

# CS531 Programming Assignment 2: Towers of Corvallis

Michael Lam, Xu Hu  
EECS, Oregon State University

October 22, 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

## 3 Recursive Best-First Search

Recursive best-first search or RBFS works by storing a best and alternative f-limits. The algorithm uses the f-limit to decide which subtree of the problem state to explore.

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 following is the pseudocode:

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

## 4 Experiments

## 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
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?
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?