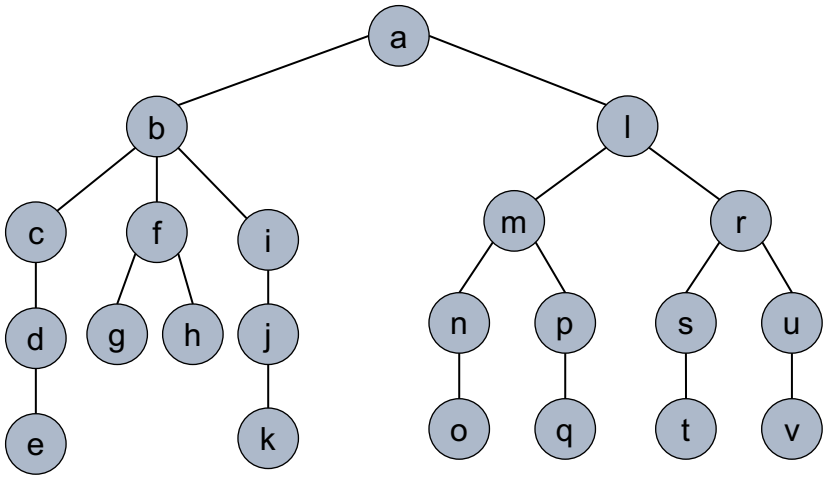
INF 4130 Exercise set 4

Exercise 1

List the order in which we extract the nodes from the Live Set queue when we do a breadth first search of the following graph (tree) with the Live Set implemented as a LIFO queue.



a l r u v s t m p q n o b i j k f h g c d e

Exercise 2

Solve exercise 23.6 from the text book (B&P). (See the course web page for a scan of the book.)

Note that it is important that we do not count the empty square when we sum up what is in the wrong spot. If we do, we won't be able to show what we are supposed to. See, for instance, this 2x2-board:

3	1
	2

Here we can get everything in place with three moves (U,R,D), but if we also count the empty square we get $h(v) = 4$. Thus, counting also the empty square, the shortest distance would be smaller than the heuristic, which can never be true for a monotone heuristic.

For monotonicity we must show:

1. $h(\text{goal}) = 0$
2. If there is an edge from v to w with cost $c(v,w)$, we must have: $h(v) \leq c(v,w) + h(w)$.

1. is obvious

2. For any simple move *(v,w)* we know that *c(v,w)* =1 (as we simply count the number of moves along a path). Thus, if *h(v) and h(w)* do not differ by more than 1, the above inequality must be true. This is obvious as only one tile is moved.

Exercise 3

Solve exercise 23.7 from the text book.

If we understood the point of the previous exercise, this one should also be easy.

Monotonicity:
1. *h*(goal) = 0
2. If there is an edge from *v* to *w* with cost *c*(*v*,*w*), we must have: $h(v) \leq c(v,w) + h(w)$.

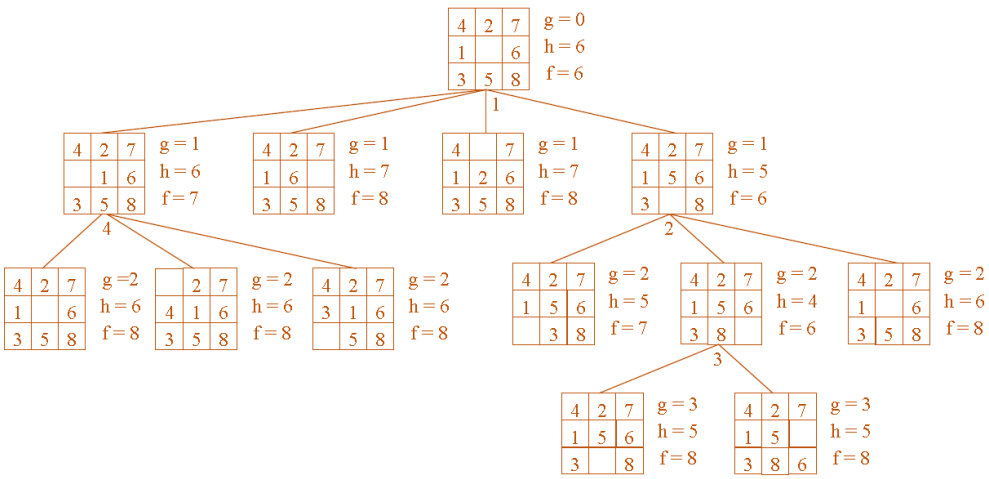
1. is obvious

2. Again *c(v,w)* is 1. Thus, if *h(v) and h(w)* do not differ by more than 1, the above inequality must be true. This is obvious as only one tile is moved one step.

Exercise 4

Solve exercise 23.8 from the text book.

The numbers under (some of) the nodes shows the order in which these nodes are taken from the queue to the tree. In the fourth move we have two nodes with $f = 7$, but we assume a FIFO order for nodes with equal priority. In the next step the other node with $f = 7$ gets expanded further.



Exercise 5

Is your answer regarding monotonicity in 23.7 also valid if we allow moving the hole diagonally?

Monotonicity:

- h(goal) = 0
- If there is an edge from v to w with cost $c(v,w)$, we must have: $h(v) \leq c(v,w) + h(w)$.

The sum of the Manhattan-distances will not be a monotone heuristic if we allow diagonal moves, because it can get larger than the number of moves necessary. Look at:

1	2	3
4		6
7	8	5

Here we get a solution in one move (DR – moving the hole into the square with 5), but the Manhattan-distance gives us $h = 2$.

It is, however, easy to see that if a square is in (x_1, y_1) and its correct position is (x_2, y_2), the fewest number of moves to (x_2,y_2) (if it is alone on the board) is: max($|x_2-x_1|$, $|y_2-y_1|$). We therefore try with the sum (over all tiles) of this value as our heuristic. For this heuristic we have h(final state) = 0.

To show monotonicity we again have to show that the h-value does not change more than 1 when we make a move. Since only one tile is moved, the question is really whether max(a,b) can change more than 1 when a can increase or decrease by 1, and the same for b (at the same time). It should be clear that it cannot (since a max-function is a kind of OR). So the modified heuristic is monotone.

Exercise 6

Is it possible to use the actual cost as our heuristic (it is after all 100% exact and should be good)? Will the actual cost always be monotone? Will we expand a smaller tree? What would the problem be, if any?

To clarify a bit, $h(s)$ is now the cost of the moves along an optimal path from s to a goal state. It is of course a bit strange to imagine that we have this as our heuristic: If we only want the length of the shortest path, we already have what we are looking for. However, if we know the length of the shortest paths to the goal, it can be used to find the path itself (but this can then probably then be done in a simpler way).

This exercise can of course tell us something about the behaviour of A* with very good heuristics. If h is as described above, we see that the value $f(s) = g(s) + h(s)$ when state s is taken out of the priority queue and placed in the tree, is equal to the shortest path from the start state to a final state, through s. It should therefore be clear that the A*-algorithm will go straight for the best solution and not waste time on sub optimal solutions. Many states may, of course, have the the same f-value, and in these cases the algorithm will follow them “in

parallel", the order will depend on how nodes with equal priority are removed from the priority queue. Another point to observe is of course that calculating the heuristics here usually amounts to solving the originally problem itself (unless you have got these distances from other sources).

Exercise 7

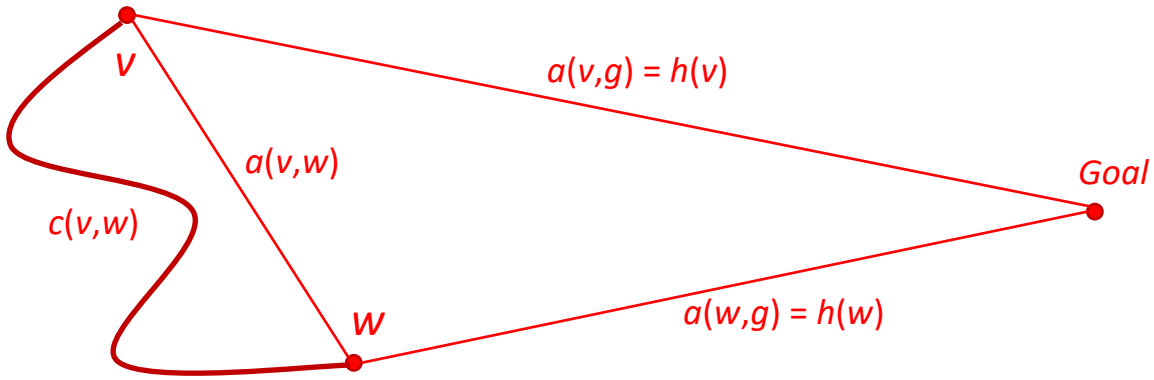
Show that the straight line (actually the circumference of a great circle, but let's not get into details) between a point and the goal point is a monotone heuristic for finding the shortest path the way it is done in chapter 23.3.3 (page 728).

Monotonicity:

- $h(\text{goal}) = 0$
- If there is an edge from v to w with cost $c(v,w)$, we must have: $h(v) \leq c(v,w) + h(w)$.

Let g be our goal. We first observe that we have $h(g) = 0$.

Now let v and w be two places that there is a road with distance $c(v,w)$ between, and let the straight line between them be $a(v,w)$. We have to show that $h(v) \leq c(v,w) + h(w)$. We have $h(v) \leq a(v,w) + h(w)$ by the triangle inequality, and since $c(v,w) \geq a(v,w)$, we get $h(v) \leq a(v,w) + h(w) \leq c(v,w) + h(w)$. It is therefore clear that the straight line is a monotone heuristic.



Exercise 8

Assign g-, h- and f-values to the states in figure 23.7 (page 727) and check that we actually avoid expanding the full breadth-first-tree in figure 23.3 (page 719).

Left to the individual student.

Exercise 9

Adjust the DFS procedure below to instead do iterative deepening with one extra level at a time. You should only check once for each node whether it is a goal node, and you need an extra parameter to the procedure DFS. Show how the procedure should be called from a "main" program/procedure for the whole thing to work properly

```
proc DFS(v) {
   if <v is a goal node> then return "…"
   v.visited = TRUE
      for <each neighbor w of v> do
         if not w.visited then DFS(w)
      od
}
```

Things get a little complicated when we search in a graph (as indicated by the variable "visited" in the assignment text). We therefore first assume that we search in a tree, so that we never come to an earlier visited node. The program below only test for goals in the "new nodes", and the outer loop increases the depth of the search by 10 (incLev). The tree (State) nodes have a procedure to test whether it is a goal node.

```
class State { <variables to keep track of its children in the tree>
      Bool proc isGoal(state node){ checks whether this node is a goal }
}

class IDDFS{                          // class for iterative deepening DFS
   int incLev = 10;                   // as an example
   int prevLev = 0; int maxLev = prevLev + incLev;
   State foundGoal = null;            // Is set when a goal is found

   State proc main(State root){    // Is the root of the tree to be searched
     repeat {
       IDSearch(root, 1);     // The recursive call for the root.
       prevLev = maxLev; maxLev = prevLev + incLev;
     }
     until foundGoal != null;

     return foundGoal;
   }

   proc IDSearch(State node, int lev){  // the recursive procedure
      if (lev > maxLev) return;
      if (lev > prevLev){
         if (node.isGoal) {foundGoal = node; return}
      }
      for each child of node do { IDSearch( child, lev + 1); }
   }
}   // End of class IDDFS
```

We should here also have added a variable holding e.g. the number iterations we should perform before we totally give up searching. The use and administration of such a variable should be straight-forward. Without such a mechanism, the program will easily enter an infinite loop.

If we want to do search in a graph, we, in the program given in the exercise text, is using a Boolean variable “visited” in each node to avoid looking at the same node twice. However, if we will use iterated deepening (ID) for the graph case we will have to set this variable back to **false** between the iterations, which will take a lot of time. We will therefore use an interger variable “visitNo” instead of the Boolean one. We will use this variable as follows:

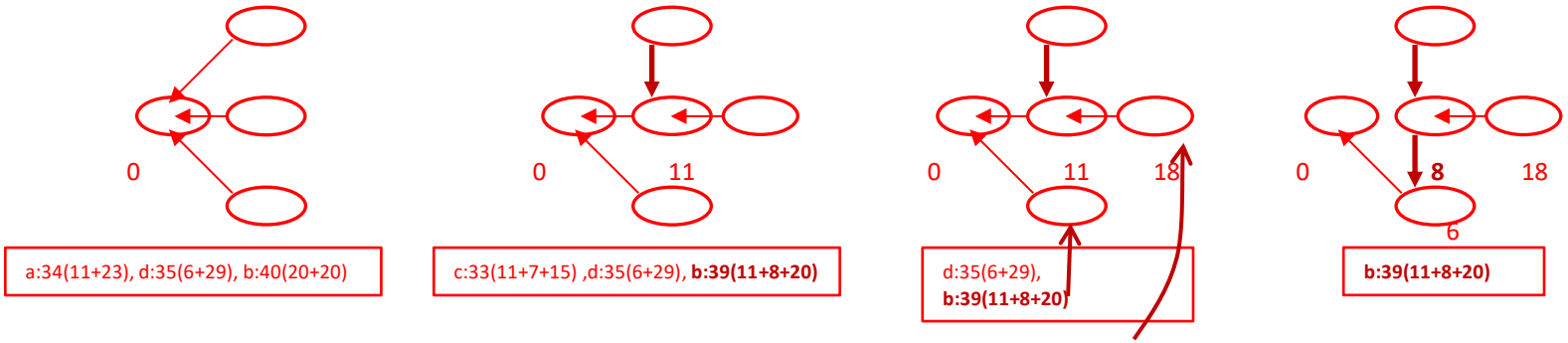
- At the start, visitNo should be 0 in all nodes.
- We will have a global (= local to the class `IDDFS`) interger variable “iterNo”, which holds the number of the current iteration. This is incremented between iterations.
- We can then test whether we have seen this node in this iteration by “node.visitNo < iterNo”.
- If this is true we should immediately do “node.visitNo = iterNo” to mark it as seen in this iteration. Note that this will also work for nodes that have not been seen in the earlier iterations, as they will have “node.visitNo == 0” (and thus the first iteration should have “iterNo == 1”.

We leave the adjustment of the above program to accommodate these ideas to the interested student.

Exercise 10

Study the example on slide 20 from September 9 (page 723 on the textbook) to confirm that when $h(v)$ is not monotone, then nodes sometimes will have to be taken back from tree to the priority queue, thus increasing execution time.

The evolution of the tree and the priority queue looks like this:



Dark red indicates a value had to be updated.

Both *b* (priority in queue) and *c* (distance) now have incorrect values, because of the shorter path to *a.*

Exercise 11

Study the A*-algorithm described (textually) on slide 26 from the lecture.

Is left to the participants in the group, or to self study.

[end]