achieve this level of configurability, and to allow both modularity and extendability in the future, we first wrote a general search library, implementing various search algorithms and supplying an infrastructure for implementation of specific search problems. The library defines an abstract search problem and search state classes, which are later extended according to the specific problem domain. The extending classes are used to create the search graph, and get heuristics for the informed search algorithms.

Based on this library, we implemented a classes representing a Sokoban search problem and a search state. A Sokoban search state is quite a reach class. Every such State contains a mapping between each box ID and its location, the location of the man and a representation of the board. A reachability map is computed for each state by using a BFS search originating in the man’s location, showing which parts of the board are reachable by the man, without moving any box. The class representing the search problem is in charge of determining what action can and should be applied to every state; Applicability is defined by the rules of the Sokoban game, while worthwhileness varies from configuration to configuration, and is dependent on deadlock detection, heuristics used and other options.

Finally, we implemented our configurable Sokoban solver, named BoxPusher. The basic approach of all different solvers is the multi-agent approach treating every box as an agent while considering the movements of the man only as a tool to move the boxes. We chose to use this approach as well, and so when exploring the possible actions (represented by a pair of box ID and direction) from each state, an action is entered to the legal actions list only if the man can reach the spot from which the box should be pushed. And so, the actions generating our Sokoban search state do not move the man itself tile-by-tile, but rather boxes, and "teleport" the agent directly to the tile resulting in the chosen box push, only checking that he can reach this tile (by referring to the aforementioned reachability map).

After a solution composed of box pushes is found, it is translated to a solution composed of man (warehouse worker) moves, by a series of IDDFS searches, each finding the optimal path from the tile the man occupied after performing the previous box push to the tile he needs to occupy on order to perform the next push. This means that for a given sequence of box pushes, the optimal sequence of man pushes that can bring about the given sequence of box pushes is found. However, different box-pushing sequences of the same length might (and many times do) require respective man moves sequences of different optimal length, and so in the man-moves sense of solution, no optimal algorithm is assured to find the optimal solution in man moves. In this case we take the common approach in research of the subject, and a solution’s optimality is determined only based on it’s box pushes length.

We also implemented a configurable search condition mechanism, that allows the user to limit the search by time or number of nodes extended, or both.

V. RESULTS

In order to compare the performance of the solver when using different search algorithms, heuristics and deadlock detection techniques, we chose a set of 10 Sokoban levels (found in a website dedicated to user-created Sokoban levels), ranging from very easy to moderate, most with up to 4 boxes in a problem (with one problem having 5 boxes and another 6); We were limited by the hardware used to run the solver, as any level with more than 3 boxes and larger than a certain (quite small) size creates an enormous search space, with at least tens of millions of states, and in which finding a
solution after searching only several hundreds of thousands of search nodes is impossible. Since the limited memory of our test machine enabled only several hundreds of thousands of nodes to be held in memory at a time, we had to limit ourselves to these simpler problems. These, however, are enough to demonstrate the space reduction achieved using the techniques we implemented, only in a smaller scale.

A. Comparing Search Algorithms

1) Breadth First Search: Unsurprisingly, BFS performs pretty poorly. It is able to solve the more simple and small Sokoban problems in our 10 level benchmark set - namely 1, 2, 4 and 8 - but of those problems, only 1 and 4 are solved in time comparable to the better configurations of "BoxPusher". Problems 2 and 8, however, are solved after searching more 12,000 and 430,000 nodes, respectively. This demonstrates that the large branching factor and solution length of even a simple Sokoban problems, result in having to search through an enormous amounts of nodes before finding the "shallowest" solution, and so why using BFS to solve Sokoban is not practical. We will later show that if coupled with some search space reduction techniques, BFS can solve even more problems and while searching significantly less states.

2) Depth First Search: DFS performs no better than BFS, and a plain version - using no space reduction techniques - is able to solve four out of the ten problems in the benchmark set (though not the same four BFS can solve, but one different), and only two of them optimally (not surprising, as it is not complete). This demonstrates not just the size of the search space, but also the low frequency of solutions in it.

3) Iterative Deepening Depth First Search: IDDFS performs no better than BFS, and a plain version - using no space reduction techniques - is able to solve four out of the ten problems in the benchmark set (though not the same four BFS can solve, but one different), and only two of them optimally (not surprising, as it is not complete). This demonstrates not just the size of the search space, but also the low frequency of solutions in it.

4) A*: As expected, A* performs much better than the former algorithms, and using a single heuristic - with no search space reduction techniques - it can solve all problems in the benchmark set, and solve those solvable by the uninformed search algorithms while searching significantly less nodes.

5) Iterative Deepening A*: IDA* performs even better than A*, naturally. Not only can it solve the whole set even when using neither deadlock detection nor move macros, but it can also find a solution when running with almost any configuration of heuristics, deadlock detection and other options, actually finding a solution with some configurations for which A* requires too much memory. This is of course due to IDA* space efficiency.

B. Search Space Reduction - Macro Moves

In order to examine the effect of "Move Box to Goal" Macro, we compared the number of expanded nodes in the search graph, when using BFS with and without the macro on our benchmark problems set. The most prominent effect is evident in problems 3, 5 6 and 7. These problems were unsolvable using BFS without the move macro (under the memory limitation we had), and solvable when using it. The numbers in the BFS row for problems that were solvable represent the largest number of nodes that can be extended without causing the program to stop of lack of memory. Therefore, we can say that in order to solve bench3 using BFS, for instance, at least 300000 nodes have to be extended. When using the macro on these problems, we got a reduction of more than 99.8% on problems 3, 5 and 6 and 95.5% on problem 7. Similar percentage of reduction was perceived on the problems that were solvable by BFS with no macro. Problems 1 and 8 had a reduction of 98 and 99.8 percent respectively, while the reduction on problem 4 was 40%. As mentioned, the performance on problem 2 was degraded by using this macro without matching box to goal, and while it was solvable by plain BFS with 12165 extended nodes, it was not solvable with the macro alone. Problems 9 and 10 were unsolvable by BFS with or without the macro.

C. Search Space Reduction - Deadlock Detection

Of the three types of deadlocks "BoxPusher" detects, the first - detecting boxes stuck in corners - is by far the most efficient, bringing about the most significant reduction in search space in all problems but one. This is to be expected, as this is also the
most frequent deadlock, as it is impossible to create a level without at least four corners - while some have significantly more - and it involves only one box. The other two deadlocks involve two boxes, and thus are much rarer. It is notable, however, that problem 7 in the benchmark set benefits more than detecting the second deadlock - a pair of adjacent boxes beside a wall - than even the first one. This shows that all deadlocks can become significant in certain problems.

These results are reflected in figure 2, that displays the reduction in the search space resulting in the detection of each type of deadlock. The graph represents the number of nodes expanded when solving the given problem using IDA* with the Sum Manhattan Distance heuristic and the given deadlock type detector, as a percentage of the number of nodes expanded when solving the given problem using IDA* with the Sum Manhattan Distance heuristic and no deadlock detection.

Figure 3: Space Reduction by Deadlock Detection

**D. Heuristics**

We now present the effectiveness of using each heuristic separately, by comparing the number of nodes expanded when solving the problem set with and without each heuristic. Four out of the five heuristics present very inconsistent performance over the range of the benchmark set problems, achieving a reduction of between 0% and 94% if the number of nodes expanded on an IDA* search. The Sum Manhattan Distance heuristic, however, presented quite a consistent record, with a reduction of over 80% of nodes expanded for all measurable. It’s addition to a plain IDA* also solves all problem unsolvable (under the memory limitation) before. Figure 4 presents the present of nodes expanded reduced by each heuristic on each of five problems of the benchmark set (problems 1, 2, 5, 6 and 8) - only problems which were solvable by a plain IDA* or for which we had a solid lower bound on the number of nodes required to find a solution by a plain IDA*.

![Space Reduction by Deadlock Detection](image)

**Figure 4: Reduction in nodes expanded by heuristics**

1) **Max Manhattan Distance:** This heuristic provides a considerable improvement in the number of nodes expanded when solving several problems in the benchmark set, namely problem 1, 2, 5 and 6. When comparing the number of nodes expanded when solving these problems using a plain IDA* (no enhancements) with an IDA* using the Max Distance heuristic, an average reduction of 77.5% is achieved. Other problems cannot be solved by a heuristic-free informed algorithm, even when using deadlock detection, unless Move Macros are used, in which case the search space is reduced to such a degree as to prevent measurement of the heuristic’s effect.

A significant outlier, however, is problem 4, for which using this heuristic increases the number of nodes expanded by IDA* from 4 to 118, and enormous blow-up (when measured in percentage). This weird result is currently unexplained.

2) **Min Manhattan Distance:** The Minimum Distance heuristic perform much worse than the maximum one; It reduces the number of nodes expanded only when solving problems 1 and 2 of the set. When solving problem 4 the number of nodes expanded remains the same, and all other problems - unsolvable by uninformed, plain IDA* - do not become solvable (at least under our memory limitations), and so cannot be used to measure this heuristic function effectiveness.

3) **Sum of Manhattan Distances:** This heuristic is by far the best one, achieving a reduction of nodes expanded when searching on all measurable problems, and solving all problem unsolvable (under memory limitations) by a plain IDA*. As mentioned, it presented quite a consistent record, with a reduction of over 80% of nodes expanded
for all measurable, and an average reduction of around 94% on the five measurable problems.

4) Number Of Empty Goals: This heuristic presented the most inconsistent results on the five measurable problems, achieving a reduction of around 60% and 86% for problems 2 and 8, respectively, while a reduction of 0% on the other three problems. The two problems for which this heuristic seems to be more effective do not present easily discernible differences from the other three as to explain this significant difference. Measuring on a much larger number of problems is required to determine the cause of this difference.

5) Consecutive Moves: This heuristic presented inconsistent results, much like the last one, though to lesser degree. It achieved a reduction of around 20% and 56% for problems 2 and 8, respectively, while a reduction of around 0% on the other three problems. Again, a larger test set is required to determine the cause of this difference. Per the data we have, this seems to be the least effective heuristic of the five tested, in average.

E. Matching Boxes To Goals

As we have explained before, an option that can bolster the performance of the first three heuristics is the option to find the optimal match between boxes and goals using the Hungarian Algorithm. This technique, however, showed an inconsistent effect when used with searches on the benchmarking set: For problem 2, it reduced the number of nodes search by 49.5% on average, and 47.3% on problem 5, but only 24% on problem 8, in which it helped only the Max Manhattan Distance heuristics, while not effecting the Sum heuristics and only hurting the Min one (other problems either had to small numbers when solved when using a heuristic to effectively measure the effect of this technique, or not solvable using only a heuristic, thus requiring introduction of other search space reduction techniques in order to measure this one, which would not have allowed a clean and independent measuring). This means that it can only be used to a limited effect on Sokoban problems, and will exhibits can performance in problems for which the initial matching becomes irrelevant after some moves.

This results, however, show also that our novel approach of computing a matching for the initial state only, and using it as a heuristic to estimate what might be the best matching for all other state is viable, and can bolster performance consideredly in some cases, while only requiring a one time \(O(n^3)\) calculation, in comparison with the \(O(n^2)\) computation for each state that a consistently optimal matching for each state requires (like the one used in “Rolling Stone” to achieve a tighter admissible Sum Distances heuristic).

F. Admissibility Versus Inadmissibility

Three major weighted combinations of heuristic functions were tested, creating three new inadmissible heuristic functions, where each of the component functions received a weight of 1. All three configurations reduced the number of nodes expanded on search significantly in several cases, when compared to the best search (least nodes expanded) made with a single heuristic (always Sum Distance, in our case).

1) Sum Manhattan Distance + Number of Empty Goals: This combination expanded 73% less nodes on average, on problems 2, 3, 5 and 7, when compared with only Sum Distance heuristic. It however expanded around 60000% more nodes on problem 8 in relation to Sum Distance (around 700 compared to 11 nodes).

2) Sum Manhattan Distance + Min Manhattan Distance: This combination expanded 42.6% less nodes on average, on problems 2, 5, 7 and 8 when compared with only Sum Distance heuristic.

3) All 5 heuristics: This combination expanded 67.72% less nodes on average, on problems 2, 5 and 7, when compared with only Sum Distance heuristic. It however expanded around 400000% more nodes on problem 8 in relation to Sum Distance (around 43660 compared to 11 nodes).

VI. Conclusion

The results of running our Sokoban solver in different configurations on our benchmarking set show that the most effective search space reduction measure is the Move Macro. The strength of this approach is in the fact that it effectively breaks down the main problem into much smaller subproblems. Perhaps the most prominent effect is the drop of the branching factor from around \(4 \cdot n\) (where \(n\) is the number of boxes) to 3 (as we allow no strict back-offs); This alone shrinks the search space significantly, and with the addition of a shorter solution length (as we have to move just one box to a goal tile), we get a much smaller problems. This means that most of the search in "main" problem is actually done to manipulate the board layout in such a way to position one or more boxes where they can reach a goal tile; Then, the instance of a sub-problems finds how to get them
there optimally. The result can be as dramatic as a reduction of the number of nodes required to search to find a solution from over a million to around ten! This is a drop of around 5 orders of magnitude!

Deadlock detection has also proved to have a significant impact on search space size, though not as much as Move Macros. It is obvious that the more common deadlocks will have a greater effect, but rarer ones also having an effects means that checking for more types of deadlocks will further help to reduce the search space. Still, it is reasonable to estimate that the majority of the improvement from deadlock detection will come from searching in a small area and for common deadlocks, while rarer ones will require more time to identify and will provide less gain. This means that a threshold exists beyond which deadlock detection becomes more of a burden than a benefit. Whether this threshold is the $4 \times 5$ area search established by the work of Junghanns and Schaeffer is yet to be seen.

Finally, several admissible heuristics has shown to have a significant effect on search when compared with uninformed search, but the total domination of the sum of Manhattan distances from boxes to goals as the tightest known admissible heuristic, leave it as the only realistic option for an admissible informed search. The significant effect of some inadmissible heuristic combinations, however, shows that the other heuristics might still has a roll to play in a more greedy inadmissible search, as a way to find non-optimal solutions quickly.

REFERENCES