Jake Bareket

jhb334 3674042

1. Intelligence is the ability to come up with effective or creative solutions to previously unencountered problems, questions or tasks, as well as the ability the ability to produce predictions, or analyses based on previously encountered data.

2a. The states are all possible arrangements of the n disks on the three polls such that every disk is on a poll and no disk is on top of a smaller disk. This can be represented by creating three stacks to represent the three polls (titled left, middle and right) and filling them with numbers 1-n (to represent the disks) according to the following rules.

1. All elements in each stack must be in increasing order from the bottom of the stack to the top of he stack.

2. Each number 1-n appears exactly once in each state and can be found in one of the three stacks.

- The initial state is a state in which the middle and right stacks are empty and the left stack has n numbers arranged sequentially in the stack. The number one is on the top of the stack and the number n on the bottom of the stack

- The goal state is a state in which the left and middle stacks are empty and the right stack has n numbers arranged sequentially in the stack. The number one is on the top of the stack and the number n is on the bottom of the stack

1. Pop the top number from stack 1 and add it to the top of stack 2

2. Pop the top number from stack 1 and add it to the top of stack 3

3. Pop the top number from stack 2 and add it to the top of stack 1

4. Pop the top number from stack 2 and add it to the top of stack 3

5. Pop the top number from stack 3 and add it to the top of stack 1

6. Pop the top number from stack 3 and add it to the top of stack 2

- the branching factor is three since if you can move a disk from stack i to stack j, you will always be unable to (on the same turn) move a disk from stack j to stack I (since if you can move a disk from stack i to stack j, then the disk on top of stack i must be smaller than the disk on top of stack j)

All edges are bidirectional(all operators are reversible)

3a. The states are two nodes the first of which contains an integer between 0 and 10 and the second of which contains an integer between 0 and 6. This can be represented formally as (jug1: int), (jug2: int).

The initial state is (jug1:0), (jug2: int) and the goal states are all states where (jug1:0), and jug2 contains an int between 0 and 6 (inclusive).

The operators are

1. fill jug1 ie. set (jug1:10)

2. fill jug2 ie. set (jug2:6)

3. Pour from jug 1 to jug 2 until either jug 2 is full or jug 1 is empty. If jug 2 is full stop pouring and whatever is left in jug 1 remains the number of jug 1.

4. Pour from jug 2 to jug 1 until either jug 1 is full or jug 2 is empty. If jug 1 is full stop pouring and whatever is left in jug 2 remains the number of jug 2.

5. Empty jug 1 ie set (jug1:0)

6. Empty jug 2 ie set (jug 2:0)

The branching factors is 4 as it will never be possible to repeat an operator. Once a jug is full it will be impossible to refill it and once a jug is empty you cannot re-empty it. Similarly when you pour from one jug to another, it will be impossible to pour a second time from the same jug into the same other jug because after the first act of pouring, either the jug from which you poured must be empty or the jug into which you poured must be full.

Similarly, it will never be possible to follow up fill jug 1 with pour from jug 2 to jug 1. Nor will it be possible to follow up fill jug 2 with pour from jug 1 to jug 2. It will also never be possible to follow up empty jug 1 with pour from jug 1 to jug 2 nor will it be possible to follow up, empty jug 2 with pour from jug 2 to jug 1. It will also be impossible to follow up a pour from jug 1 to jug 2 operator with both an empty jug 1 operator and a fill jug 2 operator as after a pour from jug 1 to jug 2, either jug 1 must be empty or jug 2 must be full. It is similarly impossible to follow up a pour from jug 2 to jug 1 operator with both a fill jug 1 operator and an empty jug 2 operator as valid options.

3b,c Blue arrows indicate an operation is reversible. Red Arrows indicate the path to the goal state.

4a. The branching factor is 6 (You can increment or decrement any of the values and there are three values so the branching factor is 6.) The search space is all possible three tuples where each tuple can be any integer.

4b. For k=0 the number of states is 1. For all other K the number of states

=

explanation: we can think of the data as points on a 3d graph

without doubling back there are 4(k-|i|) points in each plane parallel to the xy plane where i represents which is z coordinate of your plane. This does not include the ith and negative ith plane each of which has one point on it. Therefore our function for the number of points is a sum of the points on each plane. (we sum up and down and avoid double counting the middle plane).

number of states = = 2+4(k)(k+1)/2+ 4(k-1)(k)/2

=2+(4k^2)=2+4k^2

4c. H is admissible since H is always >=0 and H is always <=H\*. We can see that H>=0 because H is a sum of three absolute value terms which means H can never be negative. H<=H\* because all operators only change one value of the tuple at a time and they only change it by + or minus 1. Therefore H\* is = to the sum of the difference between the final states and the current states of each value. So H\*=|x-a|+|y-b|+|z-c|=H therefore H is <=H\*.

d. H is still admissible since H is always >=0 and H is always <=H\*. We can see that H>=0 because H is a sum of three absolute value terms which means H can never be negative.

We can see that H<=H\* since H calculate the number of steps that the current tuple (x,y,z) is from the goal tuple (a,b,c). Since the maximum amount that we can change any value by is 1, it will always take at least H steps to get from (x,y,z) to (a,b,c). So H<=H\*.

e. H is not admissible since we can construct a situation where H>H\*.

Proof: if x =1,y=0,and z=0 and a=4,b=0 and c=0 then

h(x,y,z) = |x-a|+|y-b|+|z-c|=4, however x is odd so you can increment x by either one or three. Therefore H\* =1. So we have constructed a case in which H>H\*

f. We can use the following heuristic evaluation function

f(x,y,z)= |x-a|+|y-b|+|z-c| and therefore we accept a change if it decreases f(x). This works because the heuristic function calculates the distance of each one of the values independently and will be sensitive to a change in only one of the values. If the value changes so that it is farther from the goal state f(x) will be greater and if it is closer to the goal state f(x) will be smaller.

5

5a. No because h1(s) may be greater than h\*(s) as in the following counterexample:

h(s)=2 and h\*(s)=3. In such a case h1(s) would be equal to 4 and therefore h(s)>h\*(s) which means h is not admissible.

5b. Yes because if h2(s)=h(s)/2 then for all positive h(s), h2(s) will always be less than h(s)m therefore: h2(s)<h(s)<=h\*(s) (since h(s)<=h\*(s) is part of the definition of admissibility.)

We know that h2(s) also must always be positive. This is because h(s)>=0 implies that h(s)/2 >=0

5c. No because h1(s) could be negative and one of the rules for admissibility is that h(s)>=0.

5d.

1. If a = b than a and b can be any positive values.

2. If b=0 then a can be any value except 0.

3. If a=0 than A\* is not guaranteed to find an optimal solution regardless of what b is.

4. b can be any value between 0 and 1 (inclusive) as long as b<a.

5. Neither a nor b can be negative.

Proofs: 1.Given that f(s)=g(s)+h(s) is guaranteed to find an optimal solution (proved in lecture), a x g(s)+b x h(s) is also guaranteed to find an optimal solution if a and b are equal because scaling f(s) will not effect which node is settled next since all f values are scaled by the same amount.

a and b cannot both cannot be equal to zero because then f(s)=0 which means the search is simply random and that would not be guaranteed to be optimal because if nodes are randomly settled you may find a suboptimal path first.

2. If b is 0 then that is the same as just doing a simple case of best first search and best first search was proven in lecture to be guaranteed to find an optimal solution. In a case of f(s)=a x g(s) It is irrelevant what a is because a just scales f(s) will not effect which node is settled next since all f values are scaled by the same amount.

3. If a=0 then we are searching only using h(s), however h(s) could be equal to 0 which means f(s) would equal 0. f(s)=0 means the search is simply random and that would not be guaranteed to be optimal because if nodes are randomly settled you may find a suboptimal path first.

4. Since any h(s) function is admissible as long as 0<=h(s)<=h\*(s), making b between 0 and 1 means 0<=b x h(s)<=h\*(s). So if 0<=b<=1 then b x h(s) will always be admissible. a merely must be greater than b because otherwise a situation could occur in which b x h(s) is larger than a x g(s) on the final node which would be a comparable problem to h(s) being larger than h\*(s) and could lead to a sub optimal solution.

It is ok for a to be arbitrarily large as long as it is bigger than b and b is between 0 and 1 since scaling g(s) up will still never allow a longer path final path to be chosen over a shorter final path since when settling the final node g(s)=h\*(s)>=h(s). which still holds true if we scale g(s) upwards while keeping h(s) between – and h\*(s)

5. If a is negative then a problem arises easily in the case where h(s)=0 since then f(s) would be equal to the negative values of the true distances scaled by some constant which would result in the smallest distance being the most negative distance which means best first search would always big the biggest distance first and the smallest distance last. Exactly the opposite of what it is supposed to do.

If b is negative than for all positive values of a, regardless of the magnitude of b it would be possible to construct a situation where their are two nodes with extremely close values for g(s) and extremely large values for g(s) and a function h(s) which for one of the nodes is equal to 0 and for the other node is equal to h\*(s) which means that, g(s)+b\*h(s) (where b is negative) would cause the closer node to have a greater f(s) value.