Computing a short route for a mobile guard on an arrangement of horizontal and vertical segments.

AMS 545 COMPUTATIONAL GEOMETRY. SPRING 2015

Lecturer: Professor Joseph S. B. Mitchell

Stony Brook University

Student: Caleb Andrade ID 110071013

May 7th 2015

1 Proposed Algorithm

1.1 Motivation

Given that the goal is to guard a map $\{H, V\}$ with the shortest route, a first attempt would be to use the smallest edges in the arrangement, in the spirit of Kruskal's MST algorithm, but the lenght of the edge might not be the only criteria to construct our solution. Other heuristic criteria would be to select edges in *well connected zones*, that is, on segments that intersect the most with others, so that marching through the edges of such segment would guard a good amount of the map. With this two ideas in mind the heuristic was developed. So we define a weight function

$$w(s) = \frac{l(s)}{|[s]|}$$

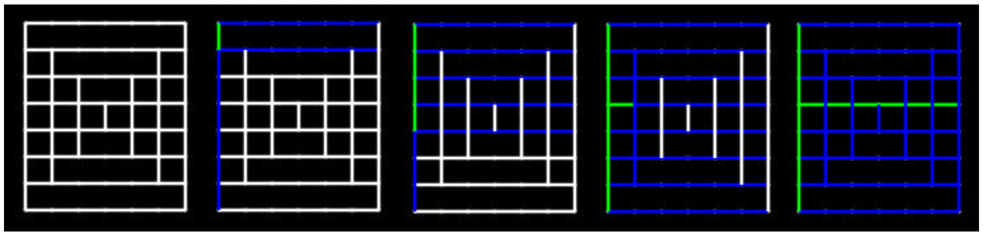
where $l(s)$ denotes the euclidean lenght of the edge, and $|[s]|$ denotes the cardinality of its class of equivalence, i.e., the number of collinear edges that come from the same line segment. Once this function is defined we can sort the edges of the arrangement and the best candidates would be in principle those whose weight is minimal, either because they are very small or because their class of equivalence has many elements or both. Now we can define a very straightforward incremental heuristic to build a solution to our problem

1.2 Algorithm

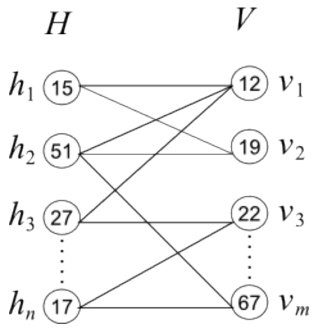
We describe the algorithm with the following pseudocode

```
Input : Map
Output: connected subset of the edges of the map that guards it
            Given a Map, pre-process it:
            Build the line arrangement (merge and split)
            Rearrange the edges in the lists lexicographically
            Sort edges according to weights, select $s_{min}$
            while map is ungarded do:
                Neighbors = Neighbors + $N(s_{min})$
                $s_{min} \leftarrow$ smallest feasible neigbhor in Neighbors
                paint $s_{min}$ and update guarded segments
                Update Neighbors, pop out unfeasible Neighbors
                Update guarded map
```

This heuristic seems to perform well when there are not many ties in weight, so the choices are somewhat *obvious*, for example



At this point there is not yet a theoretical framework that would support that this algorithm could be an approximation algorithm. However, K. Clarkson proposed a $2-approximation$ algorithm for the vertex cover problem using basically the same weight function; it happens that this problem can be stated similarly to a vertex cover problem in the connectivity graph



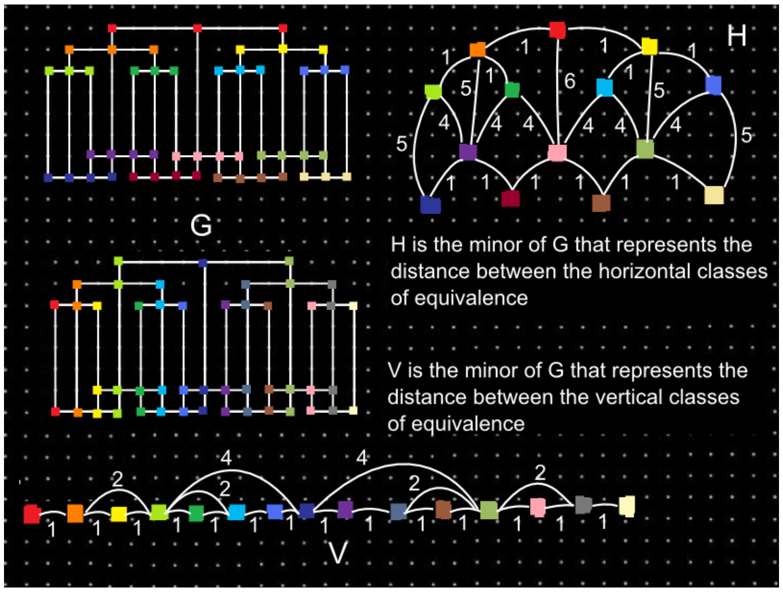
where the graph is bipartite, connecting horizontal classes of equivalence to those vertical intersected classes of equivalence. The weights in this case are the lenght of the classes of equivalence. So the approach taken in the proposed heuristic might have some chance to yield some interesting result. Theoretical work is yet to be done in this regard, as well as some refinements in the local search technique so that the algorithm always yields a connected solution (there are some examples where the current algorithm gets stuck at some point and disconnects the solution).

1.3 Algorithm analysis

The line arrangement can be constructed in $O\left(n^2\right)$ running time. The storage is basically a list $O\left(n^2\right)$ (note that each edge stores a list with its neigbhors, but they are at most 6, i.e. $O(1)$, so we are still in the same storage space). The sorting is the most expensive stage with $O\left(n^2 \log n^2\right)$ to select the smallest weights. Now, because each segment is added only once to the solution, or not added at all, and because edges that enter the neighbor bag, once they are popped will not return again, the overall running time of the algorithm is $O\left(n^2 \log n^2\right)$.

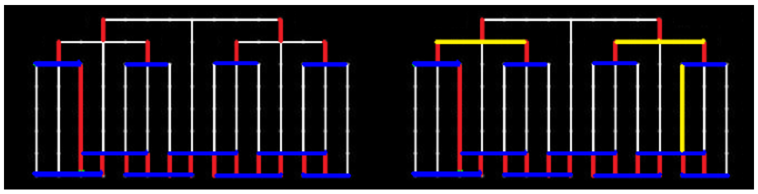
1.4 Lower bound for OPT

An important observation is that given a map it can be computed a lower bound for OPT (without knowing how loose this bound could be, but there are many examples where the bound is tight). The bound consists basically in the lenght of the MST in the distances between classes of equivalence of the same kind. The following diagram explains this idea.



H
G
H is the minor of G that represents the distance between the horizontal classes of equivalence
V is the minor of G that represents the distance between the vertical classes of equivalence
V

So we could be tempted to restrict our search to the edges that realize the distances between classes of equivalence, however they need not to be connected, as the following image shows



In red and blue are the edges that realize the distances of both MST_H and MST_V, but they are not a connected subset of edges. To the right, by adding in a smart way a couple of segments to the solution, now we obtain a connected subset that guards all the map, but still there are some unnecesary segments to remove. Now, another important observation is that when the minimum spanning trees are calculated, some of the edges that realize the distance between two clasess of equivalence are *unique*, they are like *bridges* connecting two classes of equivalence in a minimal way.

So far, what we have been trying to describe is a way to calculate a lower bound that can give us a hint of how close or how bad is our solution, from OPT. Also, this insight might help us to guide the search by trying to use this *minimal connectors* between classes of equivalence, but that is work is still to be done.

2 Implementation

The interface consists basically on a grid where the user can *hand-draw* a map by clicking segments on it, or press the *generate random map* button, then the user can construct manually a solution or ask the program to perform the aforementioned heuristic. The chosen programming language for this task was Python due to its easyness and my lack of advanced programming skills. The code runs in a web-based platform named Codeskulptor, developed at Rice University by Scott Rixner. The advantage of this platform is that there is no software installment requirement, although it has some limitations in terms of speed and the need of an internet connection. At a later stage the code can be migrated to a different platform. The code is roughly 700 lines of code, including comments, and a Kruskal implementation by Carl Kingsford, and a graph class by Bernd Klein (this code was not actually used but is intended to be used at a later stage and the authors are fully credited for it).

The code is divided basically in three parts

- Helper Functions
- Map Class
- Event Handlers

2.1 Helper functions

Due to the lack of a library that incorporates basic CG operations, the Helper Functions section consists of a set of functions to perform some basic tests and routines to preprocess data, a brief description is provided following. Let's denote by n the size of the input.

- $sorting\,(List, j)$: sorts a list of tuples according to the jth entry. Receives as argument a list and the entry of reference to perform the sort. This routine is used to sort by weight and by x and y coordinates. It's based on Python's sort subroutine. Runs in $O\,(n \log n)$.
- $orientation\,(p1, p2)$: checks if a pair of points (2-tuples) define a horizontal or vertical segment (or neither), using if statements. Returns h, v or $'neither'$. Runs in $O\,(1)$.
- $classify\,(points)$: this function receives as argument a list of paired tuples and creates for each pair a list to store the following information: [p1, p2, [intersected], color, 'h' or 'v', idx, class of equivalence, min_most, weight]; this information is updated during the algorithm. For example, the nested list [intersected] stores the list of its neighbors. Returns two lists $[H, V]$ and runs in $O\,(n)$.
- $overlap\,(ab, cd)$: checks if two intervals overlap, returns $True$ or $False$ in $O\,(1)$.
- $equivalence\,(List, j)$: this function creates a dictionary and groups the elements of a list according to some value stored in thejth entry of those elements (lists also).

- *merge*(x, y): this routine merges two collinear segments, returns the new segment. This function is used as a subroutine in the random map generator, so that randomly created overlapping collinear segments merge into one only. Runs in $O(1)$.
- *random_segment (seg)* this function receives as argument a segment and then creates a random segment incident to *seg*, constrained to the dimension of the canvas, in either horizontal or vertical direction. Runs in $O(1)$.
- *set_random_seg (seg, j)*: this routine constructs incrementally a random map, by calling j times the subroutine $random - segment$, creating a random segment based on the previously created.

Other helper functions include $feasible - neighbor$, *split* (chops the line segments into edges), $color - cce$, $tail - test$, $cce - label$, $next - route - seg$, $make - dictionary$, that are briefly documented in the code.

2.2 Map Class

To be able to handle the associated data structure of the map and the related operations, it seemed natural the creation of an object to store the segments (as a list, although in a later stage a dictionary will provide more flexibility) with some methods to manipulate its content. By initializing it, it receives a list of paired points, or creates an empty list to which points shall be added later.

- *__class_of_equivalence* ( ): it is a static method that determines the horizontal and vertical classes of equivalence for the segments of the map and groups them accordingly. Calls helper functions such as *sorting, classify, merge, equivalence.*
- *__lexicographic* ( ): as its name suggests, this static method sorts both horizontal and vertical segments in a lexicographical ordering.
- *__merge* ( ): this static method is basically used to avoid redundant overlapping segments.
- *get_length* (color): returns the sum of the lengths of the edges in the map colored as "color".
- *paint*(seg, color): paints a given segment (seg) of the map to the desired "color".
- *update*( ): updates a map after other segments might have been added or deleted.
- *undo*( ): deletes last segment added.
- *__guarded_segments* ( ): static method that is called whenever a segment is painted to mark properly the *guarded segments*
- *add_segment* ( ): adds a pair of points to the list. Ideally the stored data should be sligthly modified to insert the new segment, but for sake of simplicity, the map is updated just before performing the algorithm.
- *draw_map* ( ): method to draw the map in the canvas when it is called.

2.3 Event handlers

This is the interactive part of the code.

- *click*( ): receives the clicks of the mouse on the canvas as points. When drawing, it calls the add_segment method of the current map, to insert a pair of points as segment. When the game mode is on, it calls the method paint ( ) instead, to color the respective segment in the map.
- *draw*( ): creates a canvas and manages all the drawing routines.
- *reset*( ): resets the program when its associated button calls this function when pressed.
- *guarding_route*( ): this function is the core of the algorithm, it performs one loop every time it is called. If it is the first loop it sorts the segments of the map by weight and picks the smallest one and paints it *green.* Then it creates a list *bag* to store the discovered neighbors at each step. Ideally this list can be maintaned in $O(\log n)$ time, but for sake of simplicity it is sorted completely every time it is updated at each loop, because also at each step new edges are guarded and they might become "non-feasible" in which case they should be removed from the list of neighbors (never to come back again).

Other event handlers are: $game\_mode(\ )$, which turns on and off this feature; $random\_seed(\ )$ gets a number that the user inputs in a field-input, this is the number received as j argument when generating a random map; $clean(\ )$, resets all the parameters and cleans the screen; $random\_map(\ )$ calls the constructor function of a random map when the associated button is pressed.

In summary, the code was implemented without caring too much about optimal performance, but of functionality. Further improvements can be done to the code at a later stage, by using more sophisticated data structures and more careful implementations.

3 Future work and refinements

It is left to implement the routines that call Kruskal's MST function and a method in Map class that would return the correspondant graph to compute the lower bound to the heuristic. Implement routines in an optimal fashion. Also, if it is possible to package the software as a standalone, maybe in another language as Java or C, that would be preferable Also, take into account those *bridges* that appear as the only segments that connect two classes of equivalence. Finally, test the code exhaustively for debugging.