# A12: Spatial Graphs, Shortest Paths, Binary Heaps, and kd-Trees

April 11, 2019

## 1 Motivation

The following figure, copied from a relatively recent thesis [1], illustrates the inefficiency of a conventional implementation of Dijkstra's shortest path algorithm:

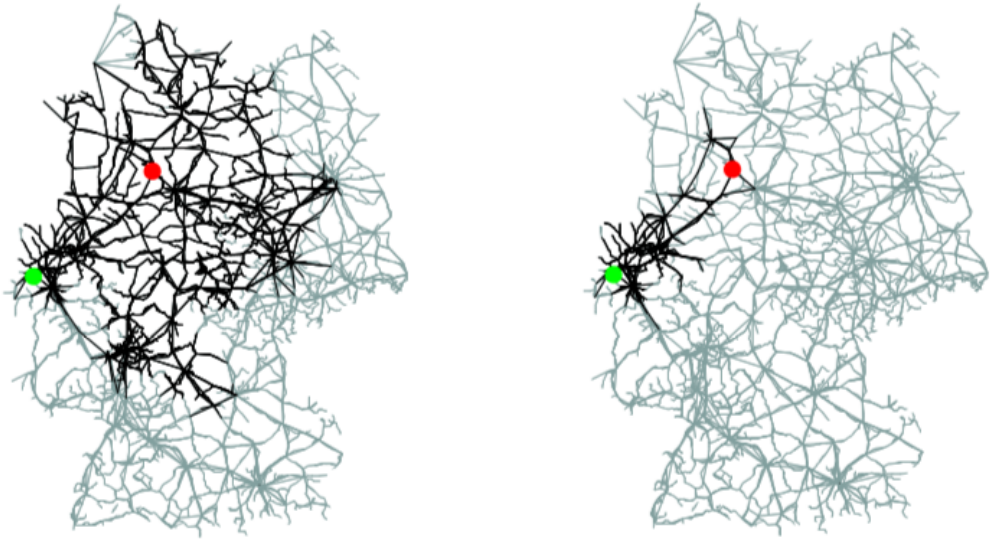
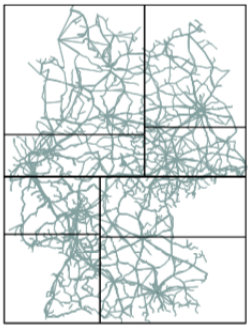


Figure 1: This figure illustrates the search space of a standard DIJKSTRA search (left) and an accelerated partition-based DIJKSTRA search (right). The source node is marked red, the target green, all touched edges during the search are drawn bold. The search stops, when the target node has been reached.

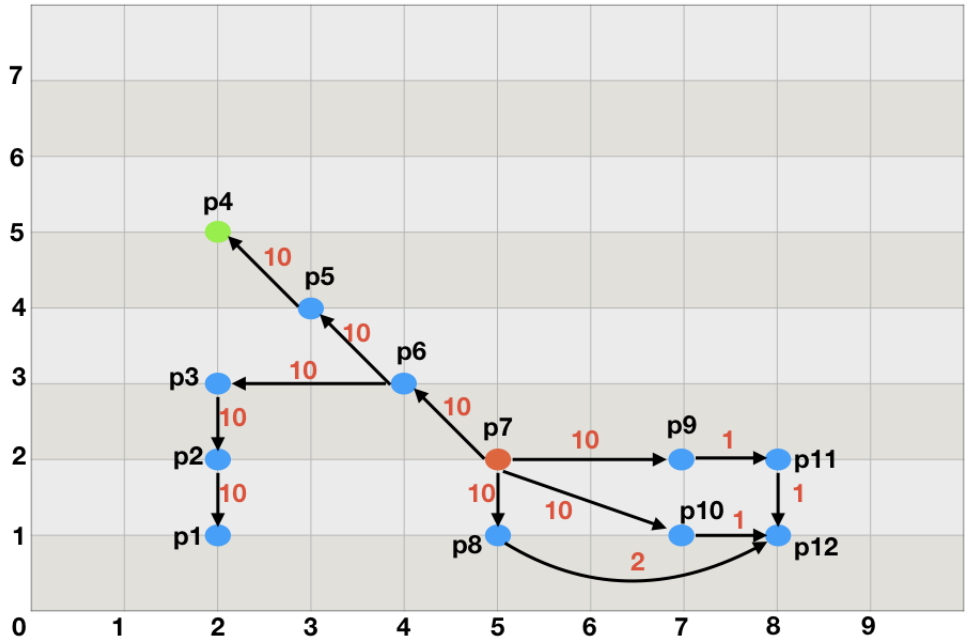
In the left map, the visited edges essentially form a circle around the start node. The intuitive reason is that the destination node (in green) cannot be safely extracted from the binary heap until every node that is closer to the source has been extracted. In the right map, the search is very much directed from the source to the destination. Our aim is to implement such a search.

The key idea will be to partition the map using a kd-tree as shown to the right. When computing shortest paths, we only consider edges that might lead to the destination's region. The information about which edges can lead to which regions is calculated once and for all during a preprocessing phase.



## 2 A Small Detailed Example

Consider the following graph:

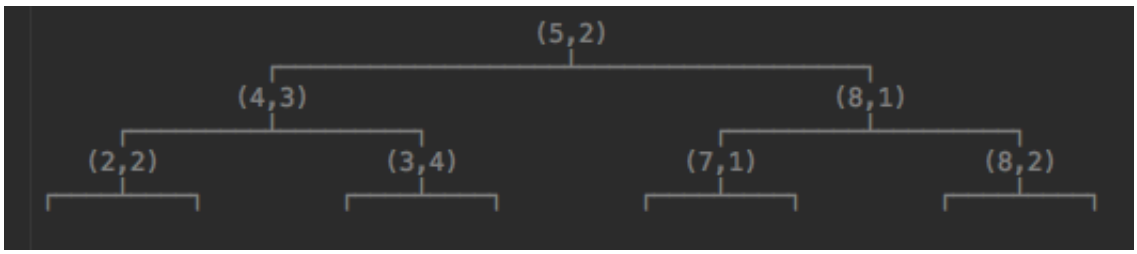



The graph consists of nodes, edges, and weights as usual, but the nodes additionally are given positions in a 2D grid. Say that we would like to use Dijkstra's shortest path algorithm to find the shortest path from **p7** (in red) to **p4** (in green). It is clear that the shortest distance is 30.

In the first few steps of the algorithm, all the neighbors of **p7** will be extracted. The point **p4** will not be extracted until after **p11**, **p12**, and **p3** have all been extracted as their distances from the source are less than 30.

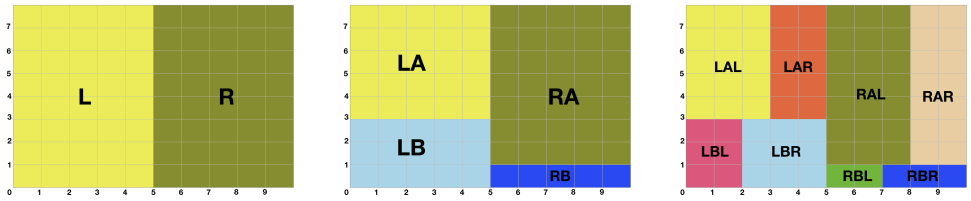
To optimize the search, we will do the following steps:

- Organize the points in a *balanced* kd-tree. The way we will force the tree to be balanced is by splitting at the median at every level until we reach a small enough number of points. In our case, the decomposition produces the following tree where we stopped splitting when the number of nodes in the relevant dimension was 0 or 1:

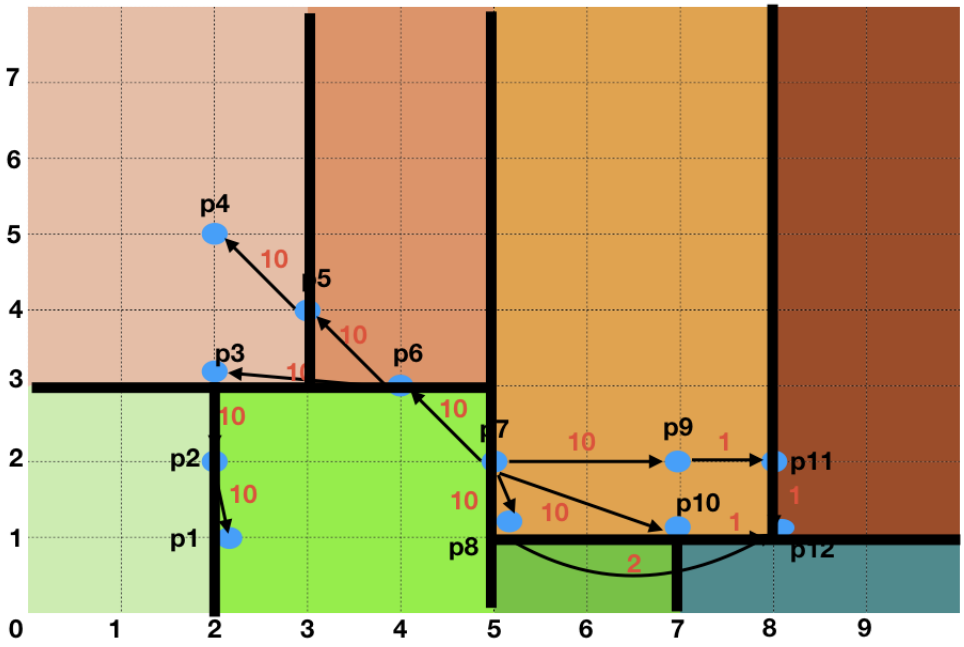


A nice explanation of the construction is available at `https://courses.cs.washington.edu/courses/cse373/02au/lectures/lecture22l.pdf`

- We will refer to each region by the sequence of L(eft), R(ight), A(bove), and B(elow) choices needed to reach it from the root of the kd-tree. The empty sequence [] refers to the entire grid:

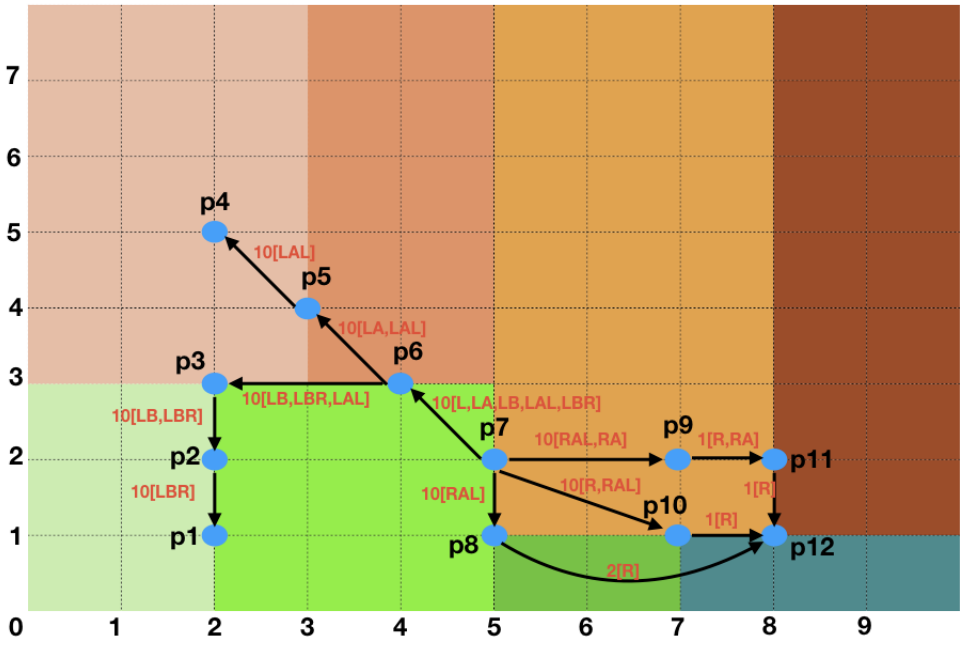



- Superimposing the kd-tree on the graph produces the following picture:




in which it is made explicit which points are in which regions. For example, the point **p1** is in the region **LBR**. The points **p3** and **p4** are in the region **LAL**. The points **p8**, **p9**, and **p10** are all in the region **RAL**. The other points lie on a splitting line and are in larger regions. For example, the point **p2** is in the region **LB** as it the point which splits it into **LBL** and **LBR**.

- The pre-processing labels the edges as follows:




The edge from **p7** to **p8** is never a part of a shortest path leading to **p12**. It is not labeled with the region for **p12** (region **R**). The edge from **p7** to **p6** can never be part of a shortest for any node in the right region: all its labels start with **L**.

- Say that we trying to calculate our original shortest path from **p7** to **p4**. We first note that **p4** is in the region **LAL**. When we relax the edges starting at **p7**, we only follow the edges whose label includes **LBL**. The only such edge is the edge leading to **p6** so we never consider any node in the right half of the grid.

# 3 Implementation Steps

We recommend that you implement your solution by working in the following order. Do not move to the next file until you have thoroughly tested the previous one!

- `Point.java`: a simple class implementing 2D points. The class is complete except for one missing method (`distanceSquaredTo`) which was written in a previous assignment. Finish the implementation of the class, test it thoroughly, and then move on.
- `Edge.java`: a simple class implementing a directed "edge" as a pair of two points (`from`, `to`). Again the class is complete except for one trivial missing method (`flip`) which simply returns an edge in the opposite direction. Finish the implementation of the class, test it thoroughly, and then move on.
- `Rect.java`: this class implementing a "rectangle". It is missing several methods but are all identical to methods you wrote for a previous assignment. Complete the implementation of the class, test it thoroughly, and then move on.

- `Region.java`: this is a new class. It is rather simple though. As illustrated above, a region is represented by a list of directions: `LEFT`, `RIGHT`, `UNDER`, and `ABOVE`. There are two methods to write: the one that requires a bit of thinking is `push` which should add the given direction to the front of the existing sequence.

- `Item.java`: the "item" class is almost identical to the one we had in earlier assignments. The main difference is that there is an additional field `previous` which points to another item. This field will be set during the shortest path traversal. If the item is connected to the source, the field `previous` will point to the previous item along the path to the source. Otherwise, if the current node is the source or if the current item is disconnected from the source, the field `previous` will be `null`. For now, we assume the field is properly set and we are only concerned with writing a new static method `pathToSource`. The method takes an item `u` and keeps chasing the `previous` pointers until one of them is `null`. It returns a (possibly empty) sequence of edges from the given point until the `previous` field is null. (Note that these edges are "backwards" from their definition in the original graph.)

- `BinaryHeap.java`: This class should be identical to the one you have previously written.

- `KDTree.java`: For this file, we removed many of the previously written methods that are not needed for this assignment, and added three new methods: `makeXTree`, `makeYTree`, and `findRegion`.

  - The method `findRegion` is relatively simple. A small example should make it clear. Say the current kd-tree was constructed using `new XNode(new Point(5,2),new YEmpty(),new YEmpty())`. You are then given a point $p = newPoint(x, y)$ and asked to return its region. If $x = 5$ and $y = 2$, then the point is the splitting point of the entire region; its region is modeled by an empty sequence of directions. If $x < 5$, then the point is the left region, and if $x \geq 5$, then the point is in the right region.
  - The methods `makeXTree` and `makeYTree` are similar to each other. We explain the first in detail. You are given three parameters `xPoints`, `yPoints`, and —bound—. The lists `xPoints` and `yPoints` contain the same set of points but one is sorted along the $x$-dimension and the other is sorted along the $y$-dimension. The `bound` parameter is used to truncate the construction of the kd-tree. Once the size of the list of `xPoints` is $\leq$ *bound*, an empty kd-tree is returned. If `bound` is 0, then the tree construction continues until it exhausts all the points. The test case `testKDTree` has a small example with 7 points that you should refer to in the following discussion. The median of the given 7 points along the $x$-dimension is **p4**. So that point becomes the root of the kd-tree. The sublist containing the points **p1**, **p2**, and **p3** will constitute the new set of `xPoints` for the left recursive call, and the sublist consisting of the points **p5**, **p6**, and **p7** will consist of the new set of `xPoints` for the right recursive call. For these recursive calls, we also need to compute the appropriate sets of `yPoints`. This will be done by traversing the original list of sorted `yPoints`, picking up the desired elements in their rank along the $y$-dimension. In other words, for the left recursive call the new sublist of `yPoints` will contain **p2**, **p3**, and **p1** in that order, and the new sublist of `yPoints` for the right recursive call will contain **p6**, **p7**, and **p5** in that order.

- `SpatialGraph.java`: This class is entirely new. The methods you should write are:

  - `allShortestPaths`: this is almost identical to the shortest path method you wrote earlier. The only difference is that we also maintain a path from each node back to the source. In particular, when a node's key is updated with a new distance, we also update its `previous` field to remember the previous node along the currently shortest path.
  - `shortestPath`: this is again almost identical to the shortest path method you wrote earlier with two small modifications. We maintain the `previous` field as above, and we immediately terminate the computation once we reach the destination.
  - `buildKDTree`: This method should call `makeXTree` with appropriate parameters.

- `preprocess`: The first step in pre-processing the graph is to build the kd-tree using the previous method. The second step is the following involved computation. First initialize `regionalEdges` appropriately. Then for each node `s`:
  - compute `allShortestPaths(s)` with that node as a source, and
  - for each node `d`,
    - calculate the last edge from `d` back to `s` (if there is one),
    - flip it,
    - calculate the region for `d`,
    - update `regionalEdges` by adding the region for `d` to the edge originating from `s` towards `d`.
- `regionalShortestPath`: this is a conventional shortest path method with one additional twist: calculate the destination's region, and only consider edges that are annotated with that region.

Small useful tips:

- You are also provided with `TreePrinter.java` which we used before to display trees. It is useful for visualizing your `KDTree` solution.
- Two objects that are `equal?` must return the same `hashCode()`.
- Every shortest path method should start with an explicit initialization to reset all the nodes and set all the nodes' values to `Integer.MAX_VALUE` except the source's value which is 0.
- Watch out for the fact that `Integer.MAX_VALUE + 1` is a negative number and hence less than `Integer.MAX_VALUE`.
- Make sure to clone the list of points before sorting it along another dimension.

# References

[1] Schütz, B. Partition-based speed-up of Dijkstra's algorithm. Master's thesis, Universtät Karlsruhe, 2004.