KF04 project report

Group 12:
Jakob Melnyk
Niklas Hansen
Emil Juul Jacobsen
Filip Hjermind Jensen
Jens Dahl Møllerhøj

May 3, 2011

Indhold

1 Features

This chapter describes the features and functions, we have included in our Map of Denmark project, part 2, Spring 2011.

- Find shortest route
- Display path using blue highlighting
- Multiple pins
- Total distance
- Estimated travel time
- Bike or car route
- Remove a single pin by clicking
- Remove all pins by using GUI button

1.1 Find shortest route

Our map is able to calculate the shortest route between two points. This is done by clicking the map. Clicking the map places a pin at the clicked location. If the user clicks on another part of the map, our map will show the shortest route between those two locations, if possible.

1.2 Display path using blue highlighting

When the user has selected two or more locations, the map will show the shortest route highlighted in blue. The algorithm we use for finding this route is pretty fast, so a user will not feel annoyed by how long it takes.

1.3 Multiple pins

It is possible for a user to place multiple pins on the map. This is done in similarly to finding the shortest route between two points. When any number of pins are placed, the user can add additional ones. This will calculate the route between the latest two pins placed. The route shown will be from pin 1-¿2-¿3-¿4..

1.4 Total distance

Our Map of Denmark program calculates the total distance of the route found, a functionality that assists the end-user in planning their trip. This was relatively simple to implement, but adds a big functionality.

1.5 Estimated travel time

As with the total travel distance, our Map of Denmark calculates the estimated time it will take to travel the route found by our algorithm. This was relatively easy to calculate and is a huge benefit for the end user.

1.6 Bike or car route

It is possible for the user to choose between a car route and a bike route. The different options will make our algorithm consider what is possible to traverse for both cars and bikes. It will make sure not to show small paths for cars and not show highways for bikes.

1.7 Remove a single pin

To make it easier to use our Map of Denmark, we have it possible to remove a pin by clicking on the location it was placed. This makes it possible for the end user to clear up a tiny missclick without having to shut down the entire Map or clearing all the pins the user has placed.

1.8 Remove all pins

In addition to enabling the user to remove a single pin by clicking on it, we have also made it possible to remove all the pins currently placed. This can be done by either pressing the "c" key on the keyboard or by clicking the button on the graphical interface. This avoids the chore of clicking every single point in order to start a new route planning.

2 Implementation

This chapter describes our implementation of interesting methods and functions in our Map of Denmark project, part 2, Spring 2011.

2.1 Model

In order to implement the mentioned pathfinding the Model has been extended to include a list of Edges. These correspond to all the roads that are in the current route. Whenever it recieves a new request to find a path from one Node to another it uses the static method in the Dijkstra class and adds the result to the list. When Control needs to draw the route it uses a method that returns the entire route as Line objects relative to the view. This approach is identical to what we use when the Control requests all roads that should be drawn as the map for a given view.

We have also included some statistics of the current route. These are calculated by the Model since it's the object that primarliy handles the data. This information can be calculated in linear time relative to the length of the current route since the route is saved in memory when first found.

2.2 Dijkstra

We use Dijkstras algorithm to find the shortest route from one node to another. This algorithm builds a Shortest Path Tree by always looks at the node nearest the source node. This means that whenever the destination node is reached, the shortest route to that node is found. When that happens, we stop the algorithm because we have found what we are looking for. In order to always choose the node closest to the source, our implementation use a priority queue, where its distance to the source indexes each node. Sometimes, we want to decrease the index of a node, because we have found a route that is shorter than the one previously indexed. Because the priority queue in javaÕs library is slow when decreasing keys, we use an implementation from the book Algorithms Fourth Editon. We have written the code for the dijkstra algoritm using our knowledge of how it works. We have been inspired by the implementation used in Algorithms Fourth Editon. Additionally we are using the IndexMinPQ directly as it is from the same book.

We have made some rather big changes to the Dijkstra algorithm to make it fit our requirements. First of all we have replaced the arrays that holds edgeTo and distTo with Maps so we can use our objects instead of integers. We have however not changed anything in the IndexMinPQ implementation

that takes integers as values, this is not a problem though, since our nodes have a unique integer index. Another major change in the Dijkstra code is that we have modified it to be a static method instead of a seperate object. This way the classes using it will not have to instantiate it before use. To be able to evaluate the edges differently depending on the type of transportation that the user has selected, we have created the Evaluator class. This must be passed to the pathfinding and is responsible for transforming the KrakEdge to a value that can be inserted into the priority queue. An important part of the pathfinding is the exclusion of the roads that are unidirectional in the opposite direction. This check is performed every time a road is going to be added to the priority queue, if the road can't be used it will skip to the next. It is important to note here that we have found an error in the data from krak. The problem is that some of the roads at the highways are oriented in the wrong direction, this implies that the pathfinding will avoid the highway because it will only be able to use short stretches of it. In order to be able to use the highway we have inserted a little snip of code that allows traveling on the highway even though it is facing the wrong way. This cheat however, gives accurate routes since the access and exit roads are without error and can only be entered from the right direction.

2.2.1 The A-star technique

In order to make our path finding algorithm even faster, we used a variation called A-star. This implementation benefits from the fact that the roads, in contrast to a theoretical graph, are part of a geometric system. This means that we can calculate the fastest possible route from one point to another. This helps us determine which routes to look at first. The fastest possible route is always a straight line from the source point to the end point. The smaller the difference from the straight line is to the fastest possible route, the more we will benefit from the A-star technique. This difference will often be bigger for the routes calculated for the car. This is because the fastest route would be a straight highway, which often is far from possible. We can therefore conclude that the A-star technique typically is more of a benefit when calculating routes for bicycling because they rely on the distance. In order to make the code work with our dynamic evaluations for the Edges we have created another method in the Evaluator that calculates the heuristic for at given Node relative to the target Node.

2.3 Control

A lot has changed in Control since our last version. We have changed a lot of methods to work as static methods in a seperate Tools class, so Control

will feel less cluttered in this version. New logic has taken the place of the old logic however. Route calculation, pin placement, pin removal and route display all goes through Control.

When a user clicks on the map, Control uses our Tools class to convert the screen coordinates to UTM coordinates. It then stores these in a List. If the list holds two or more elements, it asks the Model to calculate the shortest path between the first and second element, the second and third element and so on, if any additional elements exist beyond the first two. Every time Control calls it's internal repaint method, it clears all the pins currently in the View. It then recalculates the pin position(s) according to the new size and postion of the View.

When a user takes advantage of one of the options to clear all the pins currently on the Map of Denmark, Control empties the collection holding all the pin points, asks Model to clear the route and asks View to repaint.

If a user removes a single pin by clicking on the location of the pin, we run through our collection and remove one of the pins within a specific range of pixels. It then makes Model recalculate the route for all the pins. This is not an effective implementation - if there are a lot of pins, then removing one might make the recalculating take a while. We decided to use this implementation, for now at least, because we wanted a working functionality that we could later improve upon if nessecary.

2.4 Car/Bike routes

Another useful feature we have included is the ability to switch between multiple modes of pathfinding. We have named these two modes Car and Bike. In the Car mode the pathfinding will priorize lower travel times, since we believe this to be most important when driving. Additionally the Car mode will exclude small paths and pedestrian zones. The Bike mode prioritizes lower distances highest since intersections and speed-limits are of relatively low importance here. When using Bike mode the route will not include highways.

As described earlier these different ways of prioritizing the roads are achieved by using the Evalutator that is provided when asking the pathfinding method for a best path. We have written these two evaluators as static objects on the Evaluator class for easy use in the code, but the use of a seperate object for the evaluations open up for later variation with little impact on the original code.

2.5 View

When looking at the changes in View relatively little has happened since most of the new code is located "behind the curtain". The Canvas has been extended to allow adding and drawing of the route. This is however, as mentioned earlier, done in exactly the same way as with the roads in the map itself. The Canvas also includes some points that is positions where the pins are placed. These are drawn on top of the map and (like the map) needs provided again if the map moves.

When looking at the menu to the left there are a few more options to the user. There is a new button to remove all pins from the map and two RadioButtons to toggle between Car mode and Bike mode in the pathfinding.

Beneath the tools for pathfinding there are displayed two pieces of information (and spare space for more). This information is changed by a simple public method in the View and the units will adjust will be adjusted accordingly.

3 Other?

This is where we bla bla.

3.1 Serialization

Everytime we start the program, we loop through the entire dataset given to us by Krak. These data are huge, and therefore it takes quite a lot of time to start the program. Because we need to load all these data, the user is presented with a blank screen for a long time, before all these data are loaded and the program starts.

We started looking for a way to speed up the loading process, so the user has a map in front of him or her quickly, when the user starts the program. What takes the most time, is looping through the data and creating the needed datastructures (quadtrees and such), so if we could skip these steps, we could save a lot of time.

This is where serialization comes into play. By serializing an object, you transform your object into something that can be passed around, through streams and such. So by serializing objects, you can save them to files. If the object to be serialized contains references to other objects, these will also be serialized (if they are Serializable / implements java.io.Serializable). By doing this, we only need to build our datastructures the first time you start the program. After the objects have been created, they are been serialized and saved to files. The next time the user starts the program, we check whether the data has been changed. If it hasn't been changed, we load the objects that we saved, instead of making them all over.

3.1.1 Threading

By serializing our main objects, we cut several seconds of our load times. But we can do it even faster. We are using several objects, but only few of them are needed right from the start. So what we can do to speed it up even more, is loading the few neccesary objects, and then load the rest in the background. We do this with threads. We load the few objects we need from the start, then create a new thread to load the rest of the objects, and in the mean time, we create the window and draw the map.

The same goes for the first run. The user doesn't need to wait for the program to finish serializing and saving to files. By using threads, we can create the datastructures, and then immediately show the window to the user, while saving the objects to files in the background.