Theory

Exercise 1

	Reversi	Robotic lawn mower
Observable	Full	Partial
Deterministic	Deterministic	Stochastic
Episodic	Sequential	Episodic
Static	Semidynamic	Dynamic
Discrete	Discrete	Continuous
Single-agent	Multi-agent	Single-agent

Reversi is fully observable, as you can see what happens on the board. The next state of the environment is completely determined by the current state and the action executed by the agent, so it is deterministic. Because you take turns, it is sequential. The environment itself does not change with the passage of time but the agent's performance score does, so it is semidynamic. Reversi has a finite amount of states, so it is discrete. And you play it with 2 people, so it is multi-agent.
A robotic lawn mower is partially observable, . It is stochastic, as the current state does not matter for a robotic lawn mower. It is episodic, because it does not matter what happens before it, it will decide for every single state what happens. Because a robotic lawn mower constantly asks itself what to do every time, it is dynamic. Because anything can happen, it is continuous. Because only the robot decides what it does (it is autonomous), it is single-agent.
Explain which agent architecture is best for each domain? (do they mean simple reflex, model-based reflex agents etc?)

Exercise 2

1. It is not possible to use mazeDFS() to find a path from the red square to the yellow square because the algorithm will get stuck between square 7 and square 3. Because the algorithm looks at the squares in the order NESW, square 7 will always go to south, which is square 3, before anything else as both north and east are walls. Square 3 evaluates north first, so it will go there, which is square 7. This ends up in a loop.

2. To fix this problem, we add the visiting of squares to the (psuedo-)code. For example, if you're at square 3 after following the path that goes from square 1 to square 2 to square 7 to square 3, all these squares are 'visited'. So a new condition in the 'if allowedMove(loc, move): stack.push(neighbour(maze, loc, move))' should be that you're not allowed to go to a square that is already visited.

3. We start at square 1. At square 1 we only have 1 option, namely square 2. So we go to square 2. At square 2, we have 1 option, namely square 6 north (as square 1, west, is already visited). Because north is first, we go to square 6. At square 6, we have 2 options, square 7 east and square 5

west. Because square 7 is first, we go to square 7. At square 7, going east to square 3 is the first option, as both north and east are walls. At square 4, square 8, going east, is the first option. After that square 12 is the first option, followed by going to square 11 and finally our goal square 15. So, the path is:
$1 \Rightarrow 2 \Rightarrow 6 \Rightarrow 7 \Rightarrow 3 \Rightarrow 4 \Rightarrow 8 \Rightarrow 12 \Rightarrow 11 \Rightarrow 15$

4. We start at square 1. At square 1 we only have 1 option, namely square 2. So we go to square 2. At square 2, we have 1 option, namely square 6 north. Because north is first, we go to square 6. At square 6, we have 2 options (ignorng the visited state), square 7 east and square 5 west. Because the order is now NSWE, square 5 is first, we go to square 5. At square 5, going north to square 9 is the first option. At square 9, square 13, going north, is the first option. After that square 14 is the first option, followed by going to square 10. At this point, we have no options, as all squares around square 10 are either walls or already visited squares. Because of the stack, we go all the way back to square 7, which was still in the stack. At square 7, we follow the same path as in the exercise above, which is, going to square 3, then 4, then 8, then 12, then 11, and finally goal square 15. So, the path is:
$1 \Rightarrow 2 \Rightarrow 6 \Rightarrow 5 \Rightarrow 9 \Rightarrow 13 \Rightarrow 14 \Rightarrow 10 \Rightarrow 7 \Rightarrow 3 \Rightarrow 4 \Rightarrow 8 \Rightarrow 12 \Rightarrow 11 \Rightarrow 15$

5. Yes, this algorithm will always find a solution. BFS works in such a way that it goes through every state, so in the end the desired square will be found.

6. We start at square 1. At this square, we only have the one option, square 2. At this square, we again only have one option, square 6. Here, we have two options now, square 5 and square 7. Because they are put in the queue in order NESW, square 7 is put in the queue earlier, so we go to this square first. After that, square 5 is in the queue. Then we go to the square that square 7 goes to, which is square 3. We go back to the options that square 5 has, and this one chooses square 9. Again, we go back to square 3, which first option is 4. Then again, we go back to square 9, which first option is 13. Then, square 4's first option is 8. We do this until the one path reaches square 10, because then it has no more options. At that point, we return fully to square 11, who only needs one more step to reach the final goal, square 15. So, the path is:
$1 \Rightarrow 2 \Rightarrow 6 \Rightarrow 7 \Rightarrow 5 \Rightarrow 3 \Rightarrow 9 \Rightarrow 4 \Rightarrow 13 \Rightarrow 8 \Rightarrow 14 \Rightarrow 12 \Rightarrow 10 \Rightarrow 11 \Rightarrow 15$

7. No? i dont know how

8. Depends.

Exercise 3

a. (DFS = LIFO, BFS = FIFO)

```
s3499952@fwn-bborg-edu-74-55:~/AI1/find42$ ./search FIFO 0 99
Problem: route from 0 to 99
goal reached (28091 nodes visited)
#### fringe statistics:
#size        :  126169
#maximum size:  126170
#insertions  :  154260
#deletions   :   28091
####


s3499952@fwn-bborg-edu-74-55:~/AI1/find42$ ./search FIFO 0 100
Problem: route from 0 to 100
insertFringe(..): fatal error, out of memory.
s3499952@fwn-bborg-edu-74-55:~/AI1/find42$ ./search FIFO 0 102
Problem: route from 0 to 102
goal reached (29325 nodes visited)
#### fringe statistics:
#size        :  132124
#maximum size:  132125
#insertions  :  161449
#deletions   :   29325
####
s3499952@fwn-bborg-edu-74-55:~/AI1/find42$ ./search LIFO 0 1
Problem: route from 0 to 1
insertFringe(..): fatal error, out of memory.

LIFO uses a stack, so the last operation will be placed on the top of the stack every time.

s3499952@fwn-bborg-edu-74-55:~/AI1/find42$ ./search LIFO 1 0
Problem: route from 1 to 0
goal reached (2 nodes visited)
#### fringe statistics:
#size        :       5
#maximum size:       6
#insertions  :       7
#deletions   :       2
####

The last operation is division by 3, so when we begin with 1, $1 / 3 = 0$ and we have reache
```

b. if goal is reached or something like that

Exercise 4

Programming

Program description

We are asked to write an A^* algorithm to find the shortest path between 2 squares on a 500-by-500 squares chess board, using only steps the knight piece can take. Also, we are asked to find 2 heuristics for A^*, both of which ought to be admissible.

Problem analysis

We are given an iterative deepening algorithm, implemented in C, to check values for certain inputs. This algorithm however, is terribly slow. A^* is definitely faster, but we need to find suitable heuristics, which proved quite a challenge. The first we decide to use, is the Straight-Line-Distance (SDL). The other we will use, is the Manhattan Distance.
SDL is the most straightforward out of the two. The knight can not always move into a straight line to the goal. This is, because it is locked to a grid, and a specific set of moves. The Manhattan Distance does keep this in mind. By calculating the amount of squares that the knight has to move horizontally and vertically, and dividing it by the amount of square it can maximally travel into that direction, we get an under-estimate for the amount of steps the knight has yet to take.

Program design

Since we know how to implement A^* for a general heurisic, we do so first. We create a function `heur(..)`, which will return the value of the heuristic function we choose to use. Therefore, we can worry about finding suitable heuristic functions later. To get the lowest time complexity, we use a heap structure to implement the required priority queue, since this is $\mathcal{O}(n \log n)$, whereas an list based priority queue is $\mathcal{O}(n^2)$.
To select between both heuristic functions, we create a macro `H`, which can be set to either 1 or 0, representing SLD and Manhattan Distance respectively.
To implement SLD, we can use the Pythagorean Theorem to find the distance between the knights current position and the goal position in squares. Similarly, we can find the distance the knight travels in every move (1 in one direction, and 2 in the other). Mathmatically, we can find the straight line distance in steps by solving

$$\frac{\sqrt{x_d^2 + y_d^2}}{\sqrt{5}}$$

Where x_d and y_d are the horizontal and vertical distance between the knight and the goal.
For the Manhattan Distance, we can simply add up x_d and y_d and divide that

by 4. This is because the maximum amount of steps the knight can take in either direction is 2 (just not both in one move).

Program evaluation

When running the program for both heuristics, we get (almost) identical results. With either, the program terminates within the blink of an eye, which is significantly better than what we got before using iterative deepening. Also, for the largest case, $(0, 0) \rightarrow (499, 499)$, the program gives the correct output, and visits all 250000 nodes. This is to be expected, since this distance is maximal, and the program will thus consider every other node first since they are all closer to the starting point. For more random inputs, the Straight Line distance seems to perform better, as it visits fewer nodes than the Manhattan Distance.

Program output

What follows is an example output on this input:

```
Start location (x,y) = 0 0
Goal location (x,y)  = 499 499
```

And

```
Start location (x,y) = 238 421
Goal location (x,y)  = 23 142
```

The corresponding output is:

SLD	Manhattan

```
Length shortest path: 334
#visited states: 250000
------
Length shortest path: 166
#visited states: 198267
```

```
Length shortest path: 334
#visited states: 250000
------
Length shortest path: 166
#visited states: 227898
```

Program files

idknight.c

```
/*
* File can be compiled with the command "make"
*/

```

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

#include "heap.h"

#define N 500  /* N times N chessboard */
#define H 0    /* set to either 0 or 1 for different heuristics */

int actions[8][2] = { /* knight moves */
{-2, -1}, {-2, 1}, {-1, -2}, {-1, 2}, {1,-2}, {1,2}, {2, -1},
    {2, 1}
};
int costShortestPath[N][N];
unsigned long statesVisited = 0;

int abs(int x) {
return (x < 0 ? -x : x);
}

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

void initialize() {
int r, c;
for (r=0; r < N; r++) {
for (c=0; c < N; c++) {
costShortestPath[r][c] = 999999; /* represents infinity */
}
}
}

int knightDLS(int cost, int limit, int row, int column, int
    rowGoal, int columnGoal) {
int act;
statesVisited++;
if (row == rowGoal && column == columnGoal) {
return 1; /* goal reached */
}
if (cost == limit || cost >= costShortestPath[row][column]) {
return 0; /* limit reached, or we've been here before via a
    'cheaper' path */
}
costShortestPath[row][column] = cost;
for (act=0; act < 8; act++) {
int r = row + actions[act][0];
int c = column + actions[act][1];
if (isValidLocation(r, c) && knightDLS(cost+1, limit, r, c,
    rowGoal, columnGoal) == 1) {
```

```
return 1;
}
}
return 0;
}

int isInBounds(int x, int y){
return !(x >= N || x < 0 || y >= N || y < 0);
}

int heur(int startX, int startY, int goalX, int goalY){
int x = goalX - startX;
int y = goalY - startY;
if(!H){
// Straight Line Distance in steps. One step = sqrt(5)
return (int)sqrt((x*x+y*y)/5.0);
} else {
// Manhattan Distance in steps
return abs((x + y)/4);
}
}

int knightAstar(int row, int column, int rowGoal, int
    columnGoal){
initialize();
Heap h = newEmptyHeap();
Entry e = newEntry(heur(column, row, columnGoal, rowGoal),
    column, row);
enqueue(&h, e);
while(!isEmpty(&h)){
e = removeMin(&h);
++statesVisited;
if(e.x == columnGoal && e.y == rowGoal) {
freeHeap(h);
return costShortestPath[e.y][e.x]+1;
}
for(int i = 0; i < 8; ++i){
int newX = e.x + actions[i][0];
int newY = e.y + actions[i][1];
if(isInBounds(newX, newY) && costShortestPath[newY][newX] >
    costShortestPath[e.y][e.x]+1){
costShortestPath[newY][newX] = costShortestPath[e.y][e.x]+1;
enqueue(&h, newEntry(costShortestPath[newY][newX]+heur(newX,
    newY, column, row), newX, newY));
}
}
costShortestPath[e.y][e.x] = -1;
}
return -1;
}
```

```

int knightIDS(int row, int column, int rowGoal, int columnGoal) {
int limit = 0;
printf ("limit=0"); fflush(stdout);
initialize();
while (knightDLS(0, limit, row, column, rowGoal, columnGoal) ==
    0) {
initialize();
limit++;
printf(",%d", limit); fflush(stdout);
}
printf("\n");
return limit;
}

int main(int argc, char *argv[]) {
int x0,y0, x1,y1;
do {
printf("Start 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", &x1, &y1);
} while (!isValidLocation(x1,y1));

printf("Length shortest path: %d\n", knightAstar(x0,y0, x1,y1));
printf("#visited states: %lu\n", statesVisited);
return 0;
}
```

heap.c

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

#include "heap.h"

/* Code for a min-priority heap structure */

void swap(Entry* a, Entry* b){
Entry t = *a;
*a = *b;
*b = t;
}

Entry newEntry(int priority, int x, int y){
Entry e;
e.priority = priority;
```

```c
e.x = x;
e.y = y;
return e;
}

Heap newEmptyHeap(){
Heap h;
// indices start at 1, first array element is left undefined
h.array = malloc(2 * sizeof(Entry));
h.size = 2;
h.front = 1;
return h;
}

void freeHeap(Heap h){
free(h.array);
}

int isEmpty(Heap* h){
return (h->front == 0);
}

int doubleHeapSize(Heap* h){
int newSize = 2 * h->size;
h->array = realloc(h->array, newSize * sizeof(Entry));
if(h->array == NULL) fprintf(stderr, "Fatal error: realloc()
    failed\n");
return newSize;
}

void upheap(Heap* h, int childIdx){
if(childIdx == 1) return;
int parentIdx = childIdx/2;
if(h->array[childIdx].priority < h->array[parentIdx].priority){
swap(&(h->array[childIdx]), &(h->array[parentIdx]));
upheap(h, parentIdx);
}
}

void downheap(Heap *h, int idx){
if(2*idx < h->front){
Entry* lc = h->array + 2*idx;
Entry* rc = (2*idx+1 < h->front ? h->array + 2*idx+1 : lc);
if(h->array[idx].priority > lc->priority && lc->priority <=
    rc->priority){
swap(lc, &(h->array[idx]));
downheap(h, 2*idx);
} else if (h->array[idx].priority > rc->priority){
swap(rc, &(h->array[idx]));
downheap(h, 2*idx+1);
}
}
}

void enqueue(Heap* h, Entry n){
if(h->size == h->front) h->size = doubleHeapSize(h);
h->array[h->front] = n;
upheap(h, (h->front)++);
}

Entry removeMin(Heap *h){
Entry r = h->array[1];
h->array[1] = h->array[--(h->front)];
downheap(h, 1);
return r;
}
```

heap.h

```
#ifndef HEAP_H
#define HEAP_H

typedef struct Entry {
int priority;
int x;
int y;
} Entry;

typedef struct Heap {
Entry* array;
int size;
int front;
} Heap;

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

/* Code for a min-priority heap structure */

void swap(Entry* a, Entry* b);
Entry newEntry(int priority, int x, int y);
Heap newEmptyHeap();
void freeHeap(Heap h);
int isEmpty(Heap* h);
int doubleHeapSize(Heap* h);
void upheap(Heap* h, int childIdx);
void downheap(Heap *h, int idx);
void enqueue(Heap* h, Entry n);
Entry removeMin(Heap *h);
```

```

        #endif
```