Search Algorithms for Peg Solitaire

muthukumar suresh

February 2015

Contents

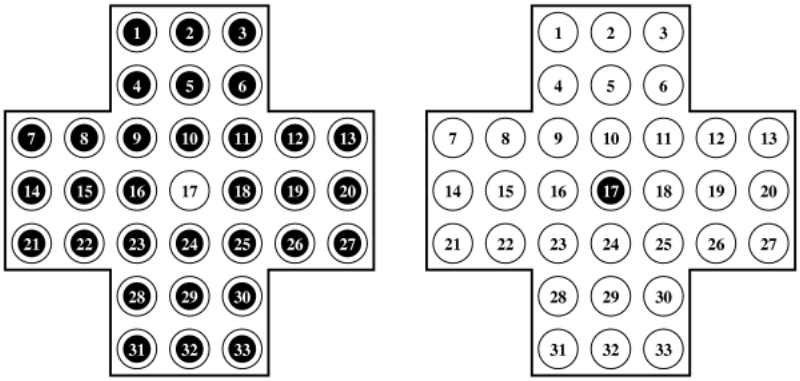

Figure 1: Peg Solitaire Starting and Ending Position

1 Introduction

Peg Solitaire, is a game that requires the player to convert the board shown in the image from the first board position shown in the figure 1 on the left to the board shown on the right. In this project, I have analyzed the various ways to solve this puzzle.
I have concentrated on two main algorithms:

- Iterative Deepning Search
- A* Algorithm

Iterative Deepning Search, is a low memory approach to solve the problem. We set the maximum depth at each iteration and do a normal depth first traversal up until the limit set by that iteration. By doing this, we can easily bound the run time of our algorithm. IDS, is an easy to implement algorithm in case of problems where we dont know much about the problem, i.e., it is hard to develop heuristics. In such cases, the only possible way is to efficiently iterate through the search space. As we will see when we discuss the algorithm in detail, IDS performs extremely well for small problems, but once we start to expand the problem and the number of states increase, it quickly becomes infeasible. In case of peg solitaire, there are ${}^{31}C_{30}$ possible states and iterating through the search space is quite impossible. We need to develop heuristics to make out problem tractable. Hence, we use the A* algorithm.
In A* algorithm, we use heuristics to navigate our search space in a more planned manner. To solve the peg solitaire puzzle, I have developed **3 heuristic measures**. Each one more knowledgeble than the previous. As we shall see, the final heuristic can solve our puzzle in very few moves. Another important aspect of developing a A* algorithm for Peg Solitaire is to prune the search space, sometimes it is possible that we navigate down stray paths which don't lead to the solution, in such cases, we can develop **pruning strategies** to avoid navigating down those paths that don't lead to a solution.
This report is organzied as follows: Section 2 talks about the IDS algorithm. Section 3 explains the A* algorithm and the heuristics used to solve the problem. The pruning strategy is also explained in section 3. Section 4 puts forward our conclusion and finding about solving the Peg Solitaire Problem.

2 Iterative Deepning Search

2.1 Implementation

The implementation of IDS is very simple. In each iteration, we choose a move to perform. A valid move is one where there are 3 valid positions in a row and the first 2 have pegs in them and the third is empty. At each stage, we choose a possible move in a FCFS fashion and now that move gives us the current state. We check all possible paths upto the depth specified in that iteration.
For our problem of peg solitaire, it makes no sense to have depth checks for anything other than $numberOfPegs - 1$. So even though our implementation has the option of iterating through depth values, it is not necessary.
Since we create no extra datastructures and we maintain only one copy of the board, we use no extra space other than the in-built stack and the maximum number of stack frames cannot be more than 31. Therfore, IDS is efficient in memory. As we shall see it is not so in time.

2.2 performance

Performance of IDS interms of time is not very great. It's fine
in terms of memory and simplicity but not in time.

NOTE: Since for the problem of peg solitaire, it makes no sense to talk of iterating through depth. We will use a single depth value of the number of nodes -1.

As we can see in figure 2, IDS performs very well in situations where we have less number of states and fewer possible transitions. In fact for this specific instance of Peg Solitaire, it performs better than A* algorithm. But the performance quickly deteriorates.

#nodes	Nodes Expanded	time
6	14	0.0007
14	19537	0.89
18	-	-
32	too much	too long

As we can see the performance quickely detereorates as we add more number of nodes, for the full board, it takes more time and expands more number of nodes that a full night of sleep. So, clearly IDS is not scalable for large problems. We shall see that even for a full 32 peg board A* algorithm performs much better and expands lesser number of nodes than a 14 board solved with IDS. So we will move on to A*, which is much more interesting than IDS.

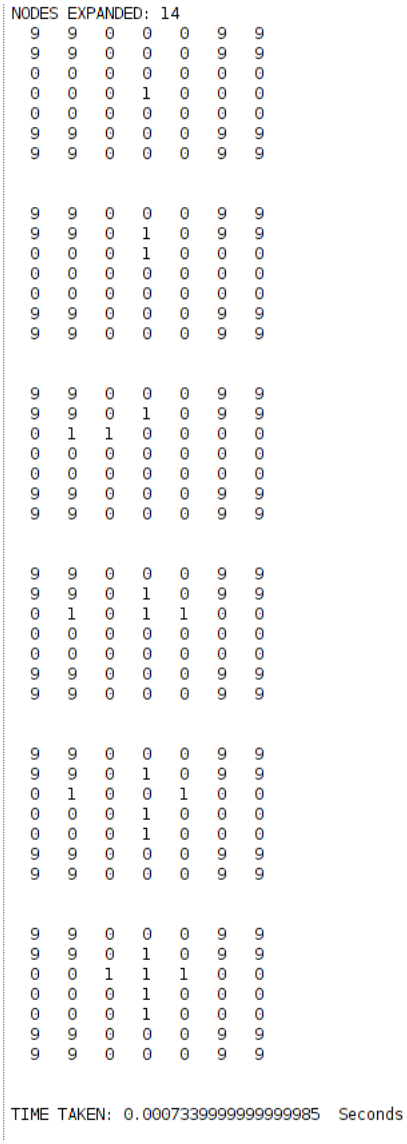

Figure 2: Sample Run of IDS

3 A* Algorithm

3.1 Important Aspects

In A*, we maintain 2 parameters g(n) which tells us the cost of reaching the current board position from the start state and h(n) represents the estimated cost of going from the current state to the final board position.

In A* algorithm, we pick the least value of $f(n) = g(n) + h(n)$. For my implementation, I choose **g(n) = number of removed pegs**. If 5 pegs, have been removed, then $f(n) = 5 + h(n)$. The h(n) is decided by the heuristic used. In general, we would like f(n) in each iteration to remain constant. It is not always possible but by heurisic 3, we reach pretty close to this.

3.2 A* Algorithm Implementation

The algorithm maintains a **priority queue**. The priority queue contains the board state and the cost associated with the state. In each iteration, we extract the least cost board position from the PQ, we add this board position to the **close list**. It checks all possible next board positions from the current state and adds it to the PQ if it is not already in the close list. We also maintain a dictionary called prev that stores the board position from which we came to the present state.

3.3 Heuristic 0: Positional Heuristic - search pruning strategy towards beginning

This is the simplest of heuristics and forms the core of all the heuristics described below. We will see it performs decently well compared to IDS. It is mainly used to prune the search tree towards the beginning of the peg solving problem and prevents us from accessing multiple initial start states.
In this heuristic, for each peg position if it is present, we check its position, if the peg is present towards the 4 corner boxes then we give it a higher cost compared to those those that are present towards the center.
This is the most natural of heuristics, since at each stage, the heuristic will try to bring pegs towards the center and more

```
9   9   1   1   1   9   9
9   9   1   1   1   9   9
1   1   1   1   1   1   1
1   1   1   0   1   1   1
1   1   1   1   1   1   1
9   9   1   1   1   9   9
9   9   1   1   1   9   9


nodes Expanded:7512491
TIME TAKEN: 805.308127  Seconds

Process finished with exit code 0
```

Figure 3: Heuristic 0 - search pruning strategy towards beginning

closer to the goal state. It also prevents pegs from coalescing towards the sides. One advantage of using this heuristic is that towards the beginning of the exploration, this heuristic 0 desperately tries to bring the nodes towards the center, therefore, there is a big drop in the heuristic initially that you will notice in all the algorithms, this is mainly because of this heuristic, around the time the heuristic begins searching for a solution, it should settle at a f(n) value of around 32 - 35. This heuristic drastically reduces our search space, and helps to bring a sort of direction to our problem. Just by using just purely positional heuristic, we can solve the 32 peg problem by expanding 7 million nodes(Figure 3). This is not impressive but compared to IDS it is a huge improvement for a simple positional bias. We will also see this heuristic actually helps us prune the initial search space and thus optimizes the number of nodes we actually have to search, since the heuristic drops rapidly in the beginning, the nodes considered are those that have had the pegs removed from the sides. It avoids searching many different initial configurations. since, our board is symmetrical, there is no need to evaluate multiple start configurations. This is also the case why given any random board position the solution takes slightly more time than it should. but solving it with this heuristic is definitely better than solving it without.

3.4 Search Pruning Strategy towards End

One thing we will notice in the 2 heuristics mentioned below is that while the algortithm is smart enough to deal with initial cases, it really struggles towards the end by going down search paths that are not fruitful, this is mainly because there is usually one node that is spoiling it for the rest of the nodes. This node has no way to be removed because it is in a board position that cannot be removed because there are no pegs that can ever reach adjacent to this node to be removed. That is why when a node is isloated, we perform another check, we check if the 4 positions on the left, right, top and bottom of the peg(if possible), have any pegs at distance 1 hop, 2 hop or 3 hops (a hop involves a skip over another node), if such a hop is not possible and this node is not in the center, then we immediately discard this search path and stop expanding this node.
In figure 4, we can see a rough idea of how this pruning strategy helps, given a peg at position P, it immediately checks if any peg can reach the nodes marked N at distance 1, 2 and 3. If no such node exists then there is no point evaluating this path. This is a way to prune the search tree towards

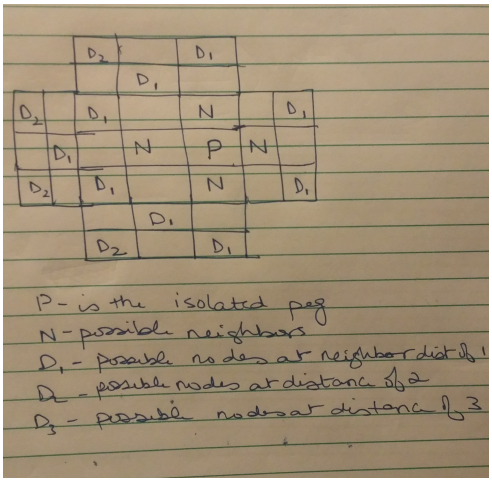

Figure 4: Search Pruning Strategy towards End

the end of the search.
We have seen 2 useful pruning strategies one towards the beginning and one towards the end using these 2 strategies to prune the search space, we develop the following 2 heuristics.

3.5 Heuristic 1: Positional Bais + Isolation of nodes + pruning

This heuristic uses the following heuristics: Other than using positional Bias, it also uses a heuristic where if any node is isolated it has a higher heuristic value than those that have a neighbor. This heuristic immediately gives better results than just using positional heuristic. Whereas just using positional heuristic will bring nodes towards the center, the main problem with the algorithm is that many a time, there are a few isolated nodes that during peg solving are left out. Leaving isolated nodes is no good as in each iteration we reduce the number of nodes, the probability of finding a way to remove this peg becomes harder, so we add a heuristic whereby we penalize the presence of isolated nodes. This is an excellent way to keep the problem compact and prevent stray nodes spoiling the game. Since isolated nodes are costly, A* desparetely tries to remove all such scenarios. This negates most of the drawbacks of heuristic 0, which evaluates around 7 million nodes. This mostly happens because of the isolated node scenario.

We can immediately see that performance is so much better compared to plain positional bias. It takes around 100,000

```
9   9   0   1   1   9   9
9   9   0   1   0   9   9
1   0   1   1   1   0   0
1   1   1   1   1   1   0
1   1   1   1   1   0   0
9   9   1   0   0   9   9
9   9   0   0   0   9   9

nodes Expanded:4242
TIME TAKEN: 0.26008499999999996 Seconds
```

Figure 5: Heuristic 1 for another board

```
9   9   1   1   1   9   9
9   9   1   1   1   9   9
1   1   1   1   1   1   1
1   1   1   0   1   1   1
1   1   1   1   1   1   1
9   9   1   1   1   9   9
9   9   1   1   1   9   9

nodes Expanded:96384
TIME TAKEN: 5.878197 Seconds
```

Figure 6: Heuristic 1 for full board

nodes in the queue but the number of nodes evaluated (number of iterations) is around 14,000 that is a massive improvement. We will analyze the algorithm using a graph of f(n) Vs the iteration# in figure 7. There are a few things we notice in the graph first, there is a sort of equilibrium around 35, that is expected since we are trying to ideally assign a cost of 1 for each move(g(n)) and there are fluctuations (lots of them!), when analyzing the algorithm, I realized the second problem, that most times there are nodes at the side of the board that have no way to get out to the center of the board, this is tricky because unless we move the whole block of 6 pegs at the side together, we risk isolating a node that has no escape route. So that forms a part of our second heuristic and we will see that that performs really well.

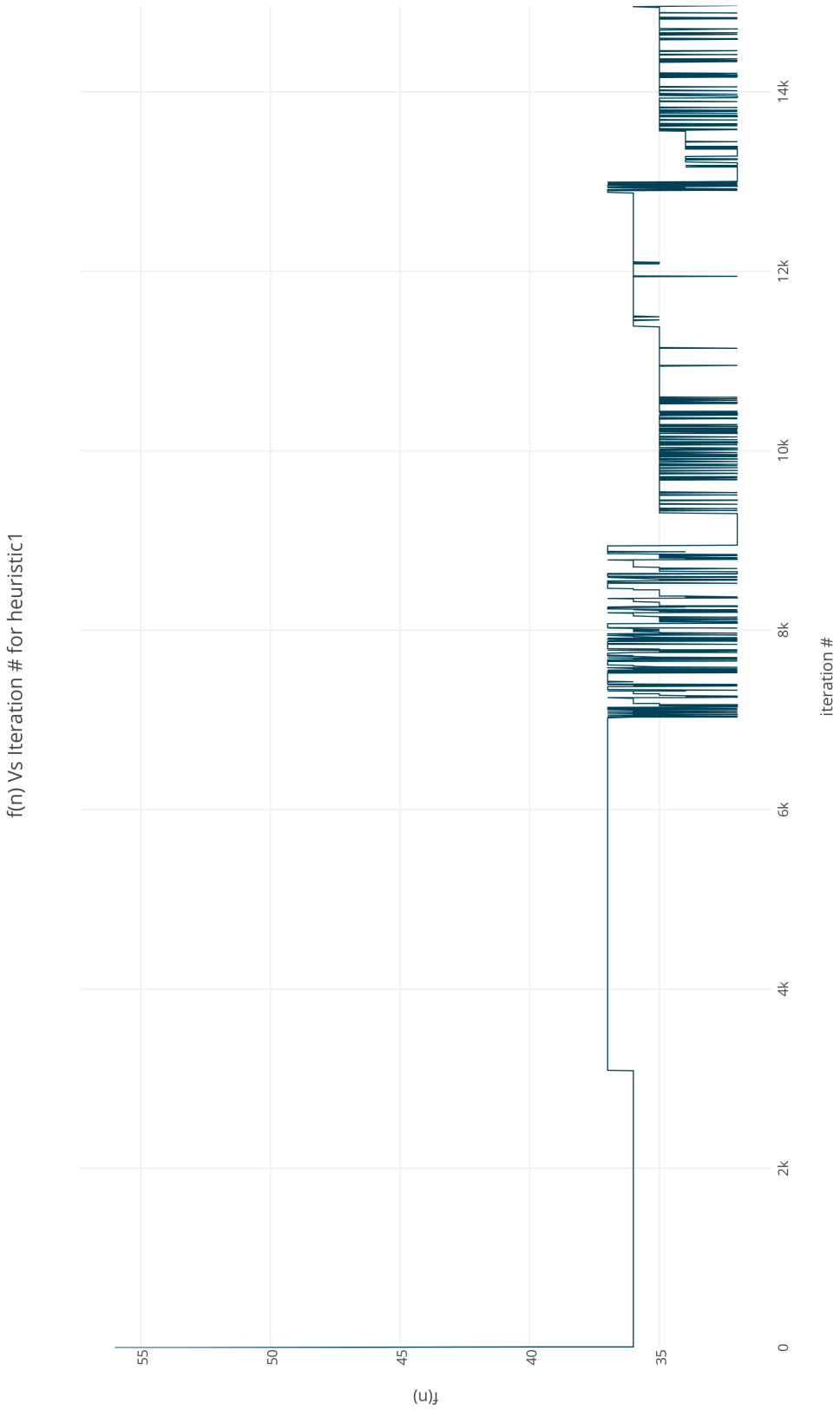

Figure 7: Graph for f(n) Vs iteration # for heuristic 1

3.6 Heuristic 2: Heuristic 1 + preventing nodes with no escape route

In our final heuristic, we do just that we prevent having nodes that have no escape route. I do this by penalizing those nodes at the extreme corners that have no peg in front of them, this solves the problem! the problem of the tall spikes we see in figure 7.
The performance of Heuristic 3 is impressive for a full board. Even more than a partial board. This makes a lot of sense because of our heuristic 0. Heuristic 0 is extremely good in choosing a good start position that removes pegs from the side but given a partial board the effectiveness of this heuristic 0 on heuristic 2 is partially negated. But the performance drop is not a lot. While a full board is solved in 300 or so nodes, a partial board takes around 4000 nodes. This is very impressive.

```
9   9   0   1   1   9   9
9   9   0   1   0   9   9
1   0   1   1   1   0   0
1   1   1   1   1   1   0
1   1   1   1   1   0   0
9   9   1   0   0   9   9
9   9   0   0   0   9   9


nodes Expanded:3771
TIME TAKEN: 0.508682 seconds
```

Figure 8: Heuristic 2 for another board

```
9   9   1   1   1   9   9
9   9   1   1   1   9   9
1   1   1   1   1   1   1
1   1   1   0   1   1   1
1   1   1   1   1   1   1
9   9   1   1   1   9   9
9   9   1   1   1   9   9


nodes Expanded:378
TIME TAKEN: 0.024222999999999998 seconds
```

Figure 9: Heuristic 2 for full board

As we can see in the graph in figure 10. the graph is much more even like how we expect a graph of A* algorithm to look like. Initial hill we see is the work of heuristic 0, that is a sort of preprocess to get the side elements to the center. Then the work of heuristic 2 starts as we can see the spikes have reduced, it is no longer choosing states that have nodes with no escap route and more importantly it maintains its evenness around the 32 mark. This means that the A* algorithm is working properly and g(n) and h(n) are balanced, i.e, as we increase g(n) h(n) reduces hence maintaining a constant f(n) and that is what we need!

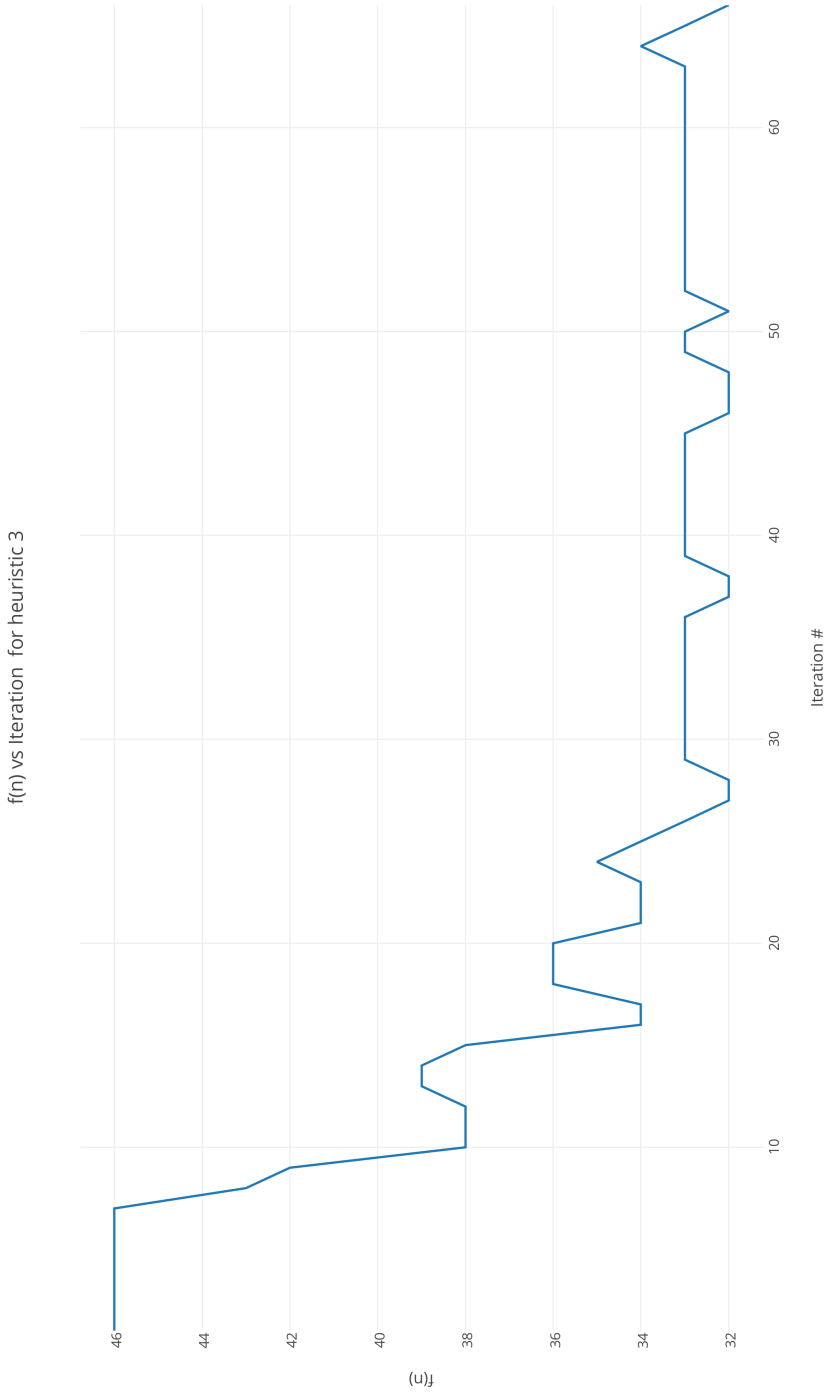

Figure 10: Graph for f(n) Vs iteration # for heuristic 2

4 Conclusion and Findings

I have used A* and IDS to solve the Peg solitaire puzzle. As we have seen IDS is not really a good mechanism when we have a lot of states to process but if the states are less or we have no available choices, IDS is a feasible option compared to other uninformed algorithms. But if we have a heuristic, A* is the algorithm to use. it uses a bit more memory and processing and book keeping but it is a much better performer since it makes informed choices.
In A*, we saw two heuristics to solve our problem, the first one relied on a penalizing isolated nodes where as the second on added onto this heuristic another heuristic for penalizing states where there are pegs which have no exit states. Also, we use two more pruning and state selection strategy, in the first heuristic 0, we make sure that we try to move the nodes towards

the center of the board. This is because in the end the pegs have to move towards the center and this is the best strategy to enforce it. This heuristic helps to point the search tree in the right direction by selecting a more probable direction to proceed.
The second strategy is a pruning strategy to make sure that towards the end when there are a few nodes we might search through a lot of states but it makes no sense to do so, because there are no nodes that can remove that peg. We will save going down those paths. In practice though, the savings are not very great. For heuristic 2, the difference is around 4000 nodes being expanded lesser. For heuristic 3, it is around 10 nodes(of 346), so it is not effective but there is a change and when the nodes expanded are more there is a lot more savings.

4.1 Findings

KEY:
N E - Nodes Expanded
T - Time
H0 - Heristic 0
H1WP - Heuristic 1 Without Pruning
H1P - Heuristic 1 with Pruning
H2WP - Heuristic 2 Without Pruning
H2P - Heuristic 2 With Pruning
T M - Too Much
T L - Too Long
As we can see from the findinds, Heuristic 1 and Heuristic 2 perform much better as there are more pegs. IDS starts out

#Nodes	N E	T	N E	T	N E	T	N E	T	N E	T	N E	T
	IDS		H0		H1WP		H1P		H2WP		H2WP	
5	14	0.0007	22	0.0017	24	0.0021	24	0.0020	26	0.0018	24	0.0019
10	219	0.0111	102	0.0080	128	0.0076	120	0.0105	124	0.0109	124	0.0084
15			2002	0.1492	527	0.0311	500	0.0349	291	0.0193	278	0.0282
20	T M	T L	37401	3.00	4521	0.25	4255	0.28	291	0.01711	278	0.21
25	T M	T L	1586887	129	2253438	123	1885097	139	596711	40	569472	43
32	T M	T L	7512491	805	98924	5.20	96384	5.96	378	0.026	374	0.025

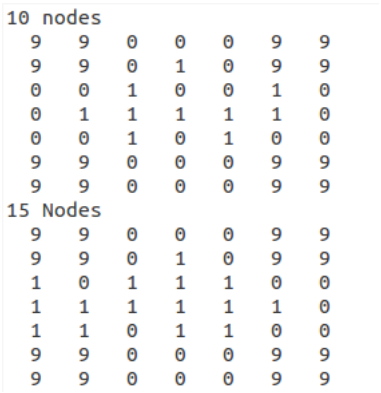

Figure 11: Boards for nodes 10 and 15

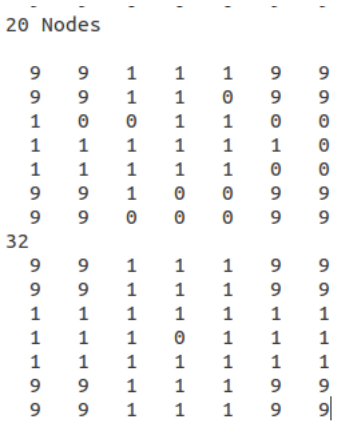

Figure 12: Boards for 20 and 25

strong but really cant compete from 15 nodes. Also, It is not necessary that just because we have more nodes, it will take longer. Heuristic 2 actually performs much better on a full board than on a partial board. This is because when we can start afresh in heuristic 3, it can apply its heuristics much better and can plan it out rather than when it is given a partial board and it might take a few wrong steps.
Pruning is something that has an effect on the time. but the change is not drastic. It works equally well without using pruning but it is an idea I wanted to try.
NOTE: Nodes expanded does not mean that those nodes are visited, just means that those nodes were there in the queue.

That is why the time might be less even if more nodes are visited.
For a full board, heuristic 2 is amazing, it solves it in around 370 moves, now considering in heuristic 0 we had 7 million nodes expanded this is a huge saving.