CS531 Programming Assignment 2: Towers of Corvallis

Michael Lam, Xu Hu
EECS, Oregon State University

October 23, 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. For each algorithm and heuristic function, we evaluate the performance by testing across different number of disks and different initial states.

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 summarized in algorithm **??**.

Algorithm 1 A* Search

```
exploredSet = ∅
frontier = [initialPath]
while number(explored) < NMAX do
  if frontier == ∅ 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 **??**.

3 Recursive Best-First Search

Recursive best-first search or RBFS works by storing an f-limit for each node. The algorithm uses the f-limit to decide which subtree of the problem tree to explore by considering the best and 2nd best (alternative) f-limits. In order to keep the search functional, RBFS also updates the f-values of each node during the search.

The advantage of RBFS over A* is that RBFS uses less memory. 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 **??**.

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
```

We will analyze several admissble and non-admissble heuristics in section **??**.

4 Experiments

4.1 Heuristics

We observed that the length of a path is the steps moving from initial state to goal state. So a natural heuristic can be designed based on an optimistic estimation of how many steps need to turn a state to its final goal. The first admissible heuristic is as following:

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

Here, s(i,j) is the jth disks in peg i. The intuition is that for a disk k in peg 1, it has to take $|k - i|$ steps to get to the place it should be, where i is its current location on the peg. For example, if the $9th$ disk is on the bottom, it has to take $9 - 0$ 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. This is in fact an admissible heuristic. A simple justification is given as follows: for any disk on peg 1, before it reaches its ideal position, there should be $|k - i|$ disks put underneath it, and for every such disk, it takes at least one step. Of course, a much trivial admissible heuristic can be used here, such as number of disks in the first peg, but for computational efficiency, we choose this admissible heuristic in our experiment.

For non-admissible heuristic, we simplily enlarge the admissible heuristic by a factor of 2. This is also reasonable since before we move a disk to its ideal position, usually we have to move another disk out of that place, which makes the number of steps double. But obviously this is not ture for all the cases. We can give a counterexample easily.

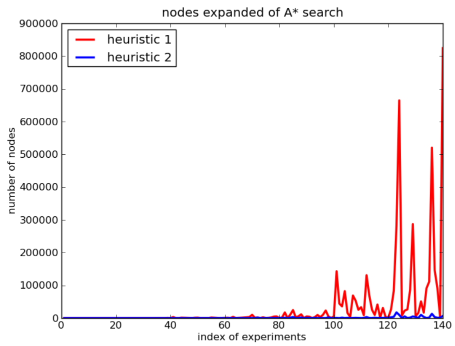
$$h(s) = 2 \times [(\sum_i |s(1,i) - i|) + numberDisks(s(2)) + numberDisks(s(3))]$$

4.2 Results

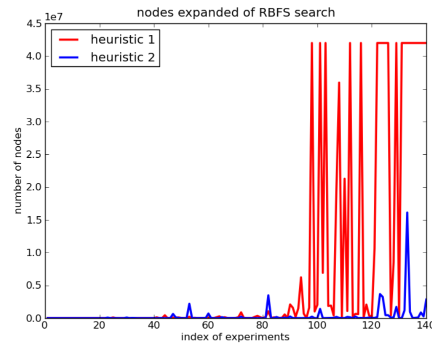
5 Discussion

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

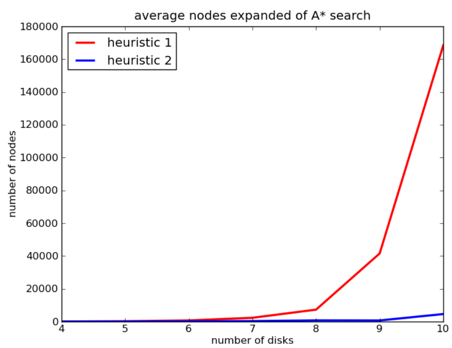
1. Show an example solution sequence for each algorithm for the largest size you tested



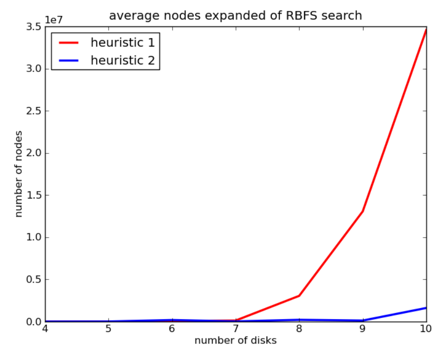
(a) A* nodes: all 140 experiments



(b) RBFS nodes: all 140 experiments



(c) A* average searched nodes

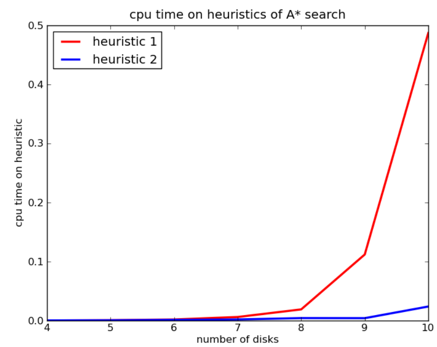


(d) RBFS average searched nodes

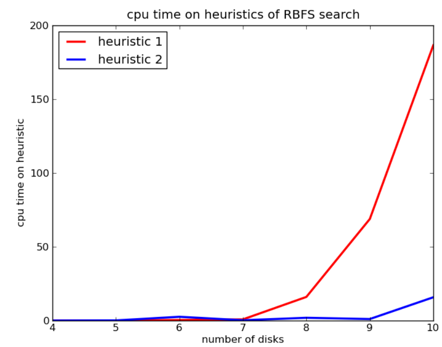
Figure 1: Plots for the number of nodes searched against the problem size for each algorithm and heuristic. For the first 2 figures, number of disks within 4-10, every 20 examples have the same problem size. We show average expanded nodes against number of disks for a clear demonstration. As the problem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

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

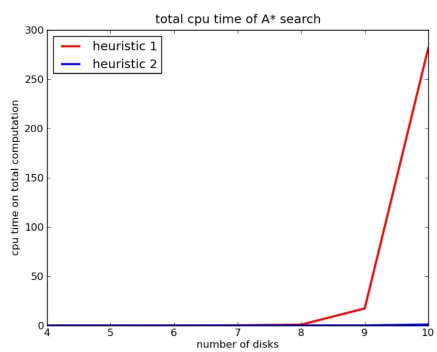
[46910517328,_,_], [126049853,7,_], [26049853,17,_], [6049853,217,_], [049853,6217,_], [49853,06217,_], [9853,406217,_], [853,9406217,_], [53,89406217,_], [3,589406217,_], [_,3589406217,_], [3,589406217,_], [_,3589406217,_], [3,589406217,_], [_,3589406217,_], [3,589406217,_], [_,3589406217,_], [3,589406217,_], [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],



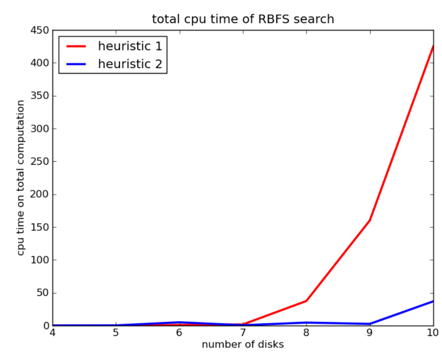
(a) A*: average cpu time (s) on heuristics



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



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



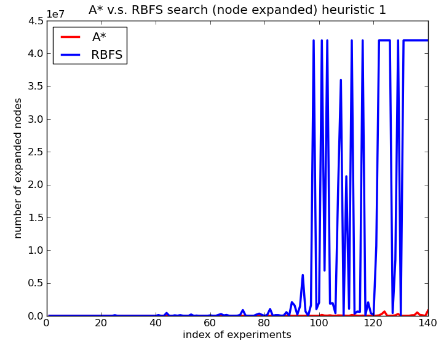
(d) RBFS: average cpu time (s) on whole problem

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

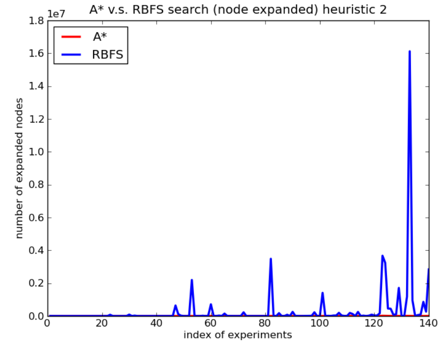
Disks:	4	5	6	7	8	9	10
A*/admissible	8.85	11.85	14.6	17.5	20.2	24.1	27.6
A*/nonadmissible	9.25	12.95	16.5	20.05	23.8	27.4	34.05
RBFS/admissible	8.85	11.85	15.25	19	22.21	26.81	32.25
RBFS/nonadmissible	11.25	17.85	24.35	32.75	40.35	50.7	64.55

Table 1: Average solution length per algorithm, heuristic and disk size.

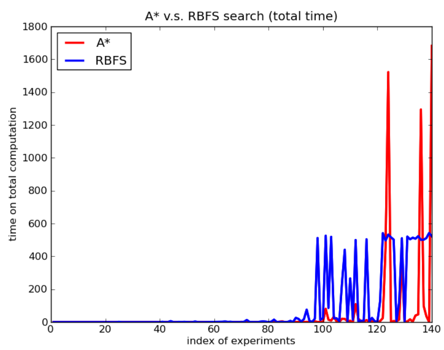
[210,67,34598], [3210,67,4598], [43210,67,598], [543210,67,98], [6543210,7,98], [76543210,_,98], [76543210,9,8], [876543210,9,_], [9876543210,_,_]



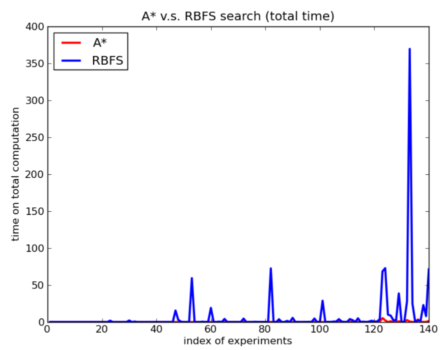
(a) Nodes against disks for heurisitic 1



(b) Nodes against disks for heurisitic 2



(c) cpu time against disks for heurisitic 1



(d) cpu time against disks for heurisitic 2

Figure 3: Performance comparisons between A* and RBFS

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?

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

For RBFS, one can sacrifice optimality by pruning parts of the search tree. While it may not necessarily yield an optimal solution, this may yield a large reduction in the number of expanded nodes if the RBFS search considers one path down the search tree without having to backtrack and expand nodes redundantly. If one is lucky, RBFS may yield an optimal solution if parts of the search tree were pruned properly.

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

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

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.

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?

8. Is there anything else you found that is of interest?