Corné Thoes
Team Two
student nr. 500697481

Knights tour assesment report

Program structure

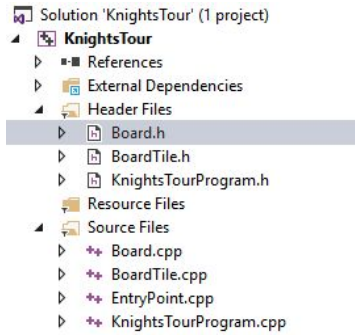
The program starts in the entry point, where it creates a knightstourprogram. The knightstourprogram has a board class which consists of boardTiles. These boardTiles have position, visited, tileValue and neighbour values. The neighbour values are used to check the amount of moves a tile can make.

The board class takes care of drawing the board and calculating the neighbours for the tiles. The knightsTourProgram class handles user input and the actual warnsdorf algorithm.

Requirements

Programming language has to be c++

The code is written in visual studio using an empty c++ project.
You can see that c++ is used because of the use of .cpp and .h files.



The program makes use of pointers and references

I use a couple of pointers and references throughout the project.
The knightstour program has a board class . This board class has a pointer to the currentTile

```
BoardTile *CurrentTile = &boardTiles[0][0];
```

Furthermore, the board class also contains a function that returns a vector with pointers to BoardTiles

```
std::vector<BoardTile*> GetNeighbours(BoardTile tile);
```

The knights tour program also
has its own function that returns a pointer to a BoardTile, namely :

```
BoardTile* GetNextStep();
```

References are used in both DrawValues and GetNeighbours. We do not need a pass-by-value.

```
void DrawValues(int& yPos);
std::vector<BoardTile*> GetNeighbours(BoardTile& tile);
```

There is a visual representation

the board is visualed using the console. In the console a board is being drawn with the steps being shown as numbers. The the number equals the amount of steps it has taken to get there, minus one for the start position.

```
Tour finished with tour:
----------------
|01|22|17|12|03|
----------------
|16|11|02|25|18|
----------------
|21|08|23|04|13|
----------------
|10|15|06|19|24|
----------------
|07|20|09|14|05|
----------------
```

The board is at least 5x5 in size

as you can see in the image, the board is 5x5. Which makes this requirement complete. You can however adjust the size by adjusting the board initializer list.

```
Board::Board()
    :BoardSize(10), boardTiles(BoardSize,std::vector<BoardTile>(BoardSize))
{
```

By increasing it to 10 we get this board

The knight has to move according to its movement rules
I could have written this in a nicer way, but working on the prototype was way more fun. The movement comes down to four directions. North/east/south.west
North has Y-2 and X -1 or X+1
East has X +2 and Y -1 or Y +1
South has Y+2 and X -1 or X+1
West has X -2 and Y -1 or Y +1

By checking if these movement patterns fit in the board reletive to the knight his current position, we can decide if something is a valid more or not.

```
//Code spagethi should clean this up, but rather work on prototype
if (tile.yPos - 2 >= 0) // north
{
    if (tile.xPos - 1 >= 0)
    {
        if (boardTiles[tile.xPos - 1][tile.yPos - 2].visited == false)
        {
            tile.neighbours.resize(tile.neighbours.size() + 1);
            tile.neighbours[tile.neighbours.size() - 1] = &boardTiles[tile.xPos - 1][tile.yPos - 2];
        }
    }
    if (tile.xPos + 1 < BoardSize)
    {
        if (boardTiles[tile.xPos + 1][tile.yPos - 2].visited == false)
        {
            tile.neighbours.resize(tile.neighbours.size() + 1);
            tile.neighbours[tile.neighbours.size() - 1] = &boardTiles[tile.xPos + 1][tile.yPos - 2];
        }
    }
}
```

This is copy/pasted for the other 3 directions with the appropriate X and Y values
The neighbours are then added to the tile, so the tile knows what the valid movements are.

The knight can only visit each square once
Because of the fact that i check if each movement is valid in the above image, this was also the obvious spot for checking if the tile had already been visited, and it is. Disallow it as a movement option. By recalculating all neighbours after every move, we maintain integrety.

Exemplars

Board size can be set by the user(NxN size)

depending on the knowledge of the user, this can be done by editing the initializer list in the board constructor.

The user decides where the knight starts

I did not implement this feature because there are alot of things you need to take into consideration. For example, if the board is uneven(5x5,7x6, etc) we can only use half of the starting points. The algorithm i used also works less good(warnsdorff's) with a random starting position. I would have needed to implement movement orders, and maybe retry a couple of times before I would get the correct order.

Uses a more efficient solution than brute force

The algorithm I use to calculate the tour is Warndorrfs. This algorithm checks each movement option and takes the movement option with the least ammount of following options. If there is a tie, a random one is taken. If warndorffs choses the random tie's wrong a couple of times in a row a solution may not be found. This is more efficient than brute-force because it does not have to calculate and backtrack many different routes.