Artificial Intelligence 1
Lab 1: Agents and Search

Albert Dijkstra (s3390071)
Niels Bugel (s3405583)
Edward Boere (s3005690)

LC[7] Team Supreme

May 7, 2018

Theory assignments

Exercise 1

	Performance measure	Environment	Actuators	Sensors
Reversi computer	The percentage of discs with your colour. How easy your discs can be flipped	An 8x8 unchecked-ered board	An apparatus to place and flip discs	A camera can be used to determine the position of all discs
Robotic lawn mower	Percentage of grass mowed/amount of power used. Time used. Noise	A grass field (with obstacles)	Throttle, braking, mowing, steering	Position locator, camera, engine/battery sensors

	Observable	Agents	Deterministic	Episodic	Static	Discrete
Reversi	Fully	Multi-agent	Deterministic	Episodic	Static	Discrete
Robotic lawn mower	Partially	Multi-Agent	Deterministic	Episodic	Dynamic	Continuous

A suitable architecture for the Reversi computer would be a programmable computer, able to interpret the game state using, for example, a camera. It would need an actuator to move and place disks. An alternative solution would be to play the game on a computer screen.

An architecture for the robotic lawn mower would be a small robot with two drive wheels and one steering wheel. It would need a cutting blade to mow the grass, and sensors to perceive its environment. An onboard computer is necessary to run the agent program.

Exercise 2

1. With DFS, the algorithm goes trough the following tiles 1-2-6-7-3-7-3-.... Due to there being no check to see if it has already been there, it keeps trying to do 7 and 3 forever.

2. We add extra information to each node about whether we have visited it or not:

```

procedure mazeDFS(maze, start, goal):
    stack = []
    stack.push(start)

    foreach node in maze
        node.visited = false

    while stack is not empty:
        loc = stack.pop()
        loc.visited = true
        if loc == goal:
            print "Goal found"
            return
        for move in [N,E,S,W]:
            if allowedMove(loc, move) and neighbour(maze, loc,
                 move).visited == false:
                stack.push(neighbour(maze, loc, move))
    print "Goal not found"
```

3. It is visited in the following order: 1-2-6-7-3-4-8-12-11-15

4. With [N, S, W, E] it is visited in the following order: 1-2-6-5-9-13-14-10-7-3-4-8-12-11-15-16

5. It will always find a position if one exists, but it can take a very long time. This is because all the neighbours are always added and due to it having a FIFO queue, newer nodes cannot overwrite it. At the same time the algorithm does not know which neighbours of a node have already been visited. If it is for example at node 2 it will add both 1 and 6 even though 1 has already been visited. This will slow the process down incredibly, but it will still find a solution at one point.

6. With BFS it is visited in the following order:
1-2-6-1-7-2-5-2-3-6-6-1-9-6-6-1-7-4-7-2-5-7-2-5-2-13-5-7-2-5-7-2-5-2-3-6-8-3-3-6-6-1-9-6-3-6-6-1-9-6-6-1-14-9-9-6-3-6-6-1-9-6-3-6-6-1-9-6-6-1-7-4-7-2-5-12-

4-7-4-7-4-7-2-5-7-2-5-2-13-5-7-2-5-7-4-7-2-5-7-2-5-2-13-5-7-2-5-7-2-5-2-10-13-13-5-13-5-7-2-5-7-4-7 2-5-7-2-5-2-13-5-7-2-5-7-4-7-2-5-7-2-5-2-13-5-7-2-5-7-2-5-2-3-6-8-3-3-6-6-1-9-6-8-11-8-3-3-6-8-3-3-6-8-3-3-6-6-1-9-6-3-6-6-1-9-6-6-1-14-9-9-6-3-6-6-1-9-6-3-6-8-3-3-6-6-1-9-6-3-6-6-1-9-6-6-1-14-9-9-6-3-6-6-1-9-6-3-6-6-1-9-6-6-1-14-14-9-14-9-9-6-14-9-9- 6-3-6-6-1-9-6-3-6-8-3-3-6-6-1-9-6-3-6-6-1-9-6-6-1-14-9-9-6-3-6-6-1-9-6-3-6-8-3-3-6-6-1-9-6-3-6-6-1-9-6-6-1-14-9-9-6-3-6-6-1-9-6-3-6-6-1-9-6-6-1-7-4-7-2-5-12-4-7-4-7-4-7-2-5-7-2-5-2-13-5-7-2-5-12-4-15-12-12-4-7-4-7-4-7-2-5-12-4-7-4-7-4-7-2-5-12-4-7-4-7- 4-7-2-5-7-2-5-2-13-5-7-2-5-7-4-7-2-5-7-2-5-2-13-5-7-2-5-7-2-5-2-10-13-13-5-13-5-7-2-5-7-4-7-2-5-7-2-5-2-13-5-7-2-5-7-4-7-2-5-12-4-7-4-7-4-7-2-5-7-2-5-2-13-5-7-2-5-7-4-7-2-5-7-2-5-2-13-5-7-2-5-7-2-5-2-10-13-13-5-13-5-7-2-5-7-4-7-2-5-7-2-5-2-13-5-7-2-5-7 4-7-2-5-7-2-5-2-13-5-7-2-5-7-2-5-2-10-13-10-13-13-5-10-13-13-5-13-5-7-2-5-10-13-13-5-13-5-7-2-5-7-4-7-2-5-7-2-5-2-13-5-7-2-5-7-4-7-2-5-12-4-7-4-7-4-7-2-5-7-2-5-2-13-5-7-2-5-7-4-7-2-5-7-2-5-2-13-5-7-2-5-7-2-5-2-10-13-13-5-13-5-7-2-5-7-4-7-2-5-7-2-5-2-1 3-5-7-2-5-7-4-7-2-5-12-4-7-4-7-4-7-2-5-7-2-5-2-13-5-7-2-5-7-4-7-2-5-7-2-5-2-13-5-7-2-5-7-2-5-2-10-13-13-5-13-5-7-2-5-7-4-7-2-5-7-2-5-2-13-5-7-2-5-7-4-7-2-5-7-2-5-2-13-5-7-2-5-7-2-5-2-3-6-8-3-3-6-6-1-9-6-8-11-8-3-3-6-8-3-3-6-8-3-3-6-6-1-9-6-3-6-6-1-9-6- 6-1-14-9-9-6-3-6-6-1-9-6-8-11-8-3

7. To reduce the number of states visited, we simply keep track of which positions we have visited already.
   Now we visit: 1-2-6-7-5-3-9-4-13-8-14-12-10-11-15

8. BFS and DFS do not differ all that much in regard to run time, when finding a path in a maze. Obviously there are certain cases where one outperforms the other, but overall the run time is quite similar. However, BFS is always guaranteed to find the shortest path, therefore it might preferable. The drawback is that it requires more memory than DFS. So at some point, the mazes are so large that you run out of memory. In that case you would have to use DFS.

Programming assignments

Program description

In this exercise we consider the following simple mathematical puzzle. Given an integer n from the domain $D = [0..10^6)$, we can perform 6 actions on n:

- $n \rightarrow n+1$: increment n (only allowed if $n+1 \in D$). The cost of this operation is 1 unit.
- $n \rightarrow n-1$: decrement n (only allowed if $n-1 \in D$). The cost of this operation is 1 unit.
- $n \rightarrow n \cdot 2$: double n (only allowed if $n \cdot 2 \in D$). The cost of this operation is 2 units.

- $n \to n \cdot 3$: triple n (only allowed if $n \cdot 3 \in D$). The cost of this operation is 2 units.
- $n \to \lfloor n/2 \rfloor$: integer division of n by two The cost of this operation is 3 units.
- $n \to \lfloor n/3 \rfloor$: integer division of n by three The cost of this operation is 3 units.

Using these actions, find a path from a start value to a goal value. This exercise was divided into 7 questions.

1. Try to use the program to find paths (using BFS) from 0 to 99, from 0 to 100, and from 0 to 102. Also try to find a path from 1 to 0, using DFS. Next, try to find a DFS path from 0 to 1. Explain the results.
2. Modify the program such that DFS and BFS can solve the above mentioned cases. Does this modification have effect on the performance of the algorithm?
3. Modify the program such that the algorithm prints a solution path and its length and cost.
4. Complete the priority queue operations of the abstract data type Fringe.
5. Change the program such that the command line ./search PRIO 0 42 will search for the optimal cost path from 0 to 42 using the UCS algorithm. What is (are) the optimal paths from 0 to 42?
6. Make (from scratch) a program that solves the problem using iterative deepening. You are advised to use explicit recursion.
7. Compare the performance of the four search algorithms for several inputs. What are your conclusions?

As there are multiple questions, instead of using the standard template, we incorporate the program design etc. into the answers where applicable

Exercises

1. BFS can find a path from 0 to 99 and from 0 to 102 but not from 0 to 100. The reason is that the paths from 0 100 and from 0 to 102 are shorter than the path from 0 to 100. Since we are using BFS, the fringe gets approximately 6 times bigger every extra step we need to take, so that is why we run out of memory if we try to find a path from 0 to 100.
For DFS and a path from 1 to 0 it easy, it adds all the actions to the fringe, then removes the last one added. This is $value/3$. Since we are using integer division the result is 0 and we have found our goal. However, this program does

not yet work if we want to start from 0. This is because there is no check which states we have visited. So the program will keeping adding 0s to the fringe until we run out of memory. This is why a path from 0 to 1 cannot be found.

2. The problem here is that we need to somehow make sure that a node is not visited more than once.
Initially we solved this with a solution that used forward checking. This is however a dirty solution(that did not solve all cases), so we made it different when we started with the 3rd exercise. In that exercise we created an array where we can easily check if a node has been visited yet or not. If the node has been visited, then we do not add it to the fringe. This should save a lot of memory in our program since it visits a lot less states. Because of this it is also a lot faster.

3. Here we want to print the path of the solution. Therefore we need to know what the action was that got us to the current node. If we have that, then we can backtrack all the way to our start value.
We make a 2 by RANGE(defined in the program) matrix where each index corresponds to that value. The first row saves the action that was used to get to that index. The second row keeps track of the cost to reach that index. So the cost to the number 42 is saved matrix[42][1]. We then make a print function that simply backtracks it with that matrix and prints the path.

4. For the priority queue operations we need to implement a heap. For this we need to do two things: insert and delete.
For the insert, we insert the node at the first free position and then restore the heap order using upheap(see code). For delete, we replace the root node with the last node and then restore the heap order using downheap(see code).

5. For it to work we needed to do a few more things: add a cost field to the state. And make sure all the functions get the correct values. We also need to make sure that in the insertValidSucc function we only insert a new value if the cost is lower, or if there is nothing at that position.
The optimal path from 0 to 42 is: $0(+1) \rightarrow 1(+1) \rightarrow 2(*3) \rightarrow 6(+1) \rightarrow 7(*3) \rightarrow 21(*2) \rightarrow 42$

6. For this one we go a little more in depth:

Problem analysis

With iterative deepening search, we apply DFS for continually increasing depth limits. So first it does a DFS search for depth 1, then for depth 2 etc. until a certain maximum limit. It is useful to use recursion here, because that is an easy implementation of DFS We need a function in our program that performs DFS and a function calls that function while continually increasing the depth limit.

Program design

In our main function we scan the input. The input consists of three variables: the start number, the goal number and the maximum depth limit.
We then call the function iterativeDeepening. This function contains a for loop that increments the depth until we are at the maximum depth limit or until we find a solution. This function calls for each depth the recursive function DFS. iterativeDeepening returns 1 if it has found a solution and 0 if it has not.
The function DFS is just a simple implementation of DFS. We have a base case where the depth limit has been reached, and six calls to itself that correspond with the six actions we can do. This function returns 1 if it has found a solution and 0 if it has not.

7. we compare the performance of the four algorithms from 0 to 42, 0 to 101 and 5 to 997

0 to 42:

	DFS	BFS	UCS	IDP
length	7	6	6	6
cost	11	9	9	9
nodes visited	999991	43	38	11730

0 to 101:

	DFS	BFS	UCS	IDP
length	75	8	8	8
cost	147	12	12	12
nodes visited	76	222	144	425724

5 to 997:

	DFS	BFS	UCS	IDP
length	253	9	9	9
cost	536	14	14	14
nodes visited	999466	1615	1370	2870290

We see that every algorithm except the one that uses a stack, finds the same solution with regard to cost and length. The algorithm that uses a stack is overall the worst. It does not find the best solution and it has a lot of nodes visited.
After that iterative deepening is the worst, even though the solutions found are a lot better than when using a stack, it still visits a lot of nodes.
The BFS and UCS algorithms are the best and very similar. In these cases only a small difference in nodes visited.

Exercise 4: Knight distance using A*

Problem description

The knights jump problem is a path finding problem. In the game of chess, a knight can move two squares horizontally and one square vertically, or two squares vertically and one square horizontally.

Consider a large 500 x 500 chessboard. A square is a pair (x, y) where $0 \leq x < 500$ and $0 \leq y < 500$. In this exercise we want to find the length of the shortest path (the smallest number of knight moves) from a starting location (x_0, y_0) to a goal location using the A* algorithm.

Problem analysis

The A* algorithm uses a heuristic function to prioritize certain nodes over others. We define function $f(n) = d(n) + h(n)$, where the result of $f(n)$ will be used to compare nodes. $d(n)$ returns the shortest path from the starting node to node n, and $h(n)$ gives an estimated path cost from n to the goal. $h(n)$ must be chosen so that it never overestimates the path cost, it must be admissible.

The goal of this problem is to get closer to the end goal. When finding a path it seems wise to choose the the node closest to the final goal. There are two distance metrics that come to mind:

1. Euclidean Distance
2. Manhattan Distance

The Euclidean distance between two coordinates in a two dimensional grid is the distance between them as the crow flies. The Euclidean distance between two coordinates can be calculated as following:

$$|AB|_E = \sqrt{(A_x - B_x)^2 + (A_y + B_y)^2}$$

We represent our chessboard as a collection of coordinates. Every jump the knight makes has a set length. The knight jumps two squares in the x or y direction, and one square in the other direction. The Euclidean distance of this jump is equal to:

$$\sqrt{2^2 + 1^2} = \sqrt{5}$$

When we divide the distance between two points by the step length we get our first heuristic function:

$$h_e(n) = \frac{|AB|_e}{\sqrt{5}} = \frac{\sqrt{(A_x - B_x)^2 + (A_y + B_y)^2}}{\sqrt{5}} = \sqrt{\frac{(A_x - B_x)^2 + (A_y + B_y)^2}{5}}$$

The Manhattan distance is the distance between two points in a grid based on a strictly horizontal and/or vertical path. Every step in the path has a length of 3, as the knight moves two and one in orthogonal directions. This yields our second heuristic function:

$$h_m(n) = \frac{|AB|_M}{3} = \frac{|A_x - B_x| + |A_y - B_y|}{3} \tag{1}$$

Program design

To implement A* in C, a number of things need to be thought of. We need an efficient way to select the position with the lowest value of $d(n) + h(n)$. This can be achieved using a priority queue. When inserting a position n into the queue, $d(n)$ and $h(n)$ will be calculated, and the state will be inserted such that the queue is ordered with increasing values of $d(n) + h(n)$

The function $d(n)$, the function that returns the shortest path from the starting location to n, can be represented by a 2 dimensional integer matrix. After a new position is enqueued the matrix is updated. If the new path is shorter than the one in the matrix, the old one is replaced. Since our graph is represented by a 2d grid the weight of every edge is equal. This means this 2d matrix is not necessary, but it is left in for completeness sake.

When we have visited a position and located the next position, we can remove it from our set of unused positions. By doing this we exclude the possibility to end up in a loop. We can do this by initializing a 2d array with all positions to "unused", and when we use one, we change the value at its index.

Program extension

The program was extended by including a greedy version of both heuristics. This means $f(n) = h(n)$, so the algorithm does not take into account the path cost already taken. The only change that had to be made was to change the priority function.

Program evaluation

The effective branching factor was calculated by integrating the script from Nestor into the program by making a separate function and calling it from main.

In the table the effective branching factors for heuristic functions $h_e(n)$ and $h_m(n)$ are displayed, using the Euclidean and Manhattan distance respectively. Goal positions at specific depths were found by hand, so not all positions at that depth were tested. The values in the table are therefore only an indication of the mean value. For lower depth, where the goal is closer to the starting position, the effective branching factors lie very close.

	Effective branching factor	
Depth	$h_e(n)$	$h_m(n)$
1	1.999999	1.999999
2	2.192582	2.192582
4	1.606702	1.606702
8	1.707536	1.700026
16	1.311072	1.387802
32	1.216787	1.206158
64	1.117478	1.125552
128	1.065879	1.066827
256	1.035672	Segmentation fault

For both heuristic functions the effective branching factor nears 1. This is because the longer the path, the smaller the influence of the inaccuracies at the end of the path become relative to the full path. Inaccuracies are bound to happen at the end of the path, because when a position is found next to the end goal, not on the end goal, it is indeed the closest. But a knight can only make jumps with a certain shape. This means getting closest is not always ideal when the end goal is almost within reach.

The Euclidean A* algorithm is generally faster than the Manhattan algorithm. But for certain positions, namely positions where the start and goal are at a 45° angle to each other, the Manhattan implementation had a lower branching factor, and reached the goal in less CPU time. However, on the test case with a depth of 256, the program ran out of memory. From this we can conclude that the Manhattan implementation put more items in the priority queue than the Euclidean alternative.

Both implementations of the A* algorithm were significantly faster than the Iterative Deepening Search algorithm provided to check the correctness of the solutions.

When compared to the greedy implementations of both heuristics, the A* version is considerably slower. The most important difference between them is that the greedy implementation can never promise a correct answer. The greedy Euclidean algorithm is fast, and calculates a path that is often the fastest. The greedy Manhattan algorithm is fast, but quite inaccurate.

Program files for exercise 3

search.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

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

#define RANGE 1000000

Fringe insertValidSucc(Fringe fringe, int value, int cost, int action, short paths[RANGE][2])
{
  State s;

  //make sure the value is not out of bounds, and check if the thing we found is more efficient
  if ((value < 0) || (value > RANGE) || (paths[value][0] != 0 && cost >= paths[value][1] && fringe.mode == HEAP) || (paths[value][0] != 0 && fringe.mode != HEAP)) {
    return fringe;
  }

  //insert the actions and costs at the right position in the array
  paths[value][0] = action;
  paths[value][1] = cost;
  s.value = value;
  s.cost = cost;
  return insertFringe(fringe, s);
}

int search(int mode, int start, int goal, short paths[RANGE][2]) {
  Fringe fringe;
  State state;
  int goalReached = 0;
  int visited = 0;
  int value;
  int cost = 0;

  fringe = makeFringe(mode);
  state.value = start;
  state.cost = 0;
  paths[start][0] = 1;
  fringe = insertFringe(fringe, state);
  while (!isEmptyFringe(fringe)) {
    /* get a state from the fringe */

```

```
    fringe = removeFringe(fringe, &state);
    visited++;
    /* is state the goal? */
    value = state.value;
    if (value == goal) {
      goalReached = 1;
      cost = state.cost;
      break;
    }
    /* insert neighbouring states */
    //each time we also check for the goal
    fringe = insertValidSucc(fringe, value+1, state.cost+1, 1, paths);
         /* rule n->n + 1 */
    fringe = insertValidSucc(fringe, 2*value, state.cost+2, 2, paths);
         /* rule n->2*n */
    fringe = insertValidSucc(fringe, 3*value, state.cost+2, 3, paths);
         /* rule n->3*n */
    fringe = insertValidSucc(fringe, value-1, state.cost+1, 4, paths);
         /* rule n->n - 1 */
        fringe = insertValidSucc(fringe, value/2, state.cost+3, 5 +
             value%2, paths); /* rule n->floor(n/2) */
        fringe = insertValidSucc(fringe, value/3, state.cost+3, 7 +
             value%3, paths); /* rule n->floor(n/3) */

  }
  if (goalReached == 0) {
    printf("goal not reachable ");
  } else {
    printf("goal reached ");
  }
  printf("(%d nodes visited)\n", visited);
  showStats(fringe);
  deallocFringe(fringe);
  return cost;
}

//function that prints the path to the goal
int printPath(short paths[RANGE][2], int start, int goal)
{
        //base case
        if(goal == start)
        {
                printf("%d ", goal);
                return 0;
        }

        int lenght = 0;

        //we go backwards through the matrix and print the path
        switch (paths[goal][0])
```

```
	{
		case 0:
			return -1;
		case 1:
			lenght = printPath(paths, start, goal - 1) + 1;
			printf("(+1)-> %d ", goal);
			return lenght;
		case 2:
			lenght = printPath(paths, start, goal / 2) + 1;
			printf("(*2)-> %d ", goal);
			return lenght;
		case 3:
			lenght = printPath(paths, start, goal / 3) + 1;
			printf("(*3)-> %d ", goal);
			return lenght;
		case 4:
			lenght = printPath(paths, start, goal + 1) + 1;
			printf("(-1)-> %d ", goal);
			return lenght;
		case 5:
		case 6:
			lenght = printPath(paths, start, goal * 2 + paths[goal][0] - 5) + 1;
			printf("(/2)-> %d ", goal);
			return lenght;
		case 7:
		case 8:
		case 9:
			lenght = printPath(paths, start, goal * 3 + paths[goal][0] - 7) + 1;
			printf("(/3)-> %d ", goal);
			return lenght;
	}
	return -1;
}

int main(int argc, char *argv[]) {
  int start, goal, fringetype;

  //make a matrix with two rows
  //in the first row, at each index we have the corresponding action we used to get to that index
  //in the second row, at each index we have the cost required to reach that index
  short paths[RANGE][2] = { 0 };

  if ((argc == 1) || (argc > 4)) {
    fprintf(stderr, "Usage: %s <STACK|FIFO|HEAP> [start] [goal]\n", argv[0]);
    return EXIT_FAILURE;
```

```c
  }
  fringetype = 0;

  if ((strcmp(argv[1], "STACK") == 0) || (strcmp(argv[1], "LIFO") == 0)) {
    fringetype = STACK;
  } else if (strcmp(argv[1], "FIFO") == 0) {
    fringetype = FIFO;
  } else if ((strcmp(argv[1], "HEAP") == 0) || (strcmp(argv[1], "PRIO") == 0)) {
    fringetype = HEAP;
  }
  if (fringetype == 0) {
    fprintf(stderr, "Usage: %s <STACK|FIFO|HEAP> [start] [goal]\n", argv[0]);
    return EXIT_FAILURE;
  }

  start = 0;
  goal = 42;
  if (argc == 3) {
    goal = atoi(argv[2]);
  } else if (argc == 4) {
    start = atoi(argv[2]);
    goal = atoi(argv[3]);
  }

  printf("Problem: route from %d to %d\n", start, goal);

  int cost = search(fringetype, start, goal, paths);
  printf("\nlenght: %d, cost: %d \n", printPath(paths, start, goal), cost);
  return EXIT_SUCCESS;
}
```

fringe.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>

#include "fringe.h"

Fringe makeFringe(int mode) {
  /* Returns an empty fringe.
   * The mode can be LIFO(=STACK), FIFO, or PRIO(=HEAP)
   */
  Fringe f;
  if ((mode != LIFO) && (mode != STACK) && (mode != FIFO) &&
```

```
      (mode != PRIO) && (mode != HEAP)) {
    fprintf(stderr, "makeFringe(mode=%d): incorrect mode. ", mode);
    fprintf(stderr, "(mode <- [LIFO,STACK,FIFO,PRIO,HEAP])\n");
    exit(EXIT_FAILURE);
  }
  f.mode = mode;
  f.size = f.front = f.rear = 0; /* front+rear only used in FIFO mode */
  f.states = malloc(MAXF*sizeof(State));
  if (f.states == NULL) {
        fprintf(stderr, "makeFringe(): memory allocation failed.\n");
    exit(EXIT_FAILURE);
  }
  f.maxSize = f.insertCnt = f.deleteCnt = 0;
  return f;
}

void deallocFringe(Fringe fringe) {
  /* Frees the memory allocated for the fringe */
  free(fringe.states);
}

int getFringeSize(Fringe fringe) {
  /* Returns the number of elements in the fringe
   */
  return fringe.size;
}

int isEmptyFringe(Fringe fringe) {
  /* Returns 1 if the fringe is empty, otherwise 0 */
  return (fringe.size == 0 ? 1 : 0);
}

Fringe insertFringe(Fringe fringe, State s) {
  /* Inserts s in the fringe, and returns the new fringe.
   * This function needs a third parameter in PRIO(HEAP) mode.
   */

  if (fringe.size == MAXF) {
    fprintf(stderr, "insertFringe(..): fatal error, out of memory.\n");
    exit(EXIT_FAILURE);
  }
  fringe.insertCnt++;
  switch (fringe.mode) {
          case LIFO: /* LIFO == STACK */
          case STACK:
               fringe.states[fringe.size] = s;
               break;
          case FIFO:
               fringe.states[fringe.rear++] = s;
               fringe.rear %= MAXF;
```

```
              break;
         case PRIO: /* PRIO == HEAP */
         case HEAP:
         {
              //insert at first free position in the heap
              int currentIndex = fringe.size;
              currentIndex++;
              fringe.states[currentIndex] = s;
              //restore heap order with upheap
              while(currentIndex > 0 &&
                  fringe.states[currentIndex/2].cost >
                  fringe.states[currentIndex].cost)
              {
                     //swap
                     State z = fringe.states[currentIndex/2];
                     fringe.states[currentIndex/2] =
                         fringe.states[currentIndex];
                     fringe.states[currentIndex] = z;

                     currentIndex = currentIndex/2;
              }
              break;
         }
  }
  fringe.size++;
  if (fringe.size > fringe.maxSize) {
    fringe.maxSize = fringe.size;
  }
  return fringe;
}

Fringe removeFringe(Fringe fringe, State *s) {
  /* Removes an element from the fringe, and returns it in s.
   * Moreover, the new fringe is returned.
   */

  if (fringe.size < 1) {
    fprintf(stderr, "removeFringe(..): fatal error, empty fringe.\n");
    exit(EXIT_FAILURE);
  }
  fringe.deleteCnt++;
  fringe.size--;
  switch (fringe.mode) {
  case LIFO: /* LIFO == STACK */
  case STACK:
    *s = fringe.states[fringe.size];
    break;
  case FIFO:
    *s = fringe.states[fringe.front++];
    fringe.front %= MAXF;
```

```
    break;
  case PRIO: /* PRIO == HEAP */
  case HEAP:
      //get the root of the heap and swap it with the last node in the
          heap
    *s = fringe.states[1];
    fringe.states[1] = fringe.states[fringe.size+1];
    int currentIndex = 1;
    int maxIndex = 1;

    //restore heap order using downheap
    while(2*currentIndex+1 <= fringe.size)
    {
            if(fringe.states[currentIndex*2].cost <
                fringe.states[currentIndex].cost)
                maxIndex = 2*currentIndex;

            if(fringe.size > 2 * currentIndex + 1 &&
                fringe.states[maxIndex].cost >
                fringe.states[2*currentIndex + 1].cost)
                maxIndex = 2* currentIndex + 1;

            if(currentIndex == maxIndex)
                break;

            //swap
            State z = fringe.states[maxIndex];
            fringe.states[maxIndex] = fringe.states[currentIndex];
            fringe.states[currentIndex] = z;

            currentIndex = maxIndex;
      }
    break;
  }
  return fringe;
}

void showStats(Fringe fringe) {
  /* Shows fringe statistics */
  printf("#### fringe statistics:\n");
  printf(" #size       : %7d\n", fringe.size);
  printf(" #maximum size: %7d\n", fringe.maxSize);
  printf(" #insertions : %7d\n", fringe.insertCnt);
  printf(" #deletions : %7d\n", fringe.deleteCnt);
  printf("####\n");
}
```

fringe.h

```c
#ifndef FRINGE_H
#define FRINGE_H
#include <stdarg.h>

#include "state.h"

#define MAXF 500000 /* maximum fringe size */

#define LIFO 1
#define STACK 2
#define FIFO 3
#define PRIO 4
#define HEAP 5

typedef struct Fringe {
  int mode;     /* can be LIFO(STACK), FIFO, or PRIO(HEAP)     */
  int size;     /* number of elements in the fringe            */
  int front;    /* index of first element in the fringe (FIFO mode) */
  int rear;     /* index of last element in the fringe (FIFO mode) */
  State *states; /* fringe data (states)                        */
  int insertCnt; /* counts the number of insertions            */
  int deleteCnt; /* counts the number of removals (deletions) */
  int maxSize;  /* maximum size of the fringe during search    */
} Fringe;

Fringe makeFringe(int mode);
/* Returns an empty fringe.
 * The mode can be LIFO(=STACK), FIFO, or PRIO(=HEAP)
 */

void deallocFringe(Fringe fringe);
/* Frees the memory allocated for the fringe */

int getFringeSize(Fringe fringe);
/* Returns the number of elements in the fringe
 */

int isEmptyFringe(Fringe fringe);
/* Returns 1 if the fringe is empty, otherwise 0 */

Fringe insertFringe(Fringe fringe, State s);
/* Inserts s in the fringe, and returns the new fringe.
 * This function needs a third parameter in PRIO(HEAP) mode.
 */

Fringe removeFringe(Fringe fringe, State *s);
/* Removes an element from the fringe, and returns it in s.
 * Moreover, the new fringe is returned.
 */
```

```
void showStats(Fringe fringe);
/* Shows fringe statistics */

#endif
```

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 {
  int value;
  int cost;
} State;

#endif
```

iterdeep.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

//function that does depth first search for current depth
int DFS(int start, int goal, int limit, int* cost, int* length, int*
    visited)
{
       *visited += 1;
       //base case
       if(limit == 0)
       {
              return (start == goal);
       }
       int currentLength = *length;
       int currentCost = *cost;

       //check each of the actions
       //each time we reset the cost and length
       *cost = currentCost + 1;
       *length = currentLength + 1;
       if(DFS(start+1, goal, limit - 1, cost, length, visited))
       {
```

```
			return 1;
		}

		*cost = currentCost + 2;
		*length = currentLength + 1;
		if(DFS(start*2, goal, limit - 1, cost, length, visited))
		{
			return 1;
		}

		*cost = currentCost + 2;
		*length = currentLength + 1;
		if(DFS(start*3, goal, limit - 1, cost, length, visited))
		{
			return 1;
		}

		*cost = currentCost + 1;
		*length = currentLength + 1;
		if(DFS(start-1, goal, limit - 1, cost, length, visited))
		{
			return 1;
		}

		*cost = currentCost + 3;
		*length = currentLength + 1;
		if(DFS(start/2, goal, limit - 1, cost, length, visited))
		{
			return 1;
		}

		*cost = currentCost + 3;
		*length = currentLength + 1;
		if(DFS(start/3, goal, limit - 1, cost, length, visited))
		{
			return 1;
		}

		return(start == goal);
}

//function that loops through all the depths <= limit
int iterativeDeepening(int start, int goal, int limit, int* cost, int* length, int* visited)
{
		for (int i = 0; i <= limit; i++)
		{
			*cost = 0;
			*length = 0;
			if(DFS(start, goal, i, cost, length, visited))
```

```
                {
                        return 1;
                }
        }

        return 0;

}

int main(int argc, char *argv[]) {
  int start, goal, limit, cost, length;
  int visited = 0;

  //make sure the input is valid
  if (argc != 4) {
    fprintf(stderr, "Usage: %s [start] [goal] [depth]\n", argv[0]);
    return EXIT_FAILURE;
  }

  //get input
  start = atoi(argv[1]);
  goal = atoi(argv[2]);
  limit = atoi(argv[3]);

  printf("Problem: route from %d to %d\n", start, goal);

  //check if we found a solution for the max limit
  if(iterativeDeepening(start, goal, limit, &cost, &length, &visited))
  {
          printf("Cost: %d\nLength: %d\nNodes visited: %d\n", cost,
              length, visited);
  }
  else
  {
          printf("No solution found for maximum limit: %d\n", limit);
  }


  return EXIT_SUCCESS;
}
```

Program files for exercise 4

astar.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include "queuesPos.h"

#define N 500 /* N times N chessboard */

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

/* The array isVisited indicates the positions that already have been visited.
 * The array shortestPath contains the shortest path from the starting position to [x][y].
 * They are used in many of the functions of the program.
 * They have been declared as global variables to avoid having to pass them on in the function parameters.
 */
int isVisited[N][N];
int shortestPath[N][N];

void initialize(){
        /* 0 in isVisited indicates that you haven't been there, 1 indicates that you have been there.
         * the shortestPath array is initialized with a large number resembling "infinite".
         */
        for (int x = 0 ; x<N ; x++){
                for (int y = 0 ; y<N ; y++){
                        isVisited[x][y] = 0;
                        shortestPath[x][y] = 999999;
                }
        }
}

/* This function is copied from Nestor, and is used to calculated effective branching factor */
double effectiveBranchingFactor(unsigned long states, int d) {
  /* approximates such that N=\Sum+{i=1}^{d} b^i */
  double lwb = 1;
  double upb = pow(states, 1.0/d);
  while (upb - lwb > 0.000001) {
    double mid = (lwb + upb) / 2.0;
    /* the following test makes use of the geometric series */
```

```
    if (mid*(1-pow(mid, d))/(1-mid) <= states) {
      lwb = mid;
    } else {
      upb = mid;
    }
  }
  return lwb;
}

/* The function min returns the smallest of two integers.*/
int min(int x, int y){
        return ((x) <= (y)) ? (x) : (y);
}

/*Returns 1 if the location (x,y) is on the board, and has not been visited before.*/
int isValidLocation(int x, int y) {
  return (0<=x && x < N && 0<= y && y < N && isVisited[x][y] == 0);
}

/* This heuristic calculates the euclidian distance from (x,y) to the goal (xG, yG),
 * and divides it by the length of a single step (sqrt(2^2 + 1^2) = sqrt(5).
 * Since a knight has to stand in the boxes on the chessboard, its path is always longer than
 * the most direct path ignoring the boxes.
 */
double heuristicEuclidianDistance(int x, int y, int xG, int yG) {
        return sqrt((pow(xG-x,2) + pow(yG-y,2))/5);
}

/* This function returns the Manhattan distance between two points and divides it by the Manhattan step length (3) */
int heuristicManhattanDistance(int x, int y, int xG, int yG) {
        return ((abs(xG-x) + abs(yG-y))/3);
}

/* This function is used in the function that inserts Positions into the priority queue.
 * It has a parameter option so the user can choose between different heuristics */
int priority(Position s, Position t, int xG, int yG, int option){
        switch(option){
                case 0: /* A* Euclidean */
                        if (shortestPath[s.x][s.y] + heuristicEuclidianDistance(s.x, s.y, xG, yG) < shortestPath[t.x][t.y] + heuristicEuclidianDistance(t.x, t.y, xG, yG)){
                            return 1; //s is closer to goal than t, so
```

```
                    s gets priority
                }
                if (shortestPath[s.x][s.y] +
                    heuristicEuclidianDistance(s.x, s.y, xG, yG) >
                    shortestPath[t.x][t.y] +
                    heuristicEuclidianDistance(t.x, t.y, xG, yG)){
                    return -1;
                    //t is closer to goal than s, so t gets
                        priority
                }

        case 1: /* A* Manhattan */
                if (shortestPath[s.x][s.y] +
                    heuristicManhattanDistance(s.x, s.y, xG, yG) <
                    shortestPath[t.x][t.y] +
                    heuristicManhattanDistance(t.x, t.y, xG, yG)){
                    return 1;
                    //s is closer to goal than t, so s gets
                        priority
                }
                if (shortestPath[s.x][s.y] +
                    heuristicManhattanDistance(s.x, s.y, xG, yG) >
                    shortestPath[t.x][t.y] +
                    heuristicManhattanDistance(t.x, t.y, xG, yG)){
                    return -1;
                    //t is closer to goal than s, so t gets
                        priority
                }

        case 2: /* Greedy Euclidean */
                if (heuristicEuclidianDistance(s.x, s.y, xG, yG) <
                    heuristicEuclidianDistance(t.x, t.y, xG, yG)){
                    return 1; //s is closer to goal than t, so
                        s gets priority
                }
                if (heuristicEuclidianDistance(s.x, s.y, xG, yG) >
                    heuristicEuclidianDistance(t.x, t.y, xG, yG)){
                    return -1;
                    //t is closer to goal than s, so t gets
                        priority
                }

        case 3: /* Greedy Manhattan */
                if (heuristicManhattanDistance(s.x, s.y, xG, yG) <
                    heuristicManhattanDistance(t.x, t.y, xG, yG)){
                    return 1;
                    //s is closer to goal than t, so s gets
                        priority
                }
                if (heuristicManhattanDistance(s.x, s.y, xG, yG) >
```

```
                          heuristicManhattanDistance(t.x, t.y, xG, yG)){
                              return -1;
                              //t is closer to goal than s, so t gets
                                  priority
                          }

        }
        return 0;
        /* t and s are equally far away so their path length will be
            identical */
}

/*Inserts a Position into a list, ordered by increasing values of
    f(n)=d(n)+h(n) */
List insertUnique(List li, Position s, int xG, int yG, int option){
        /*Empty list, so add item*/
        if(li == NULL){
                //printf("added %d at end of list\n", s.steps);
                return addItem(s,li);
        }
        /*Compares the current Position in the list, and the Position to
            be added*/
        int prio = priority(s, li->item, xG, yG, option);
        switch(prio){

                /* Add the new Position as its distance to endgoal is
                    smaller*/
                case 1:
                        //printf("added steps %d to queue\n",s.steps);
                        return addItem(s,li);

                /*They are duplicates, and t was added before s, so t has
                    less steps, don't add it*/
                case 0:
                        //printf("duplicate\n");
                        return li;

                /* New Position is further away from endgoal, so repeat
                    function at next place in list*/
                case -1:
                        li->next = insertUnique(li->next, s, xG, yG,
                            option);
        }
        /*Its been added, not possible to be here, but its here to stop
            the compiler from complaining */
        return li;
}

int knightAStar(int x0, int y0, int xG, int yG, int option, unsigned
    long *numberOfStates){
```

```c

        Position pos;
        Position pos0 = {     /* The starting position*/
                .x = x0,
                .y = y0,
                .steps = 0
        };

        Queue q = newEmptyQueue();
        q.list = insertUnique(q.list, pos0, xG, yG, option); /* Add starting position to priority queue */

        while (!isEmptyQueue(q)){
                pos = dequeue(&q); //pos with minimal value of d+h because priority queue
                if (pos.x == xG && pos.y == yG){ /* The end goal is reached */
                        return pos.steps;
                }
                isVisited[pos.x][pos.y] = 1; /* Remove the position from all positions that can be enqueued */
                for (int act = 0; act < 8; act++) { /* Make 8 new positions from the dequeued position*/
                        Position new = {
                                .y = pos.y + actions[act][0],
                                .x = pos.x + actions[act][1],
                                .steps = pos.steps + 1
                        };
                        if (isValidLocation(new.x, new.y)){ /* Check if the new position is a viable location*/
                                shortestPath[new.x][new.y] = min(new.steps, shortestPath[new.x][new.y]); /* Update the shortest path array */
                                *numberOfStates = *numberOfStates + 1; /* Increment the number of states made, this is needed to calculate branching factor */
                                q.list = insertUnique(q.list, new, xG, yG, option); /* Insert it in the priority queue */

                        }
                }
        }
        freeQueue(q);

        return 999999; /* This value resembles infinite, if this is returned, no path got to the end. */
}

```

```c
int main(int argc, char *argv[]) {
	printf("\e[1;1H\e[2J"); fflush(stdout); /* Some dodgy code to clear the screen (throws compiler warnings) */
	int x0,y0, xG,yG, option;
	while (1){
		initialize(); /* Every time we want to calculate something, we need to reset the arrays */
		unsigned long numberOfStates = 0;
		printf("Which of the following heuristics would you like to use?:\n\n 0 A* Euclidian Distance\n 1 A* Manhattan Distance\n 2 Greedy Euclidian Distance\n 3 Greedy Manhattan Distance\n 4 None, quit\n\n: "); fflush(stdout);
		scanf("%d", &option);
		if (option == 4){
			break;
		}
		do {
			printf("\nStart location (x,y) = "); fflush(stdout);
			scanf("%d %d", &x0, &y0);
		} while (!isValidLocation(x0,y0));
		do {
			printf("Goal location (x,y) = "); fflush(stdout);
			scanf("%d %d", &xG, &yG);
		} while (!isValidLocation(xG,yG));

		clock_t t = clock(); /* The time.h library is used to calculate the CPU time necessary for the calculation */
		int path = knightAStar(x0,y0, xG,yG, option, &numberOfStates);
		t = clock() - t;
		double time_taken = ((double)t)/CLOCKS_PER_SEC; /* Time taken in seconds */

		if (option == 2 || option == 3){ /* The greedy functions return an approximation, not guaranteed to be shortest. I wouldnt want to lie to you */
			printf("\nApproximated shortest path: %d\n", path); fflush(stdout);
		} else {
			printf("\nShortest path length: %d\n", path); fflush(stdout);
		}

		printf("Effective Branching factor: %f\n", effectiveBranchingFactor(numberOfStates, path)); fflush(stdout);
		printf("Approximate CPU time: %.5fs\n\n", (time_taken));
```

```
                fflush(stdout);
            printf(" 0  Again\n 1  Quit\n\n: "); fflush(stdout);

            scanf("%d",&option);
            if (option == 1) {
                break;
            }
            printf("\n"); fflush(stdout);
        }
        return 0;
}
```

queuesPos.c

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include "queuesPos.h"

/* We use the functions on lists as defined in 1.3 of the lecture notes.
    */

List newEmptyList() {
        return NULL;
}

int isEmptyList (List li) {
        return ( li==NULL );
}

void listEmptyError() {
        printf("list empty\n");
        abort();
}

List addItem(Position s, List li) {
        List newList = malloc(sizeof(struct ListNode));
        assert(newList!=NULL);
        newList->item = s;
        newList->next = li;
        return newList;
}

Position firstItem(List li) {
        if ( li == NULL ) {
                listEmptyError();
        }
        return li->item;
}
```

```
List removeFirstNode(List li) {
	List returnList;
	if ( li == NULL ) {
		listEmptyError();
	}
	returnList = li->next;
	free(li);
	return returnList;
}

void freeList(List li) {
	List li1;
	while ( li != NULL ) {
		li1 = li->next;
		free(li);
		li = li1;
	}
	return;
}

/* We define some functions on queues, based on the definitions in 1.3.1 of the
 * lecture notes. Integers are replaced by positions, and enqueue has output type void here. */

Queue newEmptyQueue () {
	Queue q;
	q.list = newEmptyList();
	q.lastNode = NULL;
	return q;
}

int isEmptyQueue (Queue q) {
	return isEmptyList(q.list);
}

void emptyQueueError () {
	printf("queue empty\n");
	exit(0);
}

void enqueue (Position s, Queue *qp) {
	if ( isEmptyList(qp->list) ) {
		qp->list = addItem(s,NULL);
		qp->lastNode = qp->list;
	} else {
		(qp->lastNode)->next = addItem(s,NULL);
		qp->lastNode = (qp->lastNode->next);
	}
```

```c
}

Position dequeue(Queue *qp) {
        Position s = firstItem(qp->list);
        qp->list = removeFirstNode(qp->list);
        if ( isEmptyList(qp->list) ) {
                qp->lastNode = NULL;
        }
        return s;
}

void freeQueue (Queue q) {
        freeList(q.list);
}
```

queuesPos.h

```c
/* queues.h, 24 March 2016 */

#ifndef QUEUESPOS_H
#define QUEUESPOS_H

/* First the type definitions. */

typedef struct Position { /* a position contains a time and the
    coordinates */
        int x, y, steps;
} Position;

typedef struct ListNode *List; /* List is the type of lists of positions
    */

struct ListNode {
        Position item;
        List next;
};

typedef struct Queue { /* a queue is a list and a pointer to the last
    node */
        List list;
        List lastNode;
} Queue;

/* Now the declaration of the functions that are defined in queues.c
 * and are to be used outside it, e.g. in hedge.c */

Queue newEmptyQueue ();
void freeQueue (Queue q);
int isEmptyQueue (Queue q);
```

```
void enqueue (Position s, Queue *qp);
Position dequeue(Queue *qp);
List addItem(Position s, List li);

#endif
```