Homework Assignment 2: Implicit Search Graphs

CSC 384 – Winter 2003

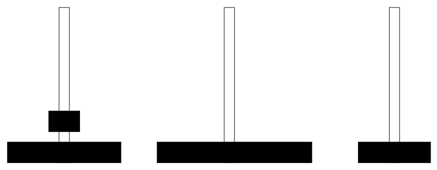
Out: February 7, 2003
Due: February 24, 2003: in class

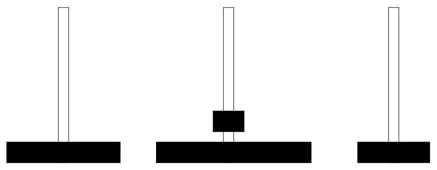
Be sure to include your name and student number with your assignment.

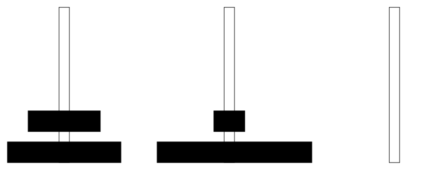
In this assignment you are going to use three different search algorithms to solve the *tower of Hanoi* puzzle. The bulk of this assignment will require that you implement a neighbor relation and three heuristic functions for the tower of Hanoi puzzle. With these defined appropriately, you will then run two versions of A* search as well as IDA* (iterative deepening A*) to attempt to solve the puzzle using each of the three heuristics. Finally, we'll get you to compare the performance of the three heuristics and the different search algorithms.

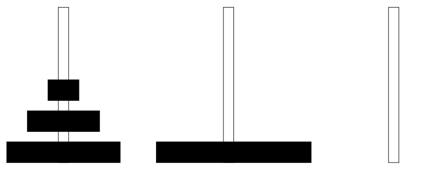
First, a bit of background. The tower of Hanoi puzzle is a simple (one-person) game. You are given a tower of N disks initially stacked in increasing size (from top to bottom) on one of three pegs. The objective is to transfer the entire tower to one of the other pegs (say the rightmost), moving only one disk at a time and never a larger one onto a smaller one. You can play the game online at `http://www.cut-the-knot.com/recurrence/hanoi.shtml`. The website has a nice applet as well as some explanation regarding an optimal strategy.

For the purpose of the assignment, we will allow any configuration of 4 disks (on 3 pegs) as a starting position. A configuration is legal as long as the disks stacked on each peg are in increasing size (from top to bottom). For example, in the figure below, we see four (legal) game configurations. Similarly, we will allow any (legal) game configuration to be the goal configuration. The objective will simply be to move the disks from one peg to another (without ever putting a larger disk on a smaller disk) until the goal configuration is obtained. Moving the smallest disk in configuration (a) to the middle peg results in configuration (b). Moving the second smallest disk in configuration (b) to the leftmost peg results in configuration (c). Finally, moving the smallest disk in configuration (c) to the leftmost peg results in configuration (d).


(a)


(b)


(c)


(d)

1. We will now model the game as a search problem. Each (legal) configuration corresponds to a state. Each possible (legal) move of a disk from one peg to another corresponds to a directed arc linking two neighboring states. Let's analyze the size of the graph representing the game. How many states are there (assuming a game with 4 disks)? What's the maximum branching factor of any state (i.e., largest number of neighbors for any node)?

2. Next, you should implement a neighbor predicate. For any *state* of the game `State`, the predicate `nb(State,ArcList)` should hold iff `ArcList` contains the list of all neighboring states of `State`. To keep things uniform, you *must* use the following representation of a state. A state is a list of pegs `[LeftPeg,MiddlePeg,RightPeg]` ordered from left to right. In turn, each peg is a list of disks stacked in increasing order (from top to bottom). Let's label the disks according to their size (i.e., 4 for the largest and 1 for the smallest). Here are examples of the state representations of the above four configurations:

   (a) `[[1,3],[4],[2]]`

   (b) `[[3],[1,4],[2]]`

   (c) `[[2,3],[1,4],[]]`

   (d) `[[1,2,3],[4],[]]`

   In addition, in the predicate `nb(State,ArcList)`, let `ArcList` be a list of `arc(Nb,Cost)` such that `Nb` is a neighboring state reachable with cost `Cost`. In this game, we are interested in minimizing the number of moves necessary to reach the goal configuration, so the cost will always be 1.

   Following this syntax for the `nb` predicate should allow immediate integration with some code provided on the course webpage to run A* and IDA* (see remainder of the assignment). In addition, for uniformity and ease of marking, the `ArcList` you produce *should list arcs in a specific order*. Recall that each arc can be viewed as moving a disk from one peg to another. Denoting L for left peg, M for middle peg and R for right peg, list the arcs in the following order: $L \rightarrow M$, $M \rightarrow L$, $L \rightarrow R$, $R \rightarrow L$, $M \rightarrow R$ and $R \rightarrow M$. Of course, if some of these moves are not possible or lead to an illegal configuration (e.g., larger disk on top of a smaller disk), then they should not be in the list. For example, the query `?nb([[1,2,3,4],[],[]],ArcList)` should return `ArcList = [arc([[2,3,4],[1],[]],1),arc([[2,3,4],[],[1]],1)]`.

   What to hand in: Hand in a listing of your code (all relevant predicate listings) and a printout of a prolog session showing that your predicate `nb` works correctly on four well-chosen test cases. Choose those test cases yourself and give a short (one sentence) explanation of what distinguishes each test case from the others.

3. You should next implement three predicates, `h1`, `h2` and `h3`, defining heurisitic functions as follows.

   - `h1(State,Goal,Dist1)` is true iff the distance between a `State` and the `Goal` configuration is estimated to be `Dist1` where `Dist1` is the number of disks in `State` that are *misplaced* with respect to the goal configuration. A disk is considered misplaced if (a) it is on the wrong peg or (b) one of the disks that should be underneath it (in the goal configuration) is misplaced. Assuming the goal state is `[[],[1,3,4],[2]]`, the estimated distance `Dist1` for the four states above are 2, 2, 3, and 3, respectively.
   - `h2(State,Goal,Dist2)` is true iff the distance between a `State` and the `Goal` configuration is estimated to be `Dist2` where `Dist2` is the sum of some quantity $q_2(mspDisk)$ for every misplaced disk $mspDisk$. This quantity $q_2(mspDsk)$ is $1 + \#$ of disks on top of $mspDsk$.

   $$Dist2 = \sum_{mspDsk} (1 + \#\text{disks\_on\_top\_of}(mspDisk))$$

   Intuitively, if a disk is misplaced, we must first move all the disks on top of it before moving it. Assuming the goal state is `[[],[1,3,4],[2]]`, the estimated distance `Dist2` for the four states above are 3, 2, 4 and 6 respectively.

- `h3(State,Goal,Dist3)` is true iff the distance between a `State` and the `Goal` configuration is estimated to be `Dist3` where `Dist3` is the sum of some quantity $q_3(mspDisk)$ for every misplaced disk $mspDisk$. This quantity $q_3(mspDsk)$ is exponential in the number of disks on top of $mspDsk$.

  $$Dist3 = \sum_{mspDsk} 2^{\#\text{disks\_on\_top\_of}(mspDisk)}$$

  Intuitively, if a disk is misplaced, we must first move all the disks on top of it and this takes an exponential number of moves. Assuming the goal state is `[[],[1,3,4],[2]]`, the estimated distance `Dist3` for the four states above are 3, 2, 4 and 7 respectively.

Tip: first code a predicate that identifies misplaced disks in each peg, then use this predicate in each heuristic predicate with, if necessary, an appropriate predicate to compute the quantity q_2 or q_3. Points will be deducted for unnecessarily complex or inelegant definitions.

What to hand in: Hand in a listing of your code (all relevant predicate listings) and a printout of a prolog session showing that your predicates `h1`, `h2` and `h3` work correctly on four well-chosen test cases. Choose those test cases yourself and give a short (one sentence) explanation of what distinguishes each test case from the others. Of course, choose test cases different from those given in example in this handout.

4. I will post a version of standard A* on the course Web page, called `asearch`, but with some tweaks to allow it to display solutions a little more clearly for this problem. You should test standard A* search on the tower of Hanoi puzzle using *all three heuristics*. Some details of the implementation will be discussed at the end of this document. You will be given a standard set of test problems (via the course Web site) within roughly a week of the assignment being handed out. *You should modify the program to help you count the number of expanded nodes to help you answer question 7 below. Since the number of nodes expanded before the goal state is reached may be much larger than what you are willing to wait for, impose a limit of 4000 nodes on the number of nodes that A* will expand.*

   What to hand in: Hand in a printout of the prolog session showing your program running on each problem in the test set, using all three heuristics. You should clearly label each run so the marker can tell which heuristic and case is being tested. *Do not hand in a code listing for the program. Do not hand in a run that has many lines of extra output, even if you need to modify the program for your own purposes to do this. Hand in only runs of a version without output except the solution.*

5. Implement a second version of A* called `asearch2`. This will not be too different from `asearch` (a lot of the same predicates can be used). But `asearch2` should perform *multiple path checking*. The simplest way to do this is to maintain a *closed list* of nodes that have been expanded. You should probably add this as an argument to the predicate `asearch2`. When you select a node from the frontier, it should not be expanded (i.e., its neighbors are not added) if it is already on the closed list. You can modify the interface `start_search` to call `asearch2` appropriately. You can use predicates `member` and `notmember` (which will be made available online) to implement your closed list test. Hand in a listing of your code.

   You should test your A*-MPC search algorithm on the tower of Hanoi puzzle using *all three heuristics*. You will be given a standard set of test problems (via the course Web site) within roughly a week of the assignment being handed out. *You should ensure your program expands no more than 4000 nodes.*

   What to hand in: Exactly as in Question 4, with the addition of your code listing.

6. I will post a version of IDA* (iterative deepening A*) suitable for use with the tower of Hanoi puzzle. You should test IDA* search on the puzzle using *all three heuristics*. You will be given a standard set of test problems (via the course Web site) within roughly a week of the assignment being handed out. *You may wish to modify the program to help you count the number of expanded nodes to help you answer question 7 below. You should ensure your program expands no more than 9000 nodes.*

   What to hand in: Exactly as in Question 4.

7. Draw up a table that compares the number of nodes expanded by each of the three search algorithms (A*, A*-MPC, IDA*) for each of the three heuristics and for each of the test cases posted. Your table should take the form:

	A*	A*-MPC	IDA*
Case 1, h1	x	x	x
Case 1, h2	x	x	x
Case 1, h3	x	x	x
Case 2, h1	x	x	x
Case 2, h2	x	x	x
Case 2, h3	x	x	x
etc.			

   If any combination reaches the bound on the number of expanded nodes before solving the problem, please indicate that fact in your table. You should indicate the number of nodes expanded before abnormal termination in your table. What conclusions can you draw about the quality of the heuristics $h1$, $h2$ and $h3$?

8. The suggested implementation for multiple path checking using a closed list will not guarantee that A* finds a least-cost path in general. But it will work if the heuristic function h satisfies the *monotone restriction* (see p.139–140 of the text). Give a convincing (rigorous) argument that $h1$, $h2$ and $h3$ all satisfy the monotone restriction. The proofs for $h1$ and $h2$ should be relatively easy. The proof for $h3$ is a bit tricky so here is a hint: the intuition behind $h3$ is that it takes at least 2^n to move a stack of n disks. Have a look at the sections *Recursive solution* and *Recurrence relations* on the website `http://www.cut-the-knot.com/recurrence/hanoi.shtml`.

Appendix: Here are a few notes on the A* implementation you are given. The IDA* implementation is similar in its representation of paths, etc. The predicate `asearch(Frnt,Path,Count)` is available from the course Web page, and is implemented under the following assumptions and design choices:

- Neighbours are represented using a predicate of the form `nb(N, [arc(N1, C1), arc(N2, C2) ...])`. Here `N` is a node and `N1`, etc. are its neighbours, with corresponding arc costs `C1`, etc. (much like in the Bicycle Courier handout, but using `arc(N2, C2)` instead of `[N2, C2]`).
- Heuristic values are represented using a predicate of the form `h(N, Hvalue)` that tells you the heuristic value of a given node `N`. In order for `h` to map to `h1` say, define `h(N,Hvalue) :- is_goal(G), h1(N,G,Hvalue).` Here `is_goal(g)` is a predicate that you must define to indicate the goal state `g`. Note that `is_goal(g)` is also used by A* to know when to end the search.
- The frontier is maintained as a *priority queue*. The queue is implemented as an ordered Prolog list, in which each path on the queue is stored in ascending order of its f-value. (Recall $f(n) = g(n) + h(n)$.)
- The frontier is managed efficiently using ordered insertion.
- Each element on the frontier is a data item containing a path with its g-value, and its f-value. The key by which the queue is sorted is the f-value. We've used the data structure `path(PC,P)` for the path and their costs, where `PC` is the path cost (its g-value) and `P` is the path (a list of nodes, from the last to the first). For instance, the term `path(3,[eif,al,mo])` represents the path from `mo` to `al` to `eif`, with an actual cost of 3.
- We have provided an interface to `asearch` in the form of predicate `start_search(Node,Path)`. This calls the A* search procedure with start node `Node` and returns the solution `Path` (including its cost).
- To run the `start_search(Node,Path)` predicate to solve a particular goal `g`, you must specify `is_goal(g)`.