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`	`Length shortest path: 334`
`#visited states: 250000`	`#visited states: 250000`
`------`	`------`
`Length shortest path: 166`	`Length shortest path: 166`
`#visited states: 198267`	`#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 colun
  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' patl
  }
  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, colun
      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] > costShortest
        costShortestPath[newY][newX] = costShortestPath[e.y][e.x]+1;
        enqueue(&h, newEntry(costShortestPath[newY][newX]+heur(newX, newY, col
      }
    }
    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;
        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
                        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
```