CS531 Programming Assignment 2: Towers of Corvallis

Michael Lam, Xu Hu
EECS, Oregon State University

October 25, 2012

Abstract

In this assignment we design, implement and discuss two different informed search algorithms and heuristics to solve the Towers of Corvallis, which is a variation of Towers of Hanoi.

1 Introduction

The Towers of Corvallis puzzle is a variation on the Towers of Hanoi puzzle. While similarly consisting of 3 pegs and n disks, the Corvallis variation allows any disk to go on top of any other disk. The goal is to find the smallest number of moves in getting from an initial state to the goal state, which is defined as the order 9876543210 on peg A for 10 disks and similarly for fewer disks.

We implement two informed search algorithms: A* and RBFS (recursive best-first search). As informed searches, we also implement two heuristics, one admissible and one non-admissible. We ended up implementing a second non-admissible heuristic to demonstrate potential for more computational efficiency. For each algorithm and heuristic function, we evaluate the performance by testing across different number of disks and different initial states.

We discovered that A* outperformed RBFS in terms of the number of nodes expanded and cpu time in general. This was especially noticeable in one of the non-admissible heuristics that we designed. The non-admissible heuristics outperformed the admissible heuristic with exception to optimality. We also discovered that the heuristic function yields the same optimal solution lengths for each algorithm.

2 A* Search

We implement the A* search algorithm for finding the paths between initial states to a given goal state in the problem of Tower of Corvallis. It is a variant of the uniform cost search by using a heuristic function for each node in addition to path costs. It is summarized in algorithm 1.

Algorithm 1 A* Search

```
exploredSet = $\emptyset$
frontier = [initialPath]
while number(explored) $<$ NMAX do
  if frontier == $\emptyset$ then
    return FALSE
  end if
  path = frontier.pop()
  state = path[0]
  exploredSet.add(state)
  if state == goalState then
    return path
  end if
  for action in state.validActions() do
    for newState in action.results() do
      newPath = path + newState
      if ismember(frontier,newPath) == FALSE then
        frontier.push(newPath)
      end if
    end for
  end for
end while
```

For implementing the frontier, we use the priority queue, which is actually a heap data structure. We use a callback function $f(state) = g(state) + h(state)$ as the priority, where $g(state)$ is the length of the path and $h(state)$

is the heuristic for estimating the distance between the current state to the goal state. We will analyze several admissble and non-admissble heuristics in section 4.

3 Recursive Best-First Search

Recursive best-first search or RBFS works by storing an f-limit variable to keep track of the f-value of the best alternative path available from any ancestor node to the current node. The algorithm uses the f-limit to decide which subtree of the problem tree to explore by considering the current path and the best alternative path. In order to keep the search functional, RBFS also updates the f-values of each node during the recursion unwinding. This allows RBFS to consider the forgotten subtree in the future.

The advantage of RBFS over A* is that RBFS uses less memory, linear space. Whereas A* stores all of its explored nodes, RBFS will only keep "relevant" nodes in memory. However, the disadvantage of RBFS over A* is that RBFS could expand more nodes than A* due to redundancy. Since RBFS does not store all nodes explored, it can re-expand the same nodes and thereby increasing computation time.

The pseudocode is listed in algorithm 2.

4 Experiments

4.1 Heuristics

We designed many heuristics for both A* and RBFS. Here we show an admissible one and two non-admissible ones for demonstration. Other heuristics will be discussed in section 5.

We designed an admissible heuristic by reasoning about simulating the game. For the second and third peg, disks on there should intuitively be moved to the first peg to get to the goal state. An admissible estimation of steps is the amount of disks in these pegs. If a disk at the bottom of the first peg is not the right one, it should be moved out first and then move to its ideal position, which takes at least $2k$ steps, where k is the number of disks on top of it. We show the heuristic below:

$$h1(s) = I(s(1,0) == goal(1,0)) \cdot 2 \cdot num(s(1)) + num(s(2)) + num(s(3))$$

Algorithm 2 RBFS Search

```
function RBFS(state, f-limit):
if state is goal state then
  return solution
end if
successor = all children of state
if successors is empty then
  return failure
else
  for s in successors do
    s.$f$ = $max$(s.$g$ + s.$h$, state.$f$)
  end for
  while true do
    best = lowest f-value node in successors
    if best.$f$ > f-limit then
      return failure, best.$f$
    end if
    alternative = second best f-value of any node in successors
    result, best.$f$ = RBFS(best, $min$(f-limit, alternative))
    if result is not failure then
      return result
    end if
  end while
end if
```

Here $I(\cdot)$ is indication function, and $num(\cdot)$ is the function for counting disks.

A non-trivial non-admissible heuristic is designed in a similar way but taking ordering into account:

$$h2(s) = (\sum_i |s(1,i) - i|) + numberDisks(s(2)) + numberDisks(s(3))$$

Here s(i,j) is the *jth* disk in peg i (pegs indexed 0 to 2 from left to right here). The intuition is that for a disk k in peg 1, it has to take $|k - i|$ steps to get to the *kth* place by moving away or filling out disks on peg 0, where i is its current location on the peg. For example, if the $9th$ disk is on the bottom, it has to take 9 minus 0 steps to get to the top. For disks on peg 2 and peg 3, they have to be moved from the current location to peg 1 in at least 1 step, which is the same as the admissible heuristic. However this is not an admissible heuristic because there are multiple counts for disks on peg 0.

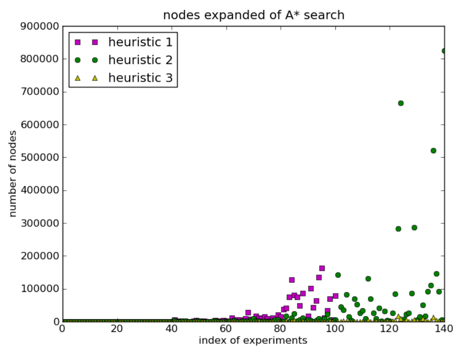
For a fast non-admissible heuristic, we simplily enlarge the former heuristic by a factor of 2. We will see that this relaxation makes the increasing of computation and expanded nodes become linear in the problem size ranging from $4 - 10$.

$$h3(s) = 2 \cdot [(\sum_i |s(1,i) - i|) + numberDisks(s(2)) + numberDisks(s(3))]$$

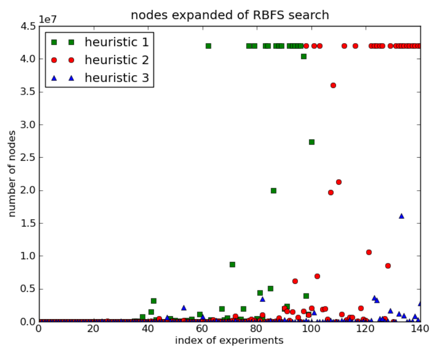
4.2 Results

We tested the three heuristics on all 140 examples for the problem size ranging from 4 to 10 with each size having 20 examples. We couldn't finish the admissible heuristic on the whole data with a ten minute max execution time (which is represented as an NMAX constant, the maximum number of nodes to expand) per problem. For a large size like 9 or 10, a few examples can take more than one hour. For RBFS, the number of expanded nodes grew rapidly and failed the NMAX constraint. We show the results on disk sizes 4 to 8 for the admissible heuristic (heuristic 1) and the whole data for non-admissible heuristics (heuristic 2 and 3).

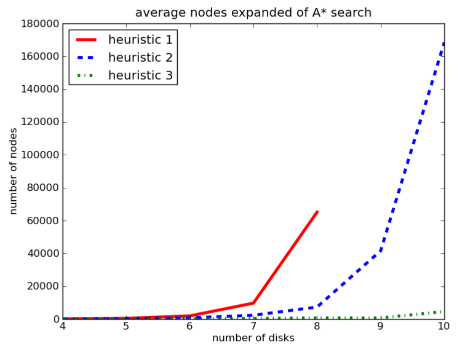
The first and second experiments are comparisons of the number of expanded nodes and cpu time between different heurisitics, which are shown in figure 1 and figure 2 respectively. We can see that both nodes and cpu time grow up expenentially. However, a well-designed heuristic function can



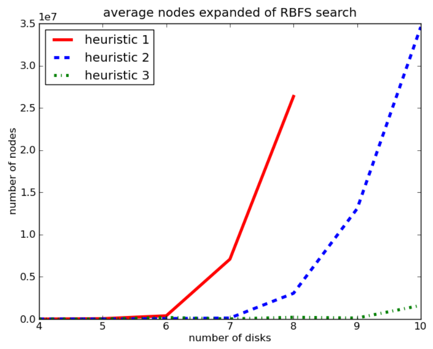
(a) A* nodes: all 140 experiments



(b) RBFS nodes: all 140 experiments



(c) A* average searched nodes

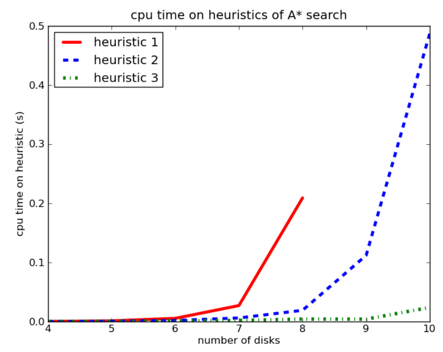


(d) RBFS average searched nodes

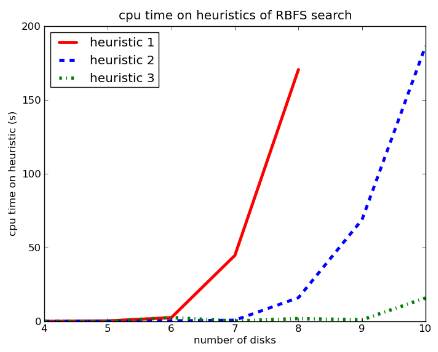
Figure 1: Plots for the number of nodes searched against the problem size for each algorithm and heuristic. For the first 2 figures, for each number of disks 4 to 10, every 20 examples have the same problem size. We show the average number of expanded nodes against the number of disks for a clear demonstration. As the problem size becomes larger, the number of searched nodes grows exponentially.

control the increase in a certain range. It may not necessarily lead to an optimal solution but it is efficient for a real-time problem.

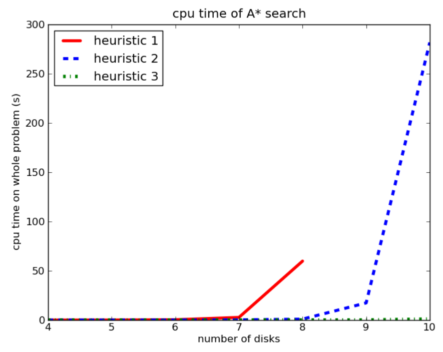
We also compared A* and RBFS (figure 3): we found that the trends of number of expanded nodes and cpu time are similar, so we only show the nodes comparison. A* outperforms RBFS in terms of cpu time and searched nodes. From figure 2, we can see RBFS spends 1/3 of the whole cpu time on heuristic computation while A* takes less than 1 second. This is



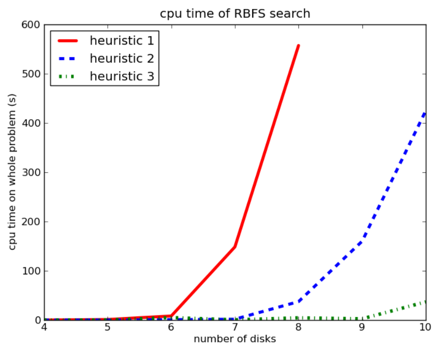
(a) A*: average cpu time (s) on heuristics



(b) RBFS: average cpu time (s) on heuristics



(c) A*: average cpu time (s) on whole problem

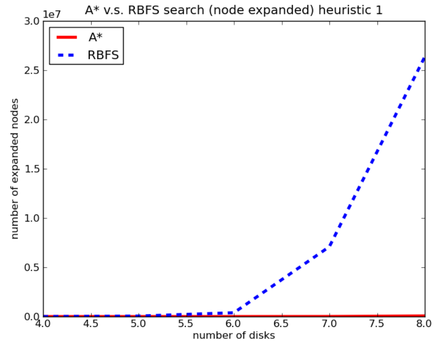


(d) RBFS: average cpu time (s) on whole problem

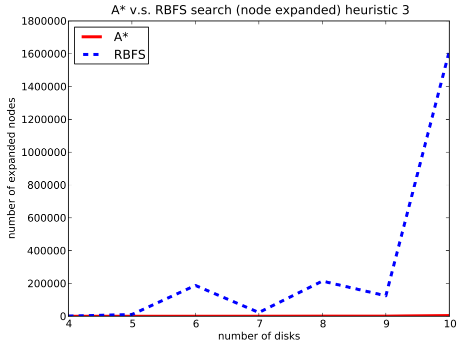
Figure 2: Plots for the average cpu time against the problem size over 20 examples for each algorithm and heuristic.

because RBFS expands more nodes than A* and every node has to compute a heuristic value.

The average solution lengths are shown in table 1. Because some experiments failed in RBFS for reaching the maximum limitation of expanded nodes, the average lengths of the admissible RBFS case are a little bit smaller without these examples, but all values are not far away. It suggests that non-admissible heuristics are also reasonable choices for large scale problems.



(a) Nodes against disks for admissible heurisitic



(b) Nodes against disks for non-admissible heurisitic

Figure 3: Performance comparisons between A* and RBFS

Disks:	4	5	6	7	8	9	10
A*/admissible	8.8	11.6	13.95	16.5	18.6	n/a	n/a
RBFS/admissible	8.8	11.6	13.95	16.35	17.11	n/a	n/a
A*/nonadmissible 1	8.85	11.85	14.6	17.5	20.2	24.1	27.6
RBFS/nonadmissible 1	8.85	11.85	15.25	19	22.21	26.81	32.25
A*/nonadmissible 2	9.25	12.95	16.5	20.05	23.8	27.4	34.05
RBFS/nonadmissible 2	11.25	17.85	24.35	32.75	40.35	50.7	64.55

Table 1: Average solution length per algorithm, heuristic and disk size. n/a means unable to compute within 10 minutes for all problems. Note that some experiments failed for completing before NMAX nodes; these were not included in the average.

5 Discussion

We discuss our results and answer the questions in this section.

1. Show an example solution sequence for each algorithm for the largest size you tested

For size $n = 10$ and A* on problem "7126049853":

[7126049853,_,_], [126049853,7,_], [26049853,17,_], [6049853,217,_], [049853,217,6], [49853,217,06], [9853,4217,06], [853,4217,906], [53,4217,8906], [3,54217,8906], [_,354217,8906], [_,8354217,906], [_,98354217,06], [0,98354217,6], [0,8354217,96], [0,354217,896], [0,54217,3896], [0,4217,53896], [0,217,453896], [0,17,2453896], [10,7,2453896], [210,7,453896], [210,47,53896], [210,547,3896], [3210,547,896], [3210,47,5896], [43210,7,5896], [543210,7,896], [543210,87,96], [543210,987,6], [6543210,987,_], [6543210,87,9], [6543210,7,89], [76543210,_,89], [876543210,_,9], [9876543210,_,_]

For size $n = 10$ and RBFS on problem "7126049853":

[7126049853,_,_], [126049853,7,_], [26049853,17,_], [6049853,217,_], [049853,6217,_], [49853,06217,_], [9853,406217,_], [853,9406217,_], [53,89406217,_], [3,589406217,_], [_,3589406217,_], [3,589406217,_], [_,3589406217,_], [3,589406217,_], [_,3589406217,_], [3,589406217,_], [_,3589406217,_], [3,589406217,_], [53,89406217,_], [53,9406217,8], [53,406217,98], [3,5406217,98], [3,406217,598], [3,06217,4598], [_,06217,34598], [0,6217,34598], [0,217,634598], [0,17,2634598], [10,7,2634598], [210,7,634598], [210,67,34598], [3210,67,4598], [43210,67,598], [543210,67,98], [6543210,7,98], [76543210,_,98], [76543210,9,8], [876543210,9,_], [9876543210,_,_]

2. 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?

The A* algorithm with nonadmissible heuristic performs the best among other heuristic type and algorithm combinations. The admissible heuristics on the two algorithms should yield the same solution lengths and nonadmissible heuristics should yield solution lengths that are not less than the solution lengths from the admissible heuristic.

3. Can a small sacrifice in optimality give a large reduction in the number of nodes expanded? What about CPU time?

The non-admissible heuristic sacrifices optimality because it does not necessarily yield an optimal solution. However, the sacrifice in optimality can yield a large reduction in the number of nodes expanded and thereby reducing the overall CPU time. For instance, an admissible heuristic can severely underestimate the true cost and yield a large number of nodes expanded. However, even though a non-admissible heuristic overestimates the true cost, it could be the case that the overestimation is very little to the true cost. Furthermore, one could overestimate to relax the problem. Therefore, there can be a large reduction in the number of nodes expanded and thus a reduction in CPU time, as compared to the admissible heuristic. A concrete example is our 2nd nonadmissible heuristic, where A* performs exceptionally well in performance.

4. How did you come up with your heuristic evaluation functions?

We created several heuristic evaluation functions based on some intuitions and for the sake of testing. One of our first heuristic evaluation functions is a simple count of the number of disks in the first peg. Then we gradually took advantage of the goal state's structure and compared it to the current state in our later heuristic designs. Hence, we came up with our heuristic evaluation functions by designing easier ones first, then gradually evolving them into more complex ones. We finally settled on three heuristics, one admissible and two non-admissible for demonstration purposes. These were covered in the previous section.

We briefly describe some of our other heuristics here, even if some were not that great:

Heuristic a Number of disks in the first peg. RBFS.

Heuristic b Total number of disks MINUS the number of matches between goal state and first peg. RBFS.

Heuristic c Double the total number of disks MINUS the number of matches between goal state and first peg, AND MINUS the streak length of the first peg to the goal state starting at the bottom. RBFS.

Heuristic d Sum the distances of all current disk positions to their goal positions plus the lengths of the second and third pegs. A*.

Heuristic e Same as heuristic d for RBFS.

Heuristic f Same as heuristic d, but enlarge heuristic value by factor of 2. A*.

Heuristic i Same as heuristic d, but instead of summing the distances, take the max distance of every disk position to their goal position on the first peg. A*.

Heuristic j Same as heuristic f for RBFS.

Note: since our two algorithms utilize different data structures, some heuristics are equivalent but use different data structures. Our organization was not the best.

5. How do the two algorithms compare in the amount of search involved and the cpu-time?

According to our results, in general, A* does a better job by expanding fewer nodes than RBFS and using less cpu-time. Furthermore, the nonadmissible heuristic does a better job than the admissible heuristic performance-wise. We can see these on our graphs, where RBFS has higher values of number of expanded nodes and cpu-time as the number of disks increases, and similarly for the admissible heuristic.

6. 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.

The algorithms could scale to even larger problems if the heuristic were designed very well. It is amazing to see an algorithm with one heuristic run very slowly with many node expansions while the same algorithm with a better heuristic finishing the job much faster with significantly fewer node expansions. Therefore, a carefully designed heuristic will allow both algorithms to scale to even larger problems by reducing the amount of expanded nodes and thereby reducing CPU time.

A smaller factor in scaling to larger problems is the algorithm design. A* may run out of memory eventually compared to RBFS since A* stores every node it expands, but this would take awhile since our computers contain a lot of memory. RBFS could scale further since it does not store every node in memory, but it will take longer to finish the job due to that trade-off. However, changing the heuristic would have the most impact because the overall number of expanded nodes would be reduced with a better heuristic, regardless of the algorithm chosen.

The largest problem we could solve is of disk size 10, using A* and the non-admissible. It took less than ten minutes. For the initial state of “”, it was able to solve the problem in 1.41 seconds and 6536 expanded nodes, with a solution length of 34 steps. As a non-admissible heuristic, it may not necessarily be the optimal solution, but the problem was nonetheless solved.

We went further and tested disks sizes above 10. We tested up to size

15. Given an initial state of $[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]$, the A* algorithm using the non-admissible heuristic was able to solve the problem in 2687 seconds and 987156 expanded nodes, with a solution length of 62 steps. That is about 44 minutes. We believe we can go higher, but it will take hours to complete. This demonstrates that our algorithms and heuristics can scale to even larger problems.

7. 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?

A heuristic function $H1$ can take longer to compute than heuristic function $H2$, but if $H1$ significantly reduces the number of nodes, then the overall time to solve the problem could be less using $H1$ than $H2$. In every experiment, we measured every time it takes to compute a heuristic function and averaged them. To take one concrete example, the heuristic of counting the number of disks in the first peg takes shorter computation time than the heuristic of calculating matches between the first peg and the goal state. As the disk size increases, the computation time noticeably but slightly increases. However, the matching heuristic performs better than the counting heuristic because it reduces the number of expanded nodes significantly. Therefore, while a heuristic evaluation function may take longer to compute, it may reduce the overall number of expanded nodes to make a significant reduction in overall computation time of the problem.