Artificial Intelligence 1
Lab 1

Name1 (student number 1) & Name2 (student number 2)
Group name

day-month-year

Theory

Exercise 1

Exercise 2

Programming

Program description

The program is used to calculate the shortest path on a 500*500 chess table from a field to another field, both asked in the input. The path is calculated on the way the knight is moving. The program can use IDS and A* search methods, with 2 heuristics for the latter: the straight line distance from goal, and the minimal number of steps that is needed to reach the goal.

Problem analysis

The task was to implement an A* search with two different heuristics, since the IDS was already implemented. We also programmed an easy way to compare solution methods with being able to chose running both modes and both heuristics after each other.

Program design

The implementation of A* can be found in the knightAStar function. It uses the heap implementation of a priority queue based BFS, but the priority is calculated on the heuristic method, taken as a user imput, and calculated in the heuristicFunction function. The matrix costShortestPath is used to store the lowest estimated total cost (path + heuristic) for all states, and we only enque the states, that give lower estimated total cost for the fields than the one already stored in the matrix (set to infinity at start). The program checks for solution when dequeueing from the heap.

We implemented two heuristic functions for the A*: The first one calculates the straight line distance based on the pythagorean theorem, and divides it by the square root of 5 to keep the admissibility, since one steps in the solution contributes to $root(5)$ in distance.
The other adds takes the absolute value of the difference of the row and column value from the goal row and column, and divides it with three, calculating the minimum steps that is needed to find the goal, since a horse takes three steps in one turn (two horizontally, one vertically, or the other way around). This fulfills the heuristic criteria.
Both heuristic are rounded to integers for simplicity, since it does not break the admissibility. As expected, in general the first one gives faster solution, since it also helps finding a direction, whereas the other one might go in wrong ways, just because the added absolute distance of rows and columns is decreasing. The branching factor

Program evaluation

The program is running with no memory leaks. The complexity depends on the mode: for the IDS method the time complexity is $O(4^d)$, where d is the solution depth and the memory complexity is O(4d) (same as O(d)). For the A*, the complexity is

Program output

The program outputs the shortest path with a knight between two fields on the chess table, and the method that was used. It also outputs the visited nodes in IDS and the enqueued and dequeued nodes in A*, to see how optimal the algorithm is.

Program files

idknight.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "heap.h"
#include <string.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}
};
int costShortestPath[N][N];
```

```
unsigned long statesVisited = 0;
unsigned long enqueued = 0;
unsigned long dequeued = 0;

int distanceFromGoal(int row, int col, int rowGoal, int colGoal){
        return (sqrt(pow((row-rowGoal), 2) + pow((col - colGoal),
            2)))/sqrt(5);
}

int minimumSteps(int row, int col, int rowGoal, int colGoal){
        return (abs(rowGoal-row) + abs(colGoal-col))/3;
}

int heuristicFunction(int row, int col, int rowGoal, int colGoal, int
    heuristic){ //both functions return an integer value for simplicity
    reasons, since the decimal part of the heuristic value would not
    make a significant difference
        switch(heuristic){
                case 1:
                        return distanceFromGoal(row, col, rowGoal,
                            colGoal);
                case 2:
                        return minimumSteps(row, col, rowGoal, colGoal);
        }
}

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 heurEval(char *heur){
        if(!strcmp(heur, "Distance")){return 1;}
        if(!strcmp(heur, "MinSteps")){return 2;}
        if(!strcmp(heur, "Both")){return 3;}
        return 0;
}

int isGoal(int row, int column, int rowGoal, int columnGoal){
        return rowGoal==row && columnGoal == column;
}
```

```
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 knightIDS(int row, int column, int rowGoal, int columnGoal) {
	printf("---------starting iterative deepening search---------\n");
  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 knightAStar(int row, int column, int rowGoal, int columnGoal, int heuristic){
	printf("---------starting A* search---------\n---------heuristic is %d---------\n", heuristic);
	initialize();
	costShortestPath[row][column] = 0;
	Heap q = makeHeap();
	enqueue(newState(row, column, 0, heuristicFunction(row, column, rowGoal, columnGoal, heuristic)), &q);
	enqueued++;
	while (q.array!=NULL){
		State position = dequeue(&q);
```

```
			dequeued++;
			row = position.row;
			column = position.column;
			if (isGoal(row, column, rowGoal, columnGoal)){
				return position.pathlen;
			}
			//printf("position is %d %d goal is %d %d\n", row, column, rowGoal, columnGoal);
			for (int act=0; act < 8; act++) {
				int r = row + actions[act][0];
				int c = column + actions[act][1];
				if (isValidLocation(r, c)){
					//if (isGoal(r, c, rowGoal, columnGoal)){
						//return position.pathlen + 1;
					//}
					int estimatedPath = position.pathlen + 1 + heuristicFunction(r, c, rowGoal, columnGoal, heuristic);
					if(estimatedPath < costShortestPath[r][c]){
						enqueue(newState(r, c, position.pathlen + 1, estimatedPath), &q);
						costShortestPath[r][c] = estimatedPath;
						enqueued++;
					}

				}
			}
		}
		return 0;
}
void IDS(int x0, int y0, int x1, int y1){
		printf("Length shortest path: %d\n", knightIDS(x0,y0, x1,y1));
		printf("#visited states: %lu\n", statesVisited);
}
void AStar(int x0, int y0, int x1, int y1, int heuristic){
		if (heuristic < 3){
			printf("Length shortest path: %d\n", knightAStar(x0,y0, x1,y1, heuristic));
			printf("#enqued states: %lu\n", enqueued);
			printf("#dequed states: %lu\n", dequeued);
		}else{
			for (int h=1; h<3; h++){
				enqueued=0;
				dequeued=0;
				printf("Length shortest path: %d\n", knightAStar(x0,y0, x1,y1, h));
				printf("#enqued states: %lu\n", enqueued);
				printf("#dequed states: %lu\n", dequeued);
```

```
                }
        }
}
int main(int argc, char *argv[]) {
  int x0,y0, x1,y1;
  char method[5];
  char heurS[10];
  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));
        do {
        printf("Give a valid method (IDS|AStar|Both)\n");
        scanf("%s", method);
        }while(strcmp(method, "IDS")&&strcmp(method,
            "AStar")&&strcmp(method, "Both"));

        int heuristic;
        do {
        printf("Give a valid heuristic method
            (Distance|MinSteps|Both)\nDistance calculates the straight
            line distance to the goal, MinSteps calculates the minimum
            number of steps to the goal\n");
        scanf("%s", heurS);
        heuristic = heurEval(heurS);
        }while(heuristic == 0);

        if (!strcmp(method, "IDS")){
                IDS(x0,y0,x1,y1);
        }else if(!strcmp(method, "AStar")){
                AStar(x0, y0, x1, y1, heuristic);
        }else{
                IDS(x0,y0,x1,y1);
                printf("\n");
                AStar(x0, y0, x1, y1, heuristic);
        }
  return 0;
}
```

heap.c

```
// heap.c is based on the version we used in course Algorithms and Data
    Structures

#include <stdio.h>
```

```c
#include <math.h>
#include <stdlib.h>
#include <assert.h>
#include "heap.h"
#include <string.h>

State newState(int row, int column, int pathlen, int total){
	State s;
	s.row = row;
	s.column = column;
	s.pathlen = pathlen;
	s.total = total;
	return s;
}

Heap makeHeap () {
	Heap h;
	h.array = malloc(1*sizeof(State));
	assert(h.array != NULL);
	h.front = 1;
	h.size = 1;
	return h;
}

int isEmptyHeap (Heap h) {
	return (h.front == 1);
}

void heapEmptyError() {
	printf("heap empty\n");
	abort();
}

void doubleHeapSize (Heap *hp) {
	//printf("doubleing heap size\n");
	int newSize = 2 * hp->size;
	hp->array = realloc(hp->array, newSize * sizeof(State));
	assert(hp->array != NULL);
	hp->size = newSize;
}

void swap(State *pa, State *pb) {
	State h = *pa;
	*pa = *pb;
	*pb = h;
}

void enqueue (State n, Heap *hp) {
	//printf ("enqueueing position with %c\n", n.type);
	int fr = hp->front;
```

```
        if ( fr == hp->size ) {
                doubleHeapSize(hp);
        }
        hp->array[fr] = n;
        upheap(hp,fr);
        //printf("the first position is %d %d\n", hp->array[1].col,
            hp->array[1].row);
        hp->front++;
}

void upheap(Heap *hp, int n){
        //printf("upheap started\n");
        if (n<=1){return;}
        if ( hp->array[n/2].total>hp->array[n].total ) {
                swap(&(hp->array[n]),&(hp->array[n/2]));
                upheap(hp,n/2);
        }
}

void downheap (Heap *hp, int n) {
        //printf("downheap started\n");
        int fr = hp->front;
        int indexMax = n;
        if ( fr < 2*n+1 ) { /* node n is a leaf, so nothing to do */
                return;
        }
        if ( hp->array[n].total > hp->array[2*n].total ) {
                indexMax = 2*n;
        }
        if ( fr > 2*n+1 && hp->array[indexMax].total >
            hp->array[2*n+1].total ) {
                indexMax = 2*n+1;
        }
        if ( indexMax != n ) {
                swap(&(hp->array[n]),&(hp->array[indexMax]));
                downheap(hp,indexMax);
        }
}

State dequeue(Heap *hp) {
        State n;
        if ( isEmptyHeap(*hp) ) {
                heapEmptyError();
        }
        n = hp->array[1];
        hp->front--;
        hp->array[1] = hp->array[hp->front];
        downheap(hp,1);
return n;
}
```

heap.

```c
// heap.h is based on the version we used in course Algorithms and Data Structures


typedef struct State {
	int row;
	int column;
	int pathlen;
	int total;
} State;

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


Heap makeHeap () ;
State newState(int row, int column, int pathlen, int total);
int isEmptyHeap (Heap h) ;
void heapEmptyError() ;
void doubleHeapSize (Heap *hp) ;
void swap(State *pa, State *pb) ;
void enqueue (State n, Heap *hp) ;
void upheap(Heap *hp, int n);
void downheap (Heap *hp, int n) ;
State dequeue(Heap *hp) ;
```