Artificial Intelligence 1
Lab 1

Roeland Lindhout (s2954524) & Younes Moustaghfir (s2909758)
AI2/AI3

May 2, 2016

Theory

Exercise 1

Reversi The performance measure is the ability to win, or on a smaller scale, the ability to turn over as many of the opponent's discs as possible. The environment is a board, divided into 8x8 squares, all of the same color. The actuators are the hands and arms of the player's, and the sensors are the senses the players have, e.g. the eyes and the nerves in their fingers. The environment can be described by the following features:
Fully observable, because the entire playing board can be seen.
Deterministic, because the next state is a consequence of the current state and the action.
Sequential, because the players should have some sort of tactic to win, therefore some form of future planning should take place.
Semi-dynamic, because the player could have a planned action, which has to be altered when the opponent makes a different move than expected.
Discrete, because the game is played move by move, and therefore not continuous.
Multi-agent, because two players are involved.
The Reversi agent architecture is a utility-based one, because there are often multiple options to place the discs, and therefore the player has to decide which option is best to reach to goal, i.e. which option has the highest utility.

Robotic Lawn Mower The performance measure is the length of the grass. The environment is a field of grass, with borders indicated by a wire. The actuators are the robot's wheels and the knives to cut the grass. The sensors change, depending on the way in which the borders are indicated. It could be some pressure sensor, that is activated when it hits the border. Sophisticated lawn-mowers can have rain sensors and sensors to detect the length of the grass, but with less sophisticated types, the robot is just always cutting, independently of the length of the grass. The environment can be described by the following

features:
Partially observable, because the robot only senses the wire used to indicate the borders when it reaches the line. It does not always know where the line is, so it is not completely aware of the environment.
Deterministic, because the robot moves on and/or cuts grass. The relevant parts of the environment state, i.e. the length of the grass, are therefore determined by the robot's action. Someone could walk into the field, but that is not relevant to the robot.
Episodic, because the robot moves somewhere, then decides to either cut or not cut the grass, and then moves to the next place, without any memory of some kind. It just systematically moves over the field. The decision to cut grass does not depend on where it has been or where it will go.
Dynamic, because objects could be placed or removed from the field, and the borders can be moved during the cutting of the grass.
Continuous, because the robot is always moving around the field until it is shut down.
Single-agent, because the only one involved is the grass-mower itself.
The best agent type for a robotic lawn mower, is a simple-reflex architecture. If the grass is long, cut it and if a wire is reached, turn are basically all it needs to be able to successfully move over a simple field.

Exercise 2

1. $mazeDFS()$ is unable to find a path from the yellow to the red square because of the order in which the actions are taken. The order is currently [N, E, S, W], which results in that particular maze in an infinite loop at position 6. Each time that position 6 is visited, position 5 will be the last to be pushed, therefore the first to be popped in the next iteration. At position 5, position 6 is the last position to be pushed, making it the first position to be popped in next iteration. This continues until the stack is full and no goal will be found.

2. A way to make sure that $mazeDFS()$ always finds a solution is to keep track of the positions that have already been visited. If the neighbour position is in an array called visited, then this neighbour should not be visited again. In the pseudocode this part will be added after checking if the move is possible. The already visited positions won't be pushed to the stack.

3. The path that the algorithm will take is: 1-2-6-5-9-13-14-10-7-3-4-8-12-11-15

4. The path for this algorithm is: 1-2-6-7-3-4-8-12-15

5. If the algorithm is fitted with a FIFO queue to turn it into an BFS version, the algorithm will turn into a complete one. If correctly implemented, this algorithm will always return a solution simply because BFS is a complete

algorithm and will search through all possible states. Whilst BFS is not the fastest algorithm, it will always find a solution if one exists.

6. The path that this algoritm will take is: 1-2-6-7-5-3-9-4-13-8-14-12-10-11-15

7. Again, implementing a visited array to check for visited states. This was already done for the previous question. Further reduction of the number of visited states is not possible for BFS.

8. For extremely large mazes, a DFS algorithm would probably be the better choice. BFS will probably run out of memory to store all the possible states and will never return a solution. DFS might end up in an infinite loop, but it also might not. In this case DFS would have a higher chance of returning a solution. In that case I'd choose the DFS approach.

Exercise 3

1. The program finds a path from 0 to 99 and from 0 to 102, using BFS. The found paths are respectively 28091 nodes and 29325 nodes. The program also finds a path from 1 to 0, using DFS. This path has a length of 2 nodes. The other paths are not found, for these the program returns a fatal error, because it does not have enough memory.

2. BFS uses a lot of memory, therefore the solution to why it did not work, is to increase the allocated memory. We multiplied it by a factor of 10. The DFS problem is solved, by making sure that the program does not continuously add 0's and 1's to the end of the stack, because that would mean that all new states would also use 0. The way to solve this, is by making sure that the multiplication uses values bigger than 0, and division only uses values other than 0 and 1.

3. To tackle the problem of finding a path in BFS, we need to change something in the struct State. We've added some integers called index, length and cost. The index is to keep track of which node we are in, length is to keep track of the length of the path and cost is to keep track of the cost of each action. To find the path that BFS explored, we need to backtrack its steps. The function findPath does this. It gets the goal node from the function search and traces each step back by figuring out which move was taken. We've added an index to each move to backtrack which move it actually made, so if one was added, subtracted or that the value was multiplied by three or two etc. We calculated each move by taking the modulo 6 of the index. The value that is returned shows which action was performed on that value. We put the values first in a stack and popped them back in an array for the right path. In the end we print the cost and length of the path as well.

4. We implemented a heap to get the lowest cost to the front of the fringe. Unfortunately, somehow the program ends up in infinite loops when the path is longer than 2 nodes. The loop occurs in the insertFringe function. It is unclear to us why this happened.

5. We did not find this path, because of the flaws from exercise 4.

6. To program an IDS-algorithm, we first made an Depth-limited algorithm. This algorithm first checks if the limit is reached, and if the current position is the goal position. If this is not the case, the children of the position are produced, according to the steps given in the exercise. These are then continuously reproduced, with each recursion decrementing the limit. This is done for all children. If the goal position is found, the path cost is printed and this function returns 1. **Important note:** The path cost is printed multiple times, because the result is calculated with the "and/or" function, instead of the "XOR", therefore multiple costs are given through to the function. After the path cost print, the function returns 1. Here comes the iteration into play. Each iteration has a deeper depth, which is passed on to the Depth-limited function as the limit. If the Depth-limited function does not return 1, the depth is increased by 1, until infinity. When it does return 1, the length of the path is equal to the depth. This is then printed, and the function goes out of the loop. The program then terminates.

7. The iterative deepening program and the UCS (if implemented correctly) find the most optimal solutions. UCS does this quicker than IDS, but both are fast. DFS only finds a solution when the path is very short, because else it uses too much memory. It can be made more effective by reducing the maximum depth. BFS looks at every node, and is therefore very slow, but it does find solutions.

Programming

Program description

This problem is about a very large chess board. Consider a chessboard about the size of 500 x 500. There exist coordinates (x, y) on that board. We are trying to find the shortest path from a starting location with some coordinates to a goal location using A*. We assume that we use a knight to reach a goal.

Problem analysis

To solve this problem, we need to implement the A* algorithm. This algorithm uses roughly the function $f(n) = g(n) + h(n)$. We will need an admissible heuristic for the $h(n)$ part of that function. That function calculates the desirability of each action we take. If the $f(n)$ value is high, then the action is least desirable. If that value if low, we know that action is more desirable. The

algorithm will choose the most desirable value to reach the shortest path if an admissible heuristic is implemented.

Program design

First, the different actions are defined. These are later called upon using a for-loop. Then, a function is described that sees if the locations are valid locations. After this, the heuristics are defined. The first heuristic calculates the Euclidian distance, the second the Manhattan distance. After this comes the actual searching. If the start position is not the goal position, new positions are made by calling on the different actions. If these positions are valid, they are stored in a state and the number of visited states is increased by 1. Then, the heuristic function is applied, and the cost of the path is incremented by one. The cost and the f(n) value are also stored in the state. This state is then enqueued into a heap. This heap sorts the different states based on their f-value. If the state-position is the goal position, the function returns the cost-value and the amount of visited states, and frees the heap. If this is not the case, the front-most value is taken from the heap, and the process starts again.

Program evaluation

The program does not always find a solution, and when it finds the solution, this is not always optimal. However, it is very fast when it finds a solution. The valgrind output is the following:

```
==7270==
==7270== HEAP SUMMARY:
==7270==    in use at exit: 0 bytes in 0 blocks
==7270==  total heap usage: 7 allocs, 7 frees, 2,032 bytes
    allocated
==7270==
==7270== All heap blocks were freed -- no leaks are possible
==7270==
==7270== For counts of detected and suppressed errors, rerun
    with: -v
==7270== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0
    from 0)
```

Program output

Some simple examples give valid results, such as the following:

```
Start location= 0 0
Goal location= 0 0
#visited states = 1
Length shortest path = 0

```

```
Start location = 0 0
Goal location = 0 5
#visited states = 65
Length shortest path = 5

Start location = 40 40
Goal location = 37 38
#visited states = 30
Length shortes path = 5

Start location = 0 0
Goal location = 499 499
#visited staes = 38751
Length shortest path = 2234 (obviously wrong)
```

For more complex problems, the program ends up in an infinite loop either at the end of all possible states, i.e. 499,499, or it just circles around close to the start position.

A*.c

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "heap.h"
#include "a.h"


int actions[8][2] = { /* knight moves */
{-2, -1}, {-2, 1}, {-1, -2}, {-1, 2}, {1,-2}, {1,2}, {2, -1},
    {2, 1}
};

int isValidLocation(int x, int y) {
return (0<=x && x < 500 && 0<= y && y < 500);
}


int heuristic1(int x, int y, int goalX, int goalY) {
int distance = sqrt(pow((goalX-x),2) + pow((goalY-y),2));
return distance/3;
}

int heuristic2(int x, int y, int goalX, int goalY) {
int distance = abs(goalX-x) + abs(goalY-y);
return distance/3;
}


```

```
28          int search (int x, int y, int goalX, int goalY, int cost, Heap
                hp) {
29          State state;
30          int a, h;
31          int visited = 1;
32          while (x != goalX || y != goalY) {
33          for (a=0; a < 8; a++) {
34          int newX = x + actions[a][0];
35          int newY = y + actions[a][1];
36          if (isValidLocation(newX, newY)){
37          visited++;
38          state.x = newX;
39          state.y = newY;
40          state.g = cost++;
41          h = heuristic1(state.x, state.y, goalX, goalY);
42          state.f = state.g + h;
43          if (state.x == goalX && state.y == goalY) {
44          printf("#visited states: %d\n", visited);
45          free(hp.array);
46          return state.g;
47          } else {
48          enqueue(state, &hp);
49          }
50          }
51          }
52          state = removeMin(&hp);
53          x = state.x;
54          y = state.y;
55          cost = state.g;
56          }
57          printf("#visited states: %d\n", visited);
58          free(hp.array);
59          return 0;
60          }
61
62          int main(int argc, char *argv[]) {
63          int x0,y0,x1,y1;
64          Heap hp;
65          hp = makeHeap();
66          do {
67          printf("Start location (x,y) = ");
68          scanf("%d %d", &x0, &y0);
69          } while (!isValidLocation(x0,y0));
70          do {
71          printf("Goal location (x,y) = ");
72          scanf("%d %d", &x1, &y1);
73          } while (!isValidLocation(x1,y1));
74
75          printf("Length shortest path: %d\n", search(x0,y0,x1,y1,0,hp));
76          return 0;
```

```
77          }
```

Heap.c

```
1           #include <stdlib.h>
2           #include <stdio.h>
3           #include <assert.h>
4           #include "heap.h"
5
6           Heap makeHeap () {
7           Heap h;
8           h.array = malloc(1*sizeof(State));
9           assert(h.array != NULL);
10          h.front = 1;
11          h.size = 1;
12          return h;
13          }
14
15          int isEmptyHeap (Heap h) {
16          return (h.front == 1);
17          }
18
19          void heapEmptyError () {
20          printf("heap empty \n");
21          abort();
22          }
23
24          void doubleHeapSize(Heap *hp){
25          int newSize=2*hp->size;
26          hp->array = realloc (hp->array, newSize * sizeof(State));
27          assert(hp->array!=NULL);
28          hp->size=newSize;
29          }
30
31          void enqueue(State s, Heap *hp){
32          int fr = hp->front;
33          if (fr == hp-> size){
34          doubleHeapSize(hp);
35          }
36          hp->array[fr] = s;
37          upHeap(hp,fr);
38          hp->front=fr+1;
39          }
40
41          State removeMin(Heap *hp){
42          State s;
43          if (isEmptyHeap(*hp)){
44          heapEmptyError();
45          }
```

```
46          s = hp->array[1];
47          hp->front--;
48          swap(&(hp->array[1]),&(hp->array[hp->front]));
49          downheap(hp,1);
50          return s;
51          }
52
53          void downheap(Heap *hp, int n) {
54          int fr = hp->front;
55          int indexMax = n;
56          if (fr < 2*n + 1) {
57          return;
58          }
59          if (hp->array[n].f >= hp->array[2*n].f) {
60          indexMax = 2*n;
61          }
62          if (fr < 2*n + 1 && hp->array[indexMax].f >= hp->array[2*n
                +1].f) {
63          indexMax = 2*n + 1;
64          }
65          if (indexMax != n) {
66          swap(&(hp->array[n]), &(hp->array[indexMax]));
67          downheap(hp,indexMax);
68          }
69          }
70
71          void upHeap(Heap *hp, int n){
72          if (n==1){
73          return;
74          }
75          if (hp->array[n].f >= hp->array[n/2].f){
76          return;
77          }
78          if (hp->array[n].f <= hp->array[n/2].f){
79          swap(&(hp->array[n]),&(hp->array[n/2]));
80          upHeap(hp, n/2);
81          }
82          }
83
84          void swap (State *pa, State *pb) {
85          State h = *pa;
86          *pa = *pb;
87          *pb = h;
88          }
```

IDS.c

```
1           #include <stdlib.h>
2           #include <stdio.h>
```

```
3
4         int dls (int start, int goal, int limit, int cost) {
5         int child1, child2, child3, child4, child5, child6;
6         int result;
7         if (limit == 0 && start == goal) {
8         return start;
9         } else if (limit > 0){
10        child1 = start+1;
11        child2= 2*start;
12        child3 = 3*start;
13        child4 = start-1;
14        child5 = start/2;
15        child6 = start/3;
16        result = dls(child1, goal, limit-1, cost+1) || dls(child2, goal,
              limit-1, cost+2) || dls(child3, goal, limit-1, cost+2) ||
              dls(child4, goal, limit-1, cost+1) ||
17        dls(child5, goal, limit-1, cost+3) || dls(child6, goal, limit-1,
              cost+3);
18        if (result != 0 || (goal == start && result == 0)) {
19        printf("path cost is %d\n", cost);
20        return 1;
21        }
22        }
23        return 0;
24        }
25
26        void ids (int start, int goal) {
27        int depth;
28        int result;
29        for (depth = 0; ;depth++){
30        result = dls(start, goal, depth, 1);
31        if (result == 1) {
32        printf("path length is %d\n", depth);
33        break;
34        }
35        }
36        }
37
38        int main(int argc, char *argv[]) {
39        int start, goal;
40        scanf("%d %d", &start, &goal);
41        ids(start,goal);
42        return 0;
43        }
```

Search.c

```
1         #include <stdio.h>
2         #include <stdlib.h>
```

```
#include <string.h>
#include <assert.h>

#include "state.h"
#include "fringe.h"


#define RANGE 1000000


typedef struct Stack{
int *array;
int top;
int size;
} Stack;

Stack newStack(int size){
Stack st;
st.array = malloc(size * sizeof(int));
if(st.array == NULL){
printf("ERROR, array could not be made\n");
exit(-1);
}
st.top = 0;
st.size = size;
return st;
}


void push(int value, Stack *stp){
stp->array[stp->top]= value;
stp->top++;
}

int pop(Stack *stp){
(stp->top)--;
return (stp->array)[stp->top];
}

void findPath(int length, int index, int cost, int start) {
Stack s;
int mode;
int current;
int j = 0;
int buffer;
int *array;
array = malloc(length*sizeof(int));
assert(array != NULL);
s = newStack(length);
while(index > 0) {
```

```
53          mode = index%6;
54          push(mode,&s);
55          index = index/6;
56          if (mode ==0) {
57          index--;
58          }
59          }
60          buffer = s.top;
61
62          while((s.top) >= 0) {
63          array[j] = pop(&s);
64          j++;
65          }
66          printf("%d ", start);
67          current = start;
68          for (j=0; j < buffer; j++) {
69          switch (array[j]) {
70          case 1:
71          current += 1;
72          printf("(+1) -> %d ", current);
73          break;
74          case 2:
75          current *= 2;
76          printf("(*2) -> %d ", current);
77          break;
78          case 3:
79          current *= 3;
80          printf("(*3) -> %d ", current);
81          break;
82          case 4:
83          current -= 1;
84          printf("(-1) -> %d ", current);
85          break;
86          case 5:
87          current /= 2;
88          printf("(/2) -> %d ", current);
89          break;
90          case 6:
91          current /= 3;
92          printf("(/3) -> %d ", current);
93          break;
94          }
95          }
96          printf("\nlength: %d, cost: %d\n", length, cost);
97          }
98
99
100         Fringe insertValidSucc(Fringe fringe, int value, int index, int
                length, int cost) {
101         State s;
```

```
102         if ((value < 0) || (value > RANGE)) {
103         /* ignore states that are out of bounds */
104         return fringe;
105         }
106         s.value = value;
107         s.index = index;
108         s.length = length;
109         s.cost = cost;
110         return insertFringe(fringe, s);
111         }
112
113
114
115         void search(int mode, int start, int goal) {
116         Fringe fringe;
117         State state;
118         int i = 0;
119         int goalReached = 0;
120         int visited = 0;
121         int value;
122         int length = 0;
123         int cost = 0;
124
125         if(mode == PRIO || mode == HEAP) {
126         fringe.size++;
127         }
128
129         fringe = makeFringe(mode);
130         state.value = start;
131         state.length = 0;
132         state.cost = 0;
133         fringe = insertFringe(fringe, state);
134
135         while (!isEmptyFringe(fringe)) {
136         /* get a state from the fringe */
137         fringe = removeFringe(fringe, &state);
138         visited++;
139         /* is state the goal? */
140         value = state.value;
141         i = state.index;
142         length = state.length;
143         cost = state.cost;
144         if (value == goal) {
145         goalReached = 1;
146         break;
147         }
148         /* insert neighbouring states */
149         fringe = insertValidSucc(fringe, value+1, 6*i+1, length+1,
                cost+1); /* rule n->n + 1 */
150         if (value != 0) {
```

```
151         fringe = insertValidSucc(fringe, 2*value, 6*i+2, length+1,
                cost+2); /* rule n->2*n */
152         fringe = insertValidSucc(fringe, 3*value, 6*i+3, length+1,
                cost+2); /* rule n->3*n */
153         fringe = insertValidSucc(fringe, value-1, 6*i+4, length+1,
                cost+1); /* rule n->n - 1 */
154         if (value != 1) {
155         fringe = insertValidSucc(fringe, value/2, 6*i+5, length+1,
                cost+3); /* rule n->floor(n/2) */
156         fringe = insertValidSucc(fringe, value/3, 6*i+6, length+1,
                cost+3); /* rule n->floor(n/3) */
157         }
158         }
159         }
160
161         if (goalReached == 0) {
162         printf("goal not reachable ");
163         } else {
164         printf("goal reached ");
165         findPath(length, i,cost, start);
166         }
167         printf("(%d nodes visited)\n", visited);
168         showStats(fringe);
169         deallocFringe(fringe);
170         }
171
172
173
174         int main(int argc, char *argv[]) {
175         int start, goal, fringetype;
176         if ((argc == 1) || (argc > 4)) {
177         fprintf(stderr, "Usage: %s <STACK|FIFO|HEAP> [start] [goal]\n",
                argv[0]);
178         return EXIT_FAILURE;
179         }
180         fringetype = 0;
181
182         if ((strcmp(argv[1], "STACK") == 0) || (strcmp(argv[1], "LIFO")
                == 0)) {
183         fringetype = STACK;
184         } else if (strcmp(argv[1], "FIFO") == 0) {
185         fringetype = FIFO;
186         } else if ((strcmp(argv[1], "HEAP") == 0) || (strcmp(argv[1],
                "PRIO") == 0)) {
187         fringetype = HEAP;
188         }
189         if (fringetype == 0) {
190         fprintf(stderr, "Usage: %s <STACK|FIFO|HEAP> [start] [goal]\n",
                argv[0]);
191         return EXIT_FAILURE;
```

```
192         }
193
194         start = 0;
195         goal = 42;
196         if (argc == 3) {
197         goal = atoi(argv[2]);
198         } else if (argc == 4) {
199         start = atoi(argv[2]);
200         goal = atoi(argv[3]);
201         }
202
203         printf("Problem: route from %d to %d\n", start, goal);
204         search(fringetype, start, goal);
205         return EXIT_SUCCESS;
206         }
```

Fringe.c

```
1         #include <stdio.h>
2         #include <stdlib.h>
3         #include <stdarg.h>
4         #include <assert.h>
5
6         #include "fringe.h"
7
8
9         void enqueue(Fringe *fringe, State s){
10        int fr = fringe->front;
11        fringe->states[fr] = s;
12        upHeap(fringe,fr);
13        fringe->front=fr+1;
14        }
15
16        State removeMin(Fringe *fringe){
17        State s;
18        s = fringe->states[1];
19        fringe->front--;
20        swap(&(fringe->states[1]),&(fringe->states[fringe->front]));
21        downheap(fringe,1);
22        return s;
23        }
24        /*
25        void doubleHeapSize(Fringe *fringe){
26        int newSize=2*fringe->size;
27        fringe->states = realloc (fringe->states, newSize *
              sizeof(State));
28        assert(fringe->states!=NULL);
29        fringe->size=newSize;
30        }*/
```

```
31
32          void downheap(Fringe *fringe, int n) {
33          int fr = fringe->front;
34          int indexMax = n;
35          if (fr < 2*n + 1) {
36          return;
37          }
38          if (fringe->states[n].cost > fringe->states[2*n].cost) {
39          indexMax = 2*n;
40          }
41          if (fr < 2*n + 1 && fringe->states[indexMax].cost >
                fringe->states[2*n +1].cost) {
42          indexMax = 2*n + 1;
43          }
44          if (indexMax != n) {
45          swap(&(fringe->states[n]), &(fringe->states[indexMax]));
46          downheap(fringe,indexMax);
47          }
48          }
49
50          void upHeap(Fringe *fringe, int n){
51          if (n==1){
52          return;
53          }
54          if (fringe->states[n].cost > fringe->states[n/2].cost){
55          return;
56          }
57          if (fringe->states[n].cost < fringe->states[n/2].cost){
58          swap(&(fringe->states[n]),&(fringe->states[n/2]));
59          upHeap(fringe, n/2);
60          }
61          }
62
63          void swap (State *pa, State *pb) {
64          State h = *pa;
65          *pa = *pb;
66          *pb = h;
67          }
68
69
70          Fringe makeFringe(int mode) {
71          /* Returns an empty fringe.
72          * The mode can be LIFO(=STACK), FIFO, or PRIO(=HEAP)
73          */
74          Fringe f;
75          if ((mode != LIFO) && (mode != STACK) && (mode != FIFO) &&
76          (mode != PRIO) && (mode != HEAP)) {
77          fprintf(stderr, "makeFringe(mode=%d): incorrect mode. ", mode);
78          fprintf(stderr, "(mode <- [LIFO,STACK,FIFO,PRIO,HEAP])\n");
79          exit(EXIT_FAILURE);
```

```
80          }
81          f.mode = mode;
82          f.size = f.front = f.rear = 0; /* front+rear only used in FIFO
                mode */
83          f.states = malloc(MAXF*sizeof(State));
84          if (f.states == NULL) {
85          fprintf(stderr, "makeFringe(): memory allocation failed.\n");
86          exit(EXIT_FAILURE);
87          }
88          f.maxSize = f.insertCnt = f.deleteCnt = 0;
89          return f;
90          }
91
92          void deallocFringe(Fringe fringe) {
93          /* Frees the memory allocated for the fringe */
94          free(fringe.states);
95          }
96
97          int getFringeSize(Fringe fringe) {
98          /* Returns the number of elements in the fringe
99          */
100         return fringe.size;
101         }
102
103         int isEmptyFringe(Fringe fringe) {
104         /* Returns 1 if the fringe is empty, otherwise 0 */
105         return (fringe.size == 0 ? 1 : 0);
106         }
107
108         Fringe insertFringe(Fringe fringe, State s, ...) {
109         /* Inserts s in the fringe, and returns the new fringe.
110         * This function needs a third parameter in PRIO(HEAP) mode.
111         */
112         if (fringe.size == MAXF) {
113         fprintf(stderr, "insertFringe(..): fatal error, out of
                memory.\n");
114         exit(EXIT_FAILURE);
115         }
116         fringe.insertCnt++;
117         switch (fringe.mode) {
118         case LIFO: /* LIFO == STACK */
119         case STACK:
120         fringe.states[fringe.size] = s;
121         break;
122         case FIFO:
123         fringe.states[fringe.rear++] = s;
124         fringe.rear %= MAXF;
125         break;
126         case PRIO: /* PRIO == HEAP */
127         case HEAP:
```

```
128         enqueue(&fringe, s);
129         break;
130         }
131         fringe.size++;
132         printf("%d\n", fringe.size);
133         if (fringe.size > fringe.maxSize) {
134         fringe.maxSize = fringe.size;
135         }
136         return fringe;
137         }
138
139         Fringe removeFringe(Fringe fringe, State *s) {
140         /* Removes an element from the fringe, and returns it in s.
141         * Moreover, the new fringe is returned.
142         */
143         if (fringe.size < 1) {
144         fprintf(stderr, "removeFringe(..): fatal error, empty
                fringe.\n");
145         exit(EXIT_FAILURE);
146         }
147         fringe.deleteCnt++;
148         fringe.size--;
149         switch (fringe.mode) {
150         case LIFO: /* LIFO == STACK */
151         case STACK:
152         *s = fringe.states[fringe.size];
153         break;
154         case FIFO:
155         *s = fringe.states[fringe.front++];
156         fringe.front %= MAXF;
157         break;
158         case PRIO: /* PRIO == HEAP */
159         case HEAP:
160         *s = fringe.states[0];
161         removeMin(&fringe);
162         break;
163         }
164         return fringe;
165         }
166
167         void showStats(Fringe fringe) {
168         /* Shows fringe statistics */
169         printf("#### fringe statistics:\n");
170         printf(" #size      : %7d\n", fringe.size);
171         printf(" #maximum size: %7d\n", fringe.maxSize);
172         printf(" #insertions : %7d\n", fringe.insertCnt);
173         printf(" #deletions : %7d\n", fringe.deleteCnt);
174         printf("####\n");
175         }
```

State.h

```
#ifndef STATE_H
#define STATE_H

/* The type State is a data type that represents a possible state
* of a search problem. It can be a complicated structure, but it
* can also be a simple type (like int, char, ..).
* Note: if State is a structure, make sure that the structure
    does not
*       contain pointers!
*/

typedef struct State {
int value;
int index;
int length;
int cost;
} State;

#endif
```