Assignment 4g

[Niels Dreesen, Mustafa Hekmat Al Abdelamir, Jakup Højgaard Lützen]

October 8, 2022

1 Report for assignment 4G

This assignment involves recursive functions to find the shortest path between two coordinate points.

(a)

In this section we write a function that calculates the distance between two points. So it takes two parameters for pos (int*int) and returns an int.

$$dist : p1 : pos- > p2 : pos- > int$$

This function splits the p1 and p2 parameters into their x and y components and calculates the x and y difference and then plugs it into a squared distance formula.

```
let dist (p1:pos) (p2:pos) : int =
    let xDelta = (fst p2) - (fst p1)
    let yDelta = (snd p2) - (snd p1)
    xDelta * xDelta +  yDelta * yDelta
```

(b)

This section involves a function which takes in a source coordinate and a target coordinate as parameters and calculates, which adjacent coordinates are closer to the target than the source.

$$candidates : src : pos- > tg : pos- > pos\ list$$

We approach this problem by making a list of all the adjacent coordinates, and then we loop through the list with a List.filter function, which throws out the coordinates which are further away from the target coordinate than the source. Then we return the filtered list.

```
let candidates (src:pos) (tg:pos) : (pos list) =
    let x, y = src
    let adjacent =[(x,y-1); (x+1,y); (x, y+1); (x-1,y)]
    List.filter (fun i -> (dist i tg) < (dist src tg)) adjacent
```

(c)

In this section we develop a function, which takes a source and a target coordinate and is supposed to outputs a different list of coordinates which represent the shortest path from source to target.

$$routes : src : pos- > tg : pos- > pos\ list\ list$$

To do this we calculate the candidates for each block on the path, and calculate the candidates for these and so on. When the function reaches the last step (target coordinate, which is reaches once for every distinct path) it gets returned and stored in a list with all the other function calls in the last step (which are all returning the target coordinate), and passes it into a List.concat function to make it of appropriate form (pos list list). We add the source coordinate (from which we got the target as a candidate) to each list and continue until every candidate coordinate is recursively concatenated with the block from which it was calculated. It ends when the original source value is reached, and the function outputs the desired result:

```
let rec routes (src:pos) (tg:pos) : pos list list =
    match src with
        a when a = tg -> [[src]]
        | _ ->
            candidates src tg
            |> List.map (fun p -> routes p tg)
            |> List.concat
            |> List.map (fun instr -> src::instr)
```

(d)

In this section we make it possible to travel diagonally and calculate the shortest path accordingly.
We firstly tried simply adding diagonal blocks to the adjacent block list and running the same algorithm as before, finding all candidates that bring the robot closer to the target, but this turned out to be a naive approach as showed in figure 1.
The problem arises when evaluating neighbouring candidates, it is possible, in some configurations, to find a point which satisfies the condition of being closer to target, keeping the list length at a minimum, without being the geometrically shortest route. This is especially problematic when it takes a position outside the normal grid, such as in figure 1, where the shortest route between (0,0) and (1,3) would return following list l, which goes into negative x-values:

```
[[(0, 0); (1, 1); (1, 2); (1,3)];
 [(0, 0); (0, 1); (1, 2); (1,3)]
 [(0, 0); (0, 1); (0, 2); (1,3)]
 [(0, 0); (-1, 1); (0, 2); (1, 3)]]
```

When looking at the path lengths, it is clear that the last path $(l\,[3])$ is longer than the others and thus should not be included:

$$|(l\,[0])| = |(l\,[1])| = |(l\,[2])|\sqrt{2}+2 \approx \underline{\underline{3.41}}$$
$$|(l\,[3])| = 3\sqrt{2} \approx \underline{\underline{4.24}}$$

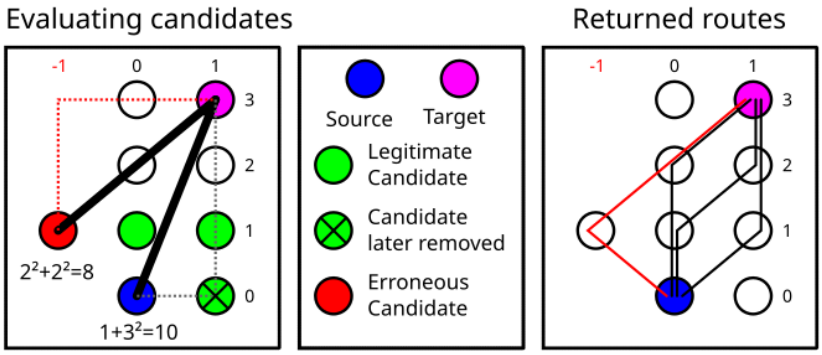

Figure 1: Sidestep-problem when adding diagonal candidates

We settled on a better method of adding diagonal candidates, by realizing, that diagonal candidates should only be proposed when there exist horizontal and vertical candidates. Furthermore, being that there can only exist one horizontal and one vertical candidate at a time, a solid method of generating the diagonal, is to add the relative movement of the two adjacent candidates together, and then prepend the diagonal. This completely avoids the problem illustrated in figure 1 as such:

```
    if fst candidateSum <> 0 && snd candidateSum <> 0 then
        let diagonalPosition =
            (x + fst candidateSum, y + snd candidateSum) // add src to candidateSum
        diagonalPosition::adjacentCandidates
    else
        adjacentCandidates
```

The second change is an extra check on our final result to remove any paths that are longer than the shortest path. We do this because our candidates function outputs candidates based on a calculation from each distinct block on the way. This means the shortest route from one block to the target could be longer than the shortest route from the source to the block to the target and could be mistakenly added. We chose this approach because the recursive routes function has no easy way of comparing the different routes before it has actually calculated every route.

To implement this we create a list of integers which represent the length of each distinct path found. We do this with List.map which maps every element to its length. We use a List.min function to find the smallest path. Then we use a List.filter function to filter out the elements which are longer than the minimum. This is how the path containing point (1,0) is removed in figure 1
This - as previously mentioned, does not however filter out the "sidestep problem" in figure 1, which is why we designed the new way of generating diagonals:

(e)

For this section we draw our result to a canvas.

The first step, is to create a pos list list. This can be understood as a two dimensional area with coordinates. In other words, a 2d plane where each point in the plane is a coordinate pair x,y. The coordinates represent pixels on screen. This means we can draw circles at each point to make a grid on screen, and we can access each circle position by using the index of the pos list list. In other words we have a itenary, where we look up a coordinate, by its column and row numbers.

We wanted the program to be flexible, allowing for different sized grids and canvases (as seen in figure 2), so we simply defined how large portion of canvas is to be used for the grid.
We then find the larges x- and y-values in the source and target input, and draw a grid sized (max x_1 x_2 + 1, max y_1 y_2 + 1), just to give some space at the top right corner.
We also wanted the program to be easily understood, so it was necessary to add grid markers. For this, we had the prefect function, the recursive circle generator function from week-1.

(a) drawRoutes (600, 400) (0,0) (2,3)

(b) drawRoutes (600, 400) (2,1) (5,7)

Figure 2: Demonstration of different sized grids, when routes span different areas