Lecture 6
Algorithms on graphs.
Path search algorithms on weighted graphs

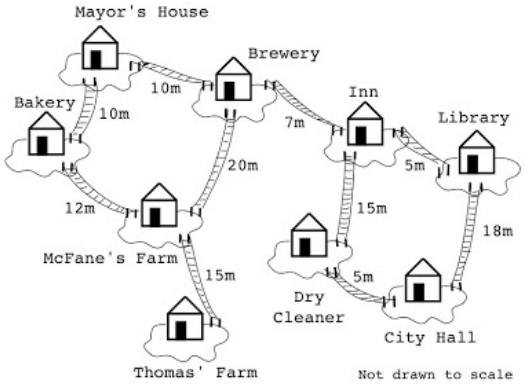
Analysis and Development of Algorithms

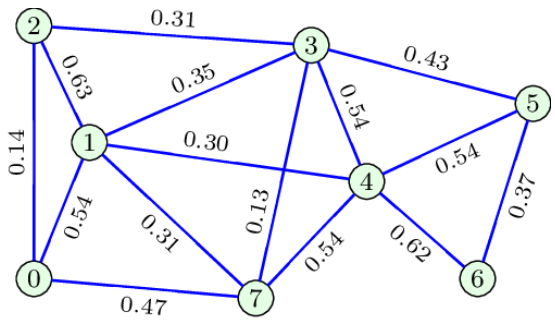
УНИВЕРСИТЕТ ИТМО

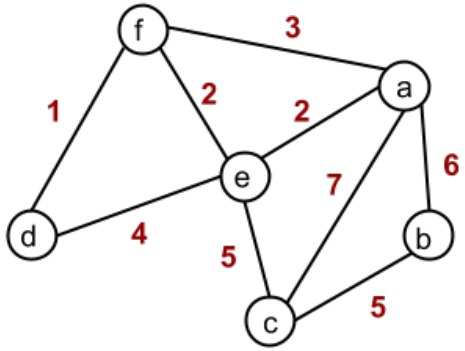
Overview

Weighted graphs

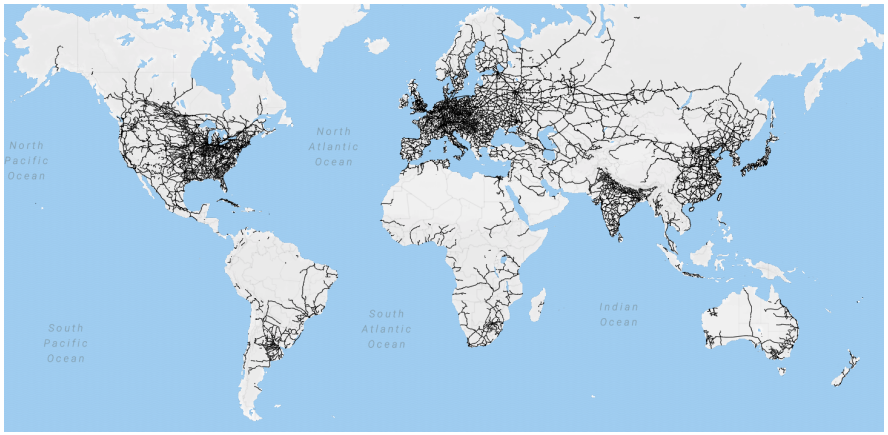
A *weighted graph* is a graph in which a *weight* is assigned to each edge.



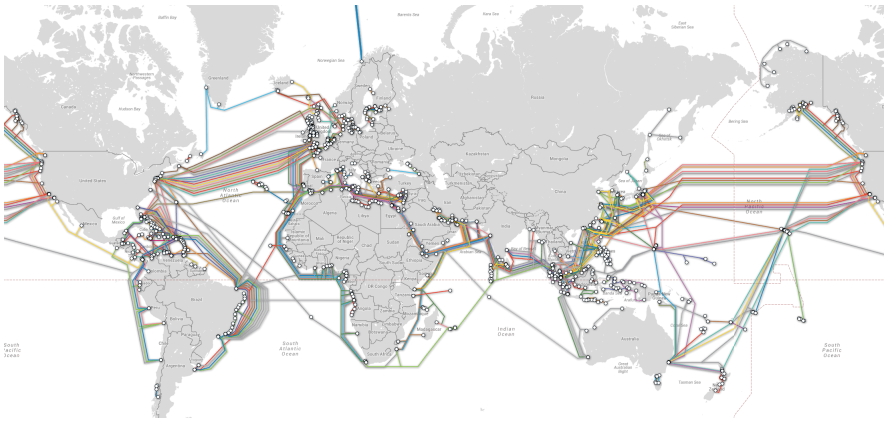




Railway network



Undersea Cable network



Dijkstra's algorithm

Problem: given a weighted graph (with *positive* weights) and a source vertex, find shortest paths from the source to all other vertices.
Main idea: It generates a shortest path tree (SPT) with the source as a root, with maintaining two sets: one set contains vertices included in SPT, other set includes vertices not yet included in SPT. At every step, it finds a vertex which is in the other set and has a minimum distance from the source.

Algorithm (time complexity is from $O(|V| \log |V|)$ to $O(|V|^2)$)

- Create an SPT set *sptSet* that keeps track of vertices included in SPT, i.e., whose minimum distance from source is calculated and finalized. Initially, this set is empty.
- Assign a distance value to all vertices in the input graph. Assign the distance value for the source vertex as 0. Assign the distance value for other vertices as ∞.
- While *sptSet* does not include all vertices:
  - Pick a vertex $u \notin sptSet$ that has a minimum distance value.
  - Include u in *sptSet*.
  - Update the distance values of all adjacent vertices of u. To update the distance values, iterate through all adjacent vertices. For every adjacent vertex v, if the sum of distance value of u (from the source) and weight of edge (u, v) is less than the distance value of v, then update the distance value of v.

Edsger Dijkstra, in an interview with Philip L. Frana, Communications of the ACM, 2001

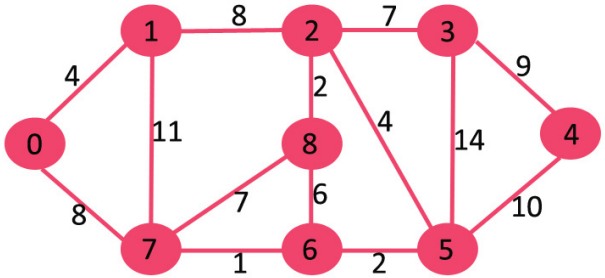
What is the shortest way to travel from Rotterdam to Groningen, in general: from given city to given city. It is the algorithm for the shortest path, which I designed in about twenty minutes.

One morning I was shopping in Amsterdam with my young fiancée, and tired, we sat down on the café terrace to drink a cup of coffee and I was just thinking about whether I could do this, and I then designed the algorithm for the shortest path. As I said, it was a twenty-minute invention.

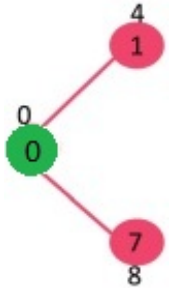
In fact, it was published in '59, three years later. The publication is still readable, it is, in fact, quite nice. One of the reasons that it is so nice was that I designed it without pencil and paper. I learned later that one of the advantages of designing without pencil and paper is that you are almost forced to avoid all avoidable complexities.

Eventually that algorithm became, to my great amazement, one of the cornerstones of my fame.

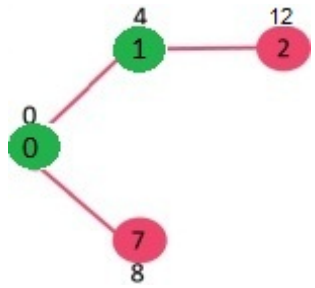
Example



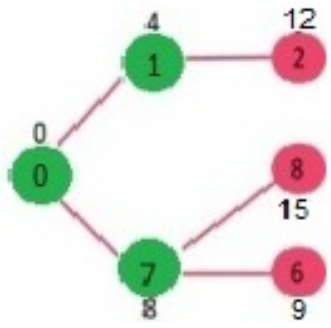
Initially, *sptSet* $= \emptyset$, the distance assigned to the source vertex 0 is 0 and the distance assigned to other vertices is ∞.
Below, the subgraph shows vertices and their distance values (only if they are finite). The vertices included in *sptSet* are shown in green.



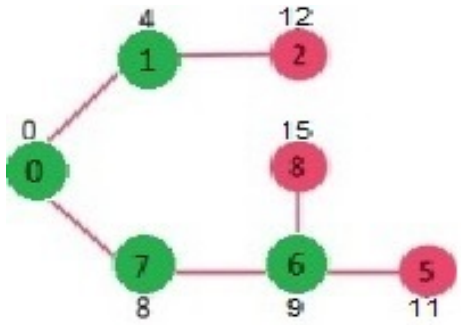
Pick the vertex $\notin$ *sptSet* with minimum distance value. The vertex 0 is picked and included in *sptSet* so *sptSet* $= \{0\}$. Update distance values of 0's adjacent vertices. Adjacent vertices of 0 are 1 and 7. The distance values of 1 and 7 become 4 and 8 respectively.



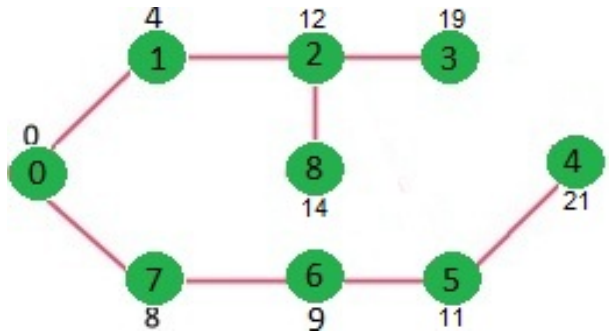
Pick the vertex $\notin$ *sptSet* with minimum distance value. The vertex 1 is picked and included in *sptSet* so *sptSet* $= \{0, 1\}$. Update the distance values of 1's adjacent vertices. The distance value of vertex 2 becomes 12.



Pick the vertex $\notin$ *sptSet* with minimum distance value. Vertex 7 is picked and included in *sptSet* so *sptSet* $= \{0, 1, 7\}$. Update the distance values of 7's adjacent vertices. The distance values of vertices 6 and 8 become 15 and 9 respectively.



Pick the vertex $\notin$ *sptSet* with minimum distance value. Vertex 6 is picked and included in *sptSet* so *sptSet* $= \{0, 1, 7, 6\}$. Update the distance values of 6's adjacent vertices. The distance values of vertices 5 and 8 become 11 and 12 respectively.



Repeat the above steps until *sptSet* does include all vertices. Finally, the indicated SPT is obtained.

Demonstration: another example, larger graph

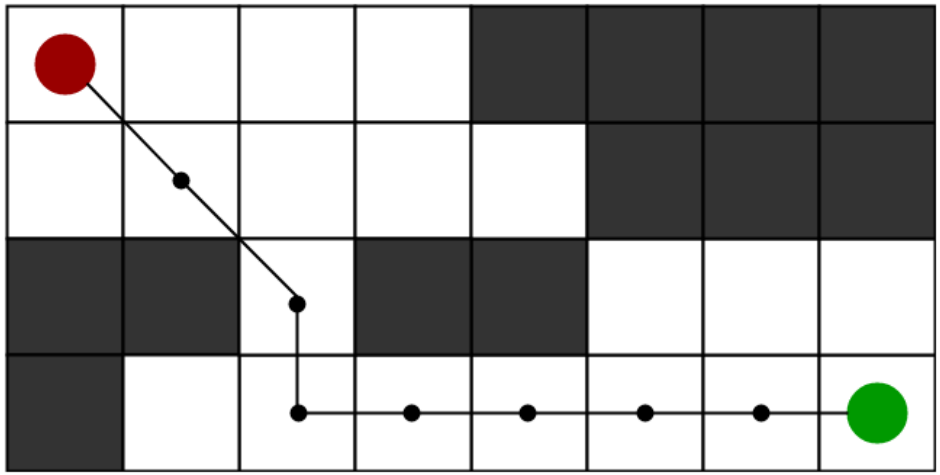
A* algorithm

Problem: given a weighted graph (with *positive* weights), a source vertex and a target vertex, find a shortest path from the source to the target.

Main idea: At each iteration, A* determines how to extend the path basing on the cost of the current path from the source and an estimate of the cost required to extend the path to the target. Time complexity is $O(|E|)$.

One gets Dijkstra's algorithm if the above-mentioned estimate is skipped.

Consider a grid with obstacles that can be represented as a weighted graph (vertices=cells, weighted edges=shortest paths and their lengths).



In general, movements to different cells may have different cost.

A* algorithm

We are given a square grid with obstacles, a source cell c_s and a target cell c_t. The aim is to reach c_t (if possible) from c_s as quickly as possible.

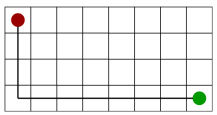
A* extends the path according to the value of $f(c_k) = g(c_k) + h(c_k)$, where g measures the cost to move from c_s to c_k by the path generated and h measures the cost to move from c_k to c_t.

How to calculate h?

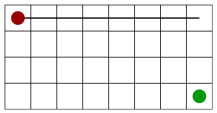
Exactly (time consuming): pre-compute the distance between each pair of cells before running A*.

Approximately: calculate the distances assuming no obstacles, e.g.

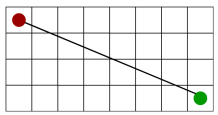
Manhattan

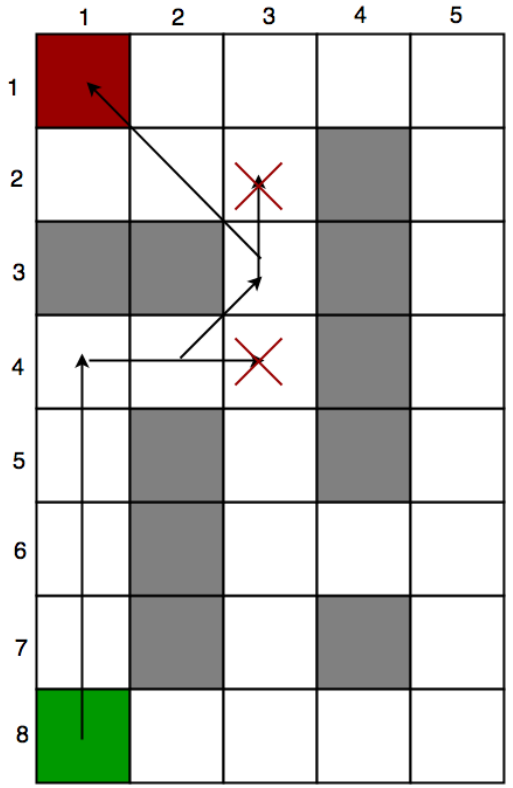


Diagonal



Euclidean





A* Search Algorithm makes the most intelligent choice at each step. Hence you can see that algorithm goes from (4,2) to (3,3) and not (4,3) (shown by cross).

Similarly the algorithm goes from (3,3) to (2,2) and not (2,3) (shown by cross).

Demonstration: A* and its applications

Bellman-Ford algorithm

Problem: Given a weighted graph (possibly directed and with *negative* weights) and a source vertex s, find shortest paths from s to all vertices in the graph. If a graph contains a negative cycle (i.e. a cycle whose edges sum to a negative value) that is reachable from s, then there is no shortest path: *any path that has a point on the negative cycle can be made shorter by one more walk around the negative cycle*. In such a case, Bellman–Ford can detect the negative cycle.

Note: Dijkstra does not work for negative weights. Bellman-Ford is simpler than Dijkstra but its time complexity is $O(|V||E|)$.

Idea: At ith iteration, Bellman-Ford calculates the shortest paths which have at most i edges. As there is maximum $|V| - 1$ edges in any simple path, $i = 1, \ldots, |V| - 1$. Assuming that there is no negative cycle, if we have calculated shortest paths with at most i edges, then an iteration over all edges guarantees to give shortest paths with at most $(i + 1)$ edges. To check if there is a negative cycle, make $|V|$th iteration. If at least one of the shortest paths becomes shorter, there is such a cycle.

Demonstration: a step by step example, an example with a negative cycle

Thank you for your attention!