

CS531 Programming assignment #2

Matt Meyn, Durga Harish, and Jon Dodge

February 6, 2017

1 Abstract

Goal based agents use a search algorithm to find a path to the goal state. The current problem we are trying to solve fits into the search framework (defined goal, deterministic, known environment, atomic representation of states). We explore two different heuristic search algorithms and compare their performance for different initial state. In particular, we present the results using informed search algorithms (A*, Beam Search). Each of these two algorithms are tested with admissible and inadmissible heuristics functions.

2 Introduction

Informed search strategies can outperform uninformed strategies, such as breadth-first search, by selecting actions to pursue based on an evaluation function. The evaluation function specifies the expected cost of the best path going through that node. Since it is generally not possible to evaluate the cost of a path in advance, this evaluation function is usually an estimate or heuristic.

Our goal in this project is to find useful heuristic functions (both admissible and inadmissible) for a variation on the Towers of Hanoi. Our problem, called Towers of Corvallis, is like the original game except that any disc can be placed on any other, regardless of size. In this problem, the goal is always to reach the state in which all discs are on the first of the three pegs, with the lowest-numbered disc at the bottom, the rest of the discs in ascending order up to the largest number at the top.

We use A* and Beam Search with beam widths 5, 10, 15, 20, 25, 50 and 100. Each of these searches is performed with two different heuristic functions, one admissible and one inadmissible.

3 Algorithms-Heuristic Functions

We tested two functions as heuristics: one admissible and one inadmissible.

Heuristic Function 1 considers the first disc on the Goal Peg (Peg A) that is out of solution order. The function adds the number of discs above to the total cost of this state. This is the minimum possible actual cost of a solution path through this state, because at least one move is needed to move each disc above the “bad” disc to another peg, and at least one more move to move that disc back onto the Goal Peg. For the other two pegs, the desired order is in reverse so that the discs can be easily placed onto the Goal Peg when possible. All discs out of desired order on Non-Goal Pegs (Pegs B&C) are assigned a cost of 1, because at least one move will be necessary to move them to the Goal Peg. Figure 1 illustrates an example of how the heuristic function works.

Because our argument for the number of necessary moves is very bare minimum, our heuristic cannot overestimate the cost to the goal. Additionally, the formulation above corresponds to the relaxed problem where we can place a disc anywhere on the destination peg, but keep the restriction of only removing from the top of the source peg. Last, we also implemented a weighted version of this function, where weights were in the range $\{0,1\}$. If the function described above is admissible, then the weighted version must be since multiplying by weights in that range will not increase the estimate, and therefore must maintain admissibility.

```

NumDiscsOutOfPlaceAllPegs( SearchState ) returns Cost
  Cost ← 0
  PegA ← SearchState.Pegs[1]
  for each Disc in PegA reverse order
    DiscsAbove ← PegANum Discs - Disc.Position
    DesiredDiscValue ← MAX_DISC_VALUE - Disc.Position
    if Disc.Value != DesiredDiscValue
      Cost ← DiscsAbove

  for each remaining Peg
    for each Disc in Peg
      if Disc.Value != Disc.Position
        Cost ← Cost + 1
return Cost

```

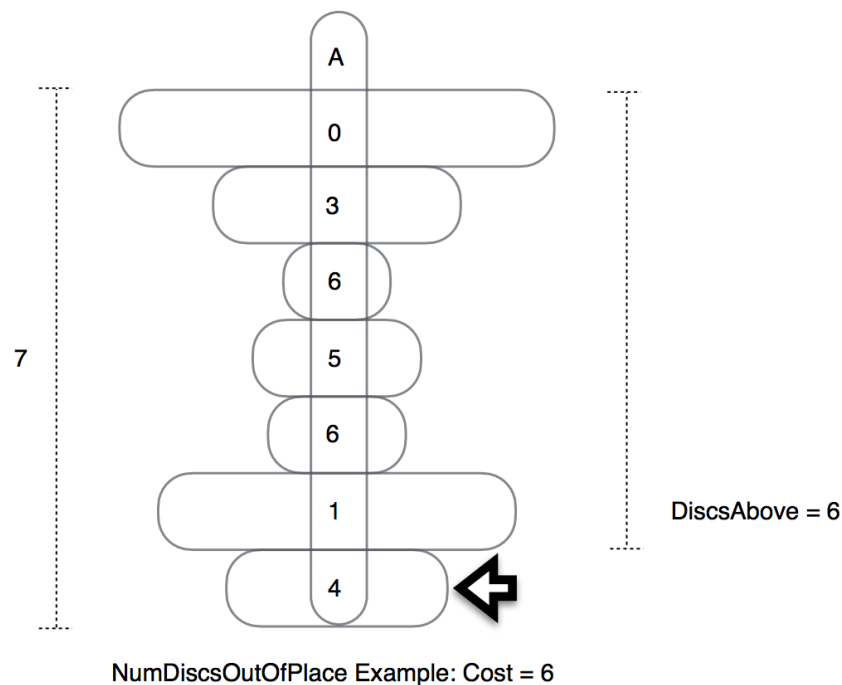


Figure 1: FIXME Matt: can you caption this image? im not sure I follow

Heuristic Function 2 Is similar to our first heuristic, but we consider the size of the discs as well. Namely, if heavy discs are above a wrongfully placed disc, we would like that state to pay a heavy price. Thus, we scale based on the difference between the disc we expect at an index vs the disc that is placed there. Note that this may break admissibility, but seems to model the physics of the problem reasonably well. Figure 2 goes into some further detail on how this heuristic function works

```

ManhattanDistanceAllPeg( SearchState )
  Cost ← 0
  PegA ← SearchState.FirstPeg
  IsInOrder ← True

```

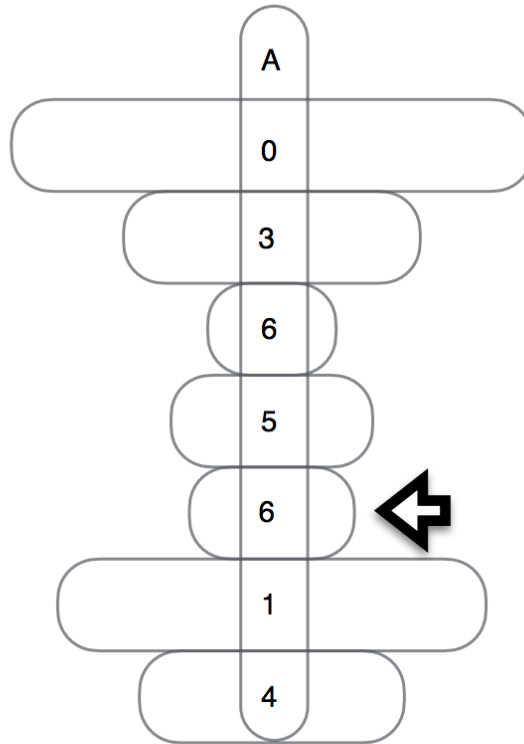
```

for each Disc in PegA
  DesiredDiscValue <- MAX_DISC_SIZE - Disc.Position
  IsInOrder <- IsInOrder AND PegA == DesiredDiscValue
  if NOT IsInOrder
    Cost <- Cost + 2 + Abs( Disc.Value - DesiredDiscValue )

for each remaining Peg
  for each Disc in Peg
    Cost <- Cost + 1
    for each OtherDisc above Disc
      delta <- Disc.Value - OtherDisc.Value
      if delta > 0
        Cost <- Cost + delta

return Cost

```



ManhattanDistance Example: Cost for this disc is $\text{abs}(6 - 2) = 4$.

Figure 2: FIXME with Content

4 Results

We carried out the experiment with two search algorithms(A-Star, Beam Search). As mentioned previously, we used two heuristic functions one admissible and one in-admissible. Table 1 illustrates one view of our

timing results.

Table 1: Table showing the results for A-Star search algorithm. Failures column indicate number of times the function failed. We ran this data giving a mere 500 iterations, since our inadmissible heuristic was easily solving it in an order of magnitude fewer iterations

Heuristic Fn	Problem Size	Failures	Solution Depth	Nodes Expanded	CPU Time
In Admissible	3	0	5	6	0.002828
In Admissible	4	0	7	8	0.006125
In Admissible	5	0	11	12	0.012716
In Admissible	6	0	14	16	0.021471
In Admissible	7	0	16	18	0.029587
In Admissible	8	0	19	25	0.050999
In Admissible	9	0	23	31	0.077606
In Admissible	10	0	26	40	0.119632
Admissible	3	0	5	15	0.007891
Admissible	4	0	7	57	0.054209
Admissible	5	6	7	163	0.324806
Admissible	6	18	1	29	0.080155

5 Discussion

Question 1. Show an example solution sequence for each algorithm for the largest size you tested in the following format: We tested for disc size of 10 and peg size of 3. Additional solution sequences may be found in our attached file `solutions-10-hb.txt`, here is one of the shorter ones:

```

d: 23 h: 0 [[9, 8, 7, 6, 5, 4, 3, 2, 1, 0], [], []]
d: 22 h: 1 [[9, 8, 7, 6, 5, 4, 3, 2, 1], [], [0]]
d: 21 h: 2 [[9, 8, 7, 6, 5, 4, 3, 2], [], [0, 1]]
d: 20 h: 3 [[9, 8, 7, 6, 5, 4, 3], [2], [0, 1]]
d: 19 h: 4 [[9, 8, 7, 6, 5, 4, 3], [2, 1], [0]]
d: 18 h: 5 [[9, 8, 7, 6, 5, 4], [2, 1], [0, 3]]
d: 17 h: 6 [[9, 8, 7, 6, 5], [2, 1, 4], [0, 3]]
d: 16 h: 7 [[9, 8, 7, 6, 5], [2, 1, 4, 3], [0]]
d: 15 h: 8 [[9, 8, 7, 6], [2, 1, 4, 3], [0, 5]]
d: 14 h: 9 [[9, 8, 7], [2, 1, 4, 3, 6], [0, 5]]
d: 13 h: 10 [[9, 8], [2, 1, 4, 3, 6], [0, 5, 7]]
d: 12 h: 11 [[9], [2, 1, 4, 3, 6], [0, 5, 7, 8]]
d: 11 h: 12 [[], [2, 1, 4, 3, 6], [0, 5, 7, 8, 9]]
d: 10 h: 16 [[6], [2, 1, 4, 3], [0, 5, 7, 8, 9]]
d: 9 h: 21 [[6, 3], [2, 1, 4], [0, 5, 7, 8, 9]]
d: 8 h: 24 [[6, 3, 9], [2, 1, 4], [0, 5, 7, 8]]
d: 7 h: 27 [[6, 3, 9, 4], [2, 1], [0, 5, 7, 8]]
d: 6 h: 31 [[6, 3, 9, 4, 8], [2, 1], [0, 5, 7]]
d: 5 h: 34 [[6, 3, 9, 4, 8, 1], [2], [0, 5, 7]]
d: 4 h: 36 [[6, 3, 9, 4, 8, 1, 2], [], [0, 5, 7]]
d: 3 h: 36 [[6, 3, 9, 4, 8, 1, 2], [7], [0, 5]]
d: 2 h: 40 [[6, 3, 9, 4, 8, 1, 2, 5], [7], [0]]
d: 1 h: 42 [[6, 3, 9, 4, 8, 1, 2, 5, 0], [7], []]
d: 0 h: 50 [[6, 3, 9, 4, 8, 1, 2, 5, 0, 7], [], []]

```

Question 2. How did the search time and the solution quality vary with the beam width? Is there a beam width that gives the best tradeoff for the two heuristics?

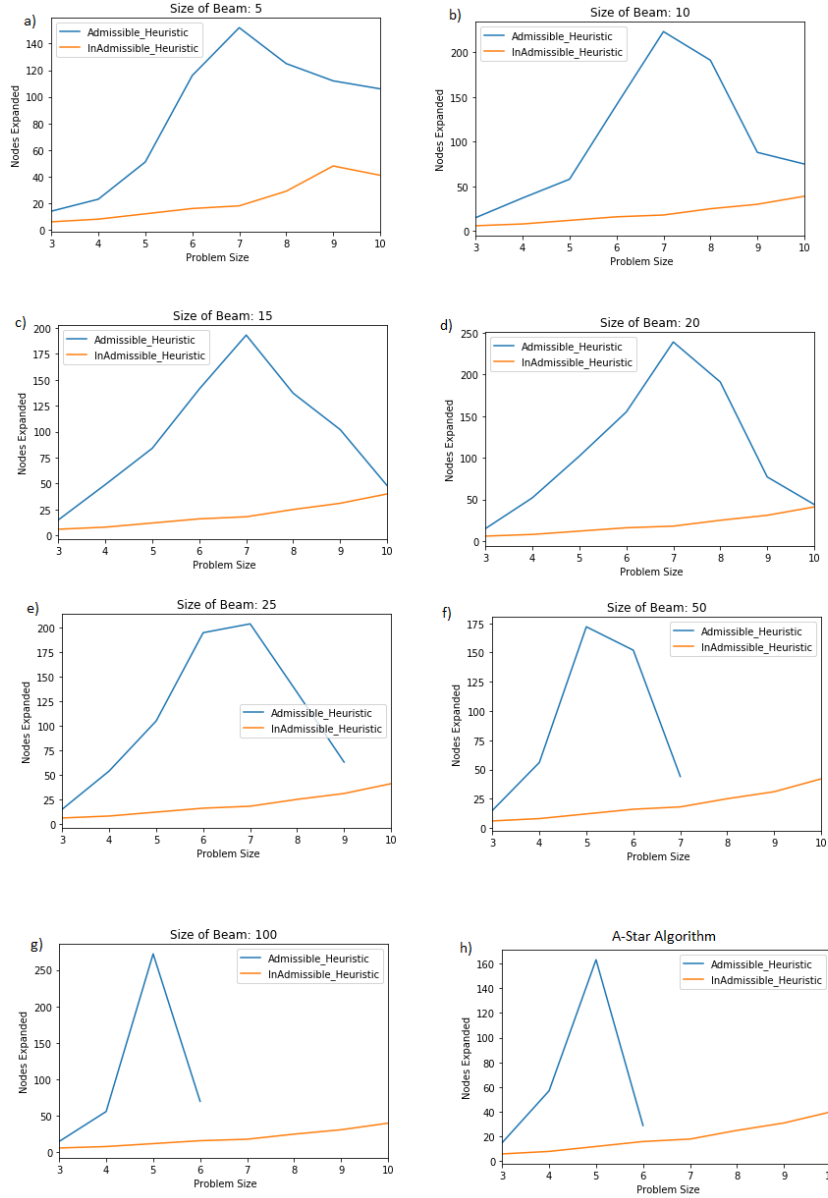


Figure 3: FIXME with Content

Up to a beam width of about 20, the search time increased only modestly. At 20 it increased significantly. With the inadmissible heuristic, the solution quality did not vary, but with the admissible heuristic, there was some improvement of solution quality as beam width increased. It is also notable that with the admissible heuristic, we started having searches take >500 iterations as beam width became large.

Question 3. Is there a clear preference ordering among the heuristics you tested considering the number of nodes searched and the total CPU time taken to solve the problems for the two algorithms?

With respect to both the number of nodes searched and the total CPU time needed, our inadmissible heuristic substantially outperformed our admissible heuristic.

The number of nodes expanded by A* and beam search (all beam sizes) using the admissible heuristic appear to have complexity exponential in the problem size. For the same beam widths, A* and beam search

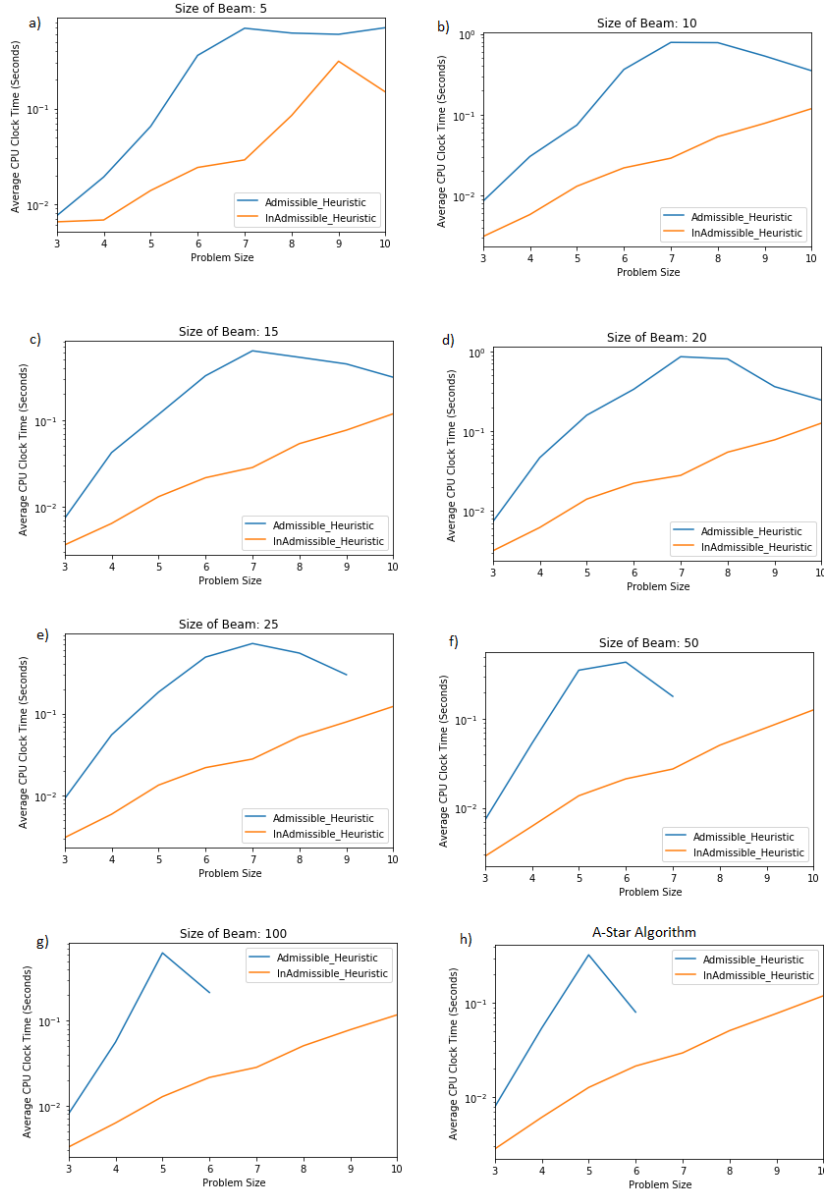


Figure 4: FIXME with Content

using the inadmissible heuristic show approximately linear complexity.

The CPU time required to find a solution shows a less dramatic but still apparent difference in time complexity.

Question 4. Can a small sacrifice in optimality give a large reduction in the number of nodes expanded? What about CPU time? A* tree search using an admissible heuristic guarantees an optimal solution, but this guarantee comes at a cost. The requirement of admissibility imposes constraints on the choices we can make for how to estimate the cost of a given state. It is clearly easier to find heuristics that lead more quickly to a goal state if those constraints are removed.

As observed above in (3), our inadmissible heuristic function enabled all searches to expand a number of nodes approximately linear in the problem size, whereas the complexity associated with the admissible

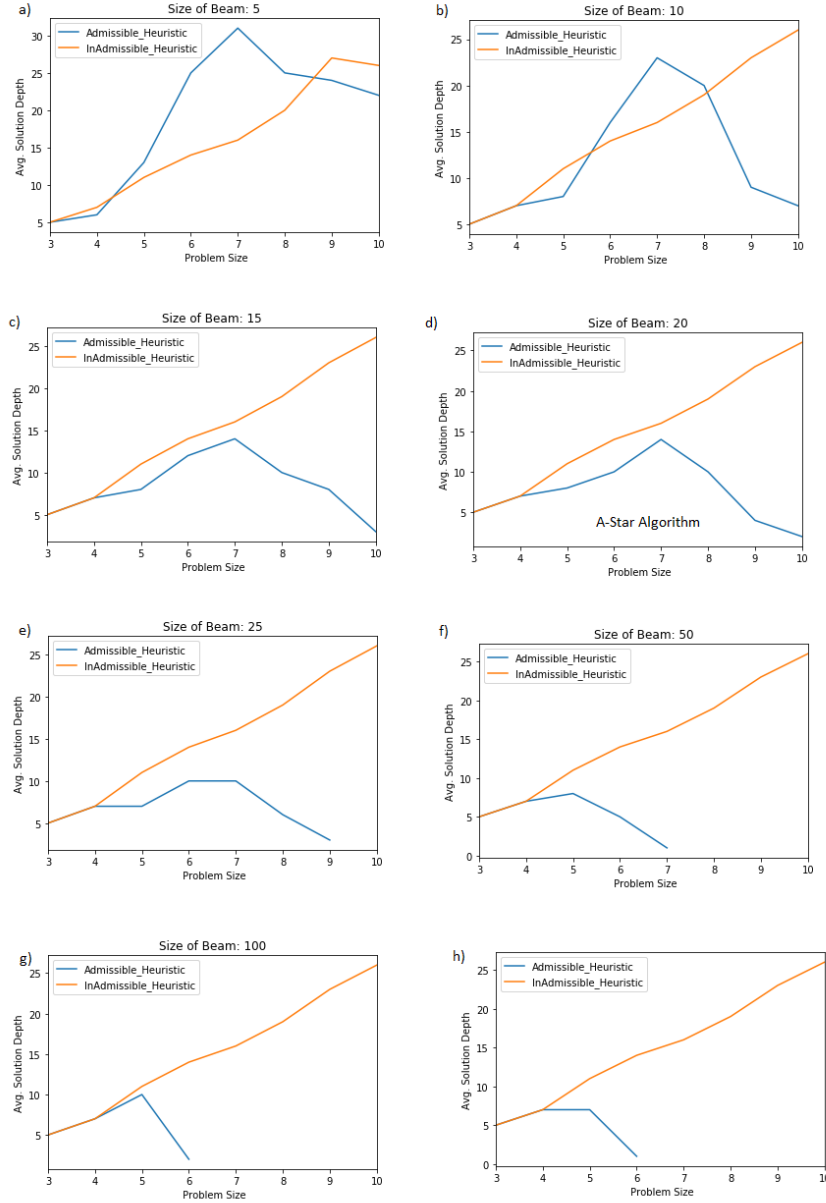


Figure 5: FIXME with Content

heuristic appears to be exponential.

Now, to discuss how much optimality is lost, we have provided 2 files, `solutions-6-aa.txt` and `solutions-6-hb.txt`, which contain the results of applying A* search with an admissible heuristic, and our inadmissible heuristic using a beam search, respectively. Table 2 summarizes the comparison of the solutions found in these two files. We found that solution quality did not degrade much at lower problem sizes, though the time cost of obtaining an optimal solution at solution size 10 was daunting, so we do not have that data.

Question 5. How did you come up with your heuristic evaluation functions? Our admissible heuristic function, Function 1, is based on the simplified problem that if a disc on Peg A is out of solution order, every disc above it must be moved. Similarly, every disc on Pegs B and C that is not sorted in reverse order also represents an extra move to get that disc out of the way.

Table 2: Table showing quality of nonoptimal solutions for problems of size 6. Optimal solution was obtained via A* search (first line) and an admissible heuristic. The non optimal search uses beam search and our inadmissible heuristic (second line). The third line shows the difference for easier comparison. Of the 20 inputs, there were 10 optimal solutions, and 7 solutions within 1 solution depth.

15	13	14	12	15	12	11	14	15	14	12	14	11	13	14	15	12	12	15	14
16	16	14	13	16	12	11	14	16	14	12	18	10	13	15	18	12	12	16	14
1	3		1	1				1			4	1		1	3			1	

Function 2, our inadmissible heuristic, was inspired by the “Manhattan Distance” heuristic. Each disc on Peg A from position 0 up that is in order is already at the “destination.” After a disc is found out of order, each disc incurs an extra cost of the absolute value of the delta between the desired disc “size” and actual disc “size.” This lets our search algorithm know that if Peg A is “approximately” in order, this is closer to the solution.

On the other pegs, we decided to sum the deltas in “size” between each disc and each lower-numbered disc above it, adding the sum of all of these to the total cost, because states with Pegs B and C stacked in ascending order from bottom to top should be favored.

Early on, we tested some heuristics which did not examine the non-goal pegs and found those did not work very well.

Question 6. How do the two algorithms compare in the amount of search involved and the cpu-time The two algorithms, A* and Beam Search, are related in that A* is Beam Search with a beam of unbounded width. Naturally, then, A* does expand more nodes and take more time than a beam search with width, say, 15. But when comparing to a beam width of 100, there does not seem to be a significant difference for this problem.

Question 7. Do you think that either of these algorithms scale to even larger problems? What is the largest problem you could solve with the best algorithm+heuristic combination? Report the wall-clock time, CPU-time, and the number of nodes searched.

Our admissible heuristic would not scale well to larger problems because it increases the number of nodes expanded exponentially. The inadmissible heuristic also is more tractable in terms of CPU time.

The curves for the admissible heuristic also increase more steeply as the beam width increases toward infinite width.

We tested some problems of size 11, but found that it was not worth waiting around for them to finish. It is possible that imperfect data structure choice limited our performance for this test, since our implementation does not use a hash table to store the search frontier or explored region.

Question 8. Is there any tradeoff between how good a heuristic is in cutting down the number of nodes and how long it took to compute? Can you quantify it?

In our case yes, since our admissible heuristic runs as $O(n)$, where n is the number of discs, while our inadmissible heuristic runs as $O(n^2)$.

Question 9. Is there anything else you found that is of interest? Our implementation made it pretty easy to increase the number of pegs used in the test. We figured that the results would be that adding pegs would increase the search time, since the branching factor increases quite sharply. However, as the number of pegs approaches the number of discs, a trivial solution becomes available. Our inadmissible heuristic did not seem able to reliably find that trivial solution.

First, here is the data table associated with those tests, and it will be followed by a solution that our program found with a high peg count

```
d: 55 h: 0.0 [[9, 8, 7, 6, 5, 4, 3, 2, 1, 0], [], [], [], [], [], [], [], [], []]
d: 54 h: 0.0 [[9, 8, 7, 6, 5, 4, 3, 2, 1], [0], [], [], [], [], [], [], [], []]
d: 53 h: 0.0 [[9, 8, 7, 6, 5, 4, 3, 2], [0, 1], [], [], [], [], [], [], [], []]
d: 52 h: 0.0 [[9, 8, 7, 6, 5, 4, 3], [0, 1, 2], [], [], [], [], [], [], [], []]
d: 51 h: 0.0 [[9, 8, 7, 6, 5, 4], [0, 1, 2, 3], [], [], [], [], [], [], [], []]
```


Table 3: Table showing the results for varying peg count’s effect on number of nodes expanded. Problems of size 9 discs were examined in this study (5x problems, one per row)

Peg count	3	4	5	6	7	8	9	10
	90	131	170	202	409	991	1117	1251
	95	135	175	207	866	461	1116	1249
	239	555	428	201	730	796	841	704
	708	3001	3227	3951	5732	2558	3340	2818
	92	141	200	254	307	364	427	496

d: 50 h: 0.0 [[9, 8, 7, 6, 5], [0, 1, 2, 3, 4], [], [], [], [], [], [], [], []]
d: 49 h: 2.0 [[9, 8, 7, 6], [0, 1, 2, 3, 4], [5], [], [], [], [], [], [], []]
d: 48 h: 2.4 [[9, 8, 7, 6, 4], [0, 1, 2, 3], [5], [], [], [], [], [], [], []]
d: 47 h: 1.0 [[9, 8, 7, 6, 4, 5], [0, 1, 2, 3], [], [], [], [], [], [], [], []]
d: 46 h: 1.42857142857 [[9, 8, 7, 6, 4, 5, 3], [0, 1, 2], [], [], [], [], [], [], [], []]
d: 45 h: 3.42857142857 [[9, 8, 7, 6, 4, 5, 3], [0, 1], [], [2], [], [], [], [], [], []]
d: 44 h: 3.5 [[9, 8, 7, 6, 4, 5], [0, 1], [], [2, 3], [], [], [], [], [], []]
d: 43 h: 3.56666666667 [[9, 8, 7, 6, 4], [0, 1, 5], [], [2, 3], [], [], [], [], [], []]
d: 42 h: 3.15 [[9, 8, 7, 6, 4], [0, 1, 5, 3], [], [2], [], [], [], [], [], []]
d: 41 h: 2.8 [[9, 8, 7, 6], [0, 1, 5, 3, 4], [], [2], [], [], [], [], [], []]
d: 40 h: 1.2 [[9, 8, 7, 6, 2], [0, 1, 5, 3, 4], [], [], [], [], [], [], [], []]
d: 39 h: 1.41666666667 [[9, 8, 7, 6, 2, 4], [0, 1, 5, 3], [], [], [], [], [], [], [], []]
d: 38 h: 3.33333333333 [[9, 8, 7, 6, 2, 4], [0, 1, 5], [], [3], [], [], [], [], [], []]
d: 37 h: 3.14285714286 [[9, 8, 7, 6, 2, 4, 5], [0, 1], [], [3], [], [], [], [], [], []]
d: 38 h: 1.75 [[9, 8, 7, 6, 2, 4, 5, 3], [0, 1], [], [], [], [], [], [], [], []]
d: 37 h: 2.22222222222 [[9, 8, 7, 6, 2, 4, 5, 3, 1], [0], [], [], [], [], [], [], [], []]
d: 36 h: 2.22222222222 [[9, 8, 7, 6, 2, 4, 5, 3, 1], [], [0], [], [], [], [], [], [], []]
d: 35 h: 1.75 [[9, 8, 7, 6, 2, 4, 5, 3], [], [0, 1], [], [], [], [], [], [], []]
d: 34 h: 3.14285714286 [[9, 8, 7, 6, 2, 4, 5], [], [0, 1], [3], [], [], [], [], [], []]
d: 33 h: 3.16666666667 [[9, 8, 7, 6, 2, 4], [], [0, 1], [3, 5], [], [], [], [], [], []]
d: 32 h: 3.4 [[9, 8, 7, 6, 2], [], [0, 1], [3, 5, 4], [], [], [], [], [], []]
d: 31 h: 3.0 [[9, 8, 7, 6], [], [0, 1, 2], [3, 5, 4], [], [], [], [], [], []]
d: 30 h: 2.9 [[9, 8, 7, 6, 4], [], [0, 1, 2], [3, 5], [], [], [], [], [], []]
d: 29 h: 2.9 [[9, 8, 7, 6, 4], [], [0, 1, 2, 5], [3], [], [], [], [], [], []]
d: 28 h: 2.6 [[9, 8, 7, 6], [], [0, 1, 2, 5, 4], [3], [], [], [], [], [], []]
d: 27 h: 1.0 [[9, 8, 7, 6], [], [0, 1, 2, 5, 4, 3], [], [], [], [], [], []]
d: 26 h: 3.0 [[9, 8, 7], [], [0, 1, 2, 5, 4, 3], [6], [], [], [], [], [], []]
d: 25 h: 3.1 [[9, 8, 7], [], [0, 1, 2, 5, 4], [6, 3], [], [], [], [], [], []]
d: 24 h: 3.5 [[9, 8, 7], [], [0, 1, 2, 5], [6, 3, 4], [], [], [], [], [], []]
d: 23 h: 3.5 [[9, 8, 7, 5], [], [0, 1, 2], [6, 3, 4], [], [], [], [], [], []]
d: 22 h: 4.2 [[9, 8, 7, 5, 2], [], [0, 1], [6, 3, 4], [], [], [], [], [], []]
d: 21 h: 4.7 [[9, 8, 7, 5, 2], [], [0], [6, 3, 4, 1], [], [], [], [], [], []]
d: 20 h: 4.5 [[9, 8, 7, 5], [], [0], [6, 3, 4, 1, 2], [], [], [], [], [], []]
d: 19 h: 4.16666666667 [[9, 8, 7], [], [0], [6, 3, 4, 1, 2, 5], [], [], [], [], [], []]
d: 18 h: 5.16666666667 [[9, 8], [], [0, 7], [6, 3, 4, 1, 2, 5], [], [], [], [], [], []]
d: 17 h: 5.66666666667 [[9, 8], [], [0, 7, 5], [6, 3, 4, 1, 2], [], [], [], [], [], []]
d: 16 h: 5.75 [[9, 8], [], [0, 7, 5, 2], [6, 3, 4, 1], [], [], [], [], [], []]
d: 15 h: 5.8 [[9, 8], [], [0, 7, 5, 2, 1], [6, 3, 4], [], [], [], [], [], []]
d: 14 h: 6.33333333333 [[9], [], [0, 7, 5, 2, 1, 8], [6, 3, 4], [], [], [], [], [], []]
d: 13 h: 6.35714285714 [[9], [], [0, 7, 5, 2, 1, 8, 4], [6, 3], [], [], [], [], [], []]
d: 12 h: 6.875 [[], [], [0, 7, 5, 2, 1, 8, 4, 9], [6, 3], [], [], [], [], [], []]
d: 11 h: 6.88888888889 [[], [], [0, 7, 5, 2, 1, 8, 4, 9, 3], [6], [], [], [], [], [], []]
d: 10 h: 6.88888888889 [[6], [], [0, 7, 5, 2, 1, 8, 4, 9, 3], [], [], [], [], [], [], []]

d: 9 h: 7.375 [[6, 3], [], [0, 7, 5, 2, 1, 8, 4, 9], [], [], [], [], [], [], []]
d: 8 h: 7.85714285714 [[6, 3, 9], [], [0, 7, 5, 2, 1, 8, 4], [], [], [], [], [], [], []]
d: 7 h: 8.33333333333 [[6, 3, 9, 4], [], [0, 7, 5, 2, 1, 8], [], [], [], [], [], [], []]
d: 6 h: 8.8 [[6, 3, 9, 4, 8], [], [0, 7, 5, 2, 1], [], [], [], [], [], [], []]
d: 5 h: 9.25 [[6, 3, 9, 4, 8, 1], [], [0, 7, 5, 2], [], [], [], [], [], [], []]
d: 4 h: 9.66666666667 [[6, 3, 9, 4, 8, 1, 2], [], [0, 7, 5], [], [], [], [], [], [], []]
d: 3 h: 10.0 [[6, 3, 9, 4, 8, 1, 2, 5], [], [0, 7], [], [], [], [], [], [], []]
d: 2 h: 11.0 [[6, 3, 9, 4, 8, 1, 2, 5], [7], [0], [], [], [], [], [], [], []]
d: 1 h: 12.0 [[6, 3, 9, 4, 8, 1, 2, 5, 0], [7], [], [], [], [], [], [], [], []]
d: 0 h: 11.0 [[6, 3, 9, 4, 8, 1, 2, 5, 0, 7], [], [], [], [], [], [], [], [], []]