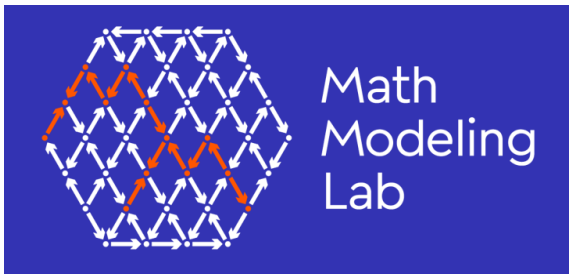
Lecture 7. Breadth First Search on Graphs

Dmitri Shmelkin

Math Modeling Competence Center at Moscow Research Center of Huawei
Shmelkin.Dmitri@huawei.com

April 21, 2023



Overview

1. Graph presentations
2. Bread First Search as concept and examples
3. Shortest Path Problem
4. BFS Traverse
5. Dijkstra Algorithm
6. Bellman-Ford Algorithm

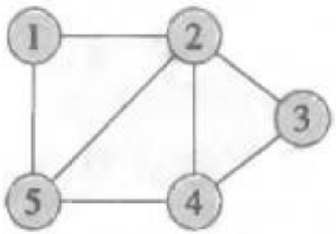
Graph presentations

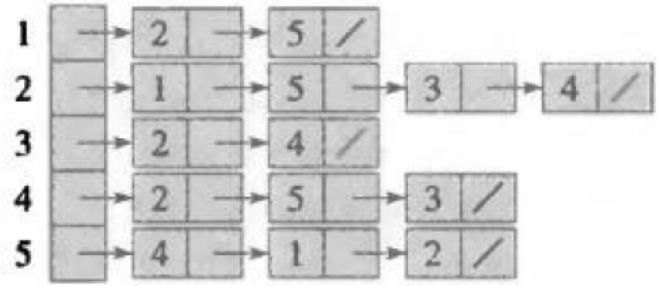
• Graph may be directed or undirected, which may be thought of as bi-directed.
• In both cases a graph, G, is a set of nodes or vertices, V, and that of edges, E, and the presentation of edges may be different:
• The edges of a graph with $|V| = n$ may be defined via an $n \times n$ adjacency matrix A such that $A[i, j] = k$ if there are k edges from i to j. In this respect, if G is undirected, then A is supposed to be symetric.
• The size of adjacency matrix is n^2 and in most cases the graphs may happen to be *sparse* so that $|E| \ll n^2$. Therefore, the edges may be stored as adjacency lists of neighbour nodes, per each node, for example as dynamic arrays (vectors).
• For directed graphs, one may consider 2 adjacency lists, for incoming and for outgoing edges, separetely. Or, to avoid redundancy, it is enough to store only the list of nodes, **to which** we may pass by an outgoing edge.
• The latter data type also applies to undirected graphs thought of as directed.
• The graphs in many cases may also have weights assigned to the edges.
• If adjacency lists are used, the weights are stored together with the edge.
• If the graph has no multiple edges (of the same direction), then the adjacency matrix may hold the weight as $A[i, j]$, instead of the edge multiplicity.

First of all, graph are *graphical* objects and the most intuitive is to draw a picture

Graph presentations: examples

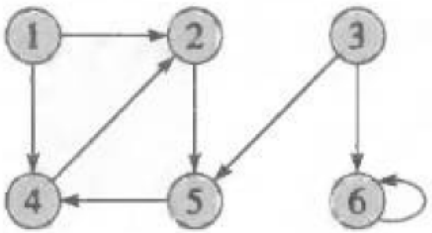
A graph, from picture to adjacency lists and matrix:

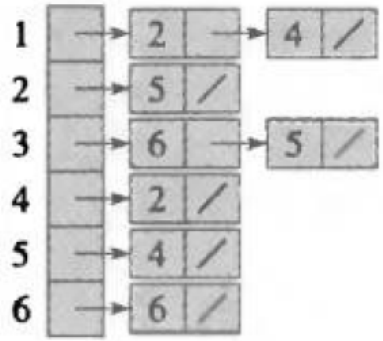




	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

A directed graph, from picture to adjacency lists and matrix:





	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

And how to return from adjacency data to a picture?

Drawing a graph: Planar graphs

In many practical situation a graph comes defined via its adjacency data and the problem occurs of how to draw the graph on $\mathbb{R}^2$, that is to assign (x, y) coordinates to the vertices such that the picture will be nice?
Different formalization of the above "nice" are possible for example this one:
Graph Drawing: Input is a graph, $G = (V, E)$ given as adjacency data. Output is a map $V \to \mathbb{R}^2$ such that the number of pairs $(e_i, e_j) \in E^2$ of edges without common vertices but having an interior intersection is to be minimal.

Definition 1

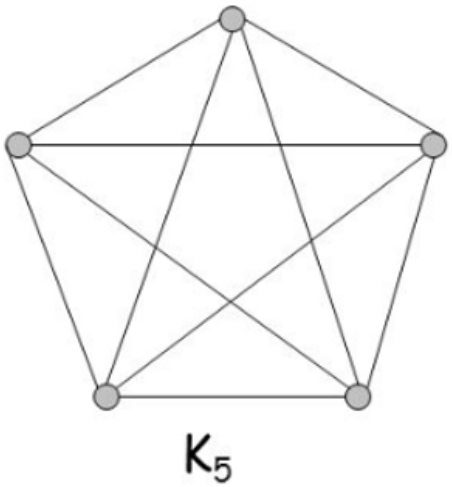
A graph is called *planar* if Graph Drawing solves with zero intersections.

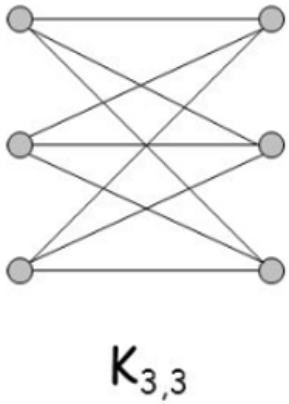
Theorem 2

[Kuratowski, 1930]: A graph is planar if and only if it does not contain a subgraph that is a **subdivision** *of either of K_5, a full graph on 5 vertices, or $K_{3,3}$ a full bi-partite graph on 3 and 3 vertices.*

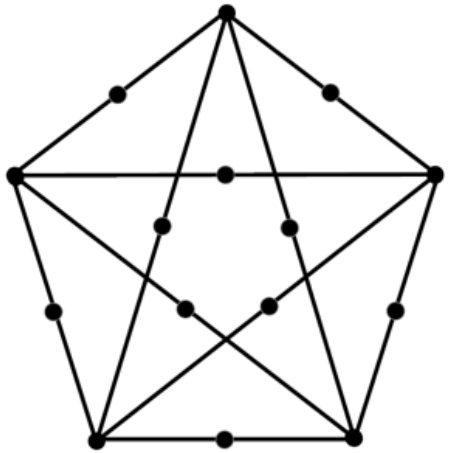
To check the condition of Kuratowski Theorem, an algorithm was proposed in [Williamson 1984] based on the main topic of this lecture, BFS.

Drawing a graph: non-planar graph examples





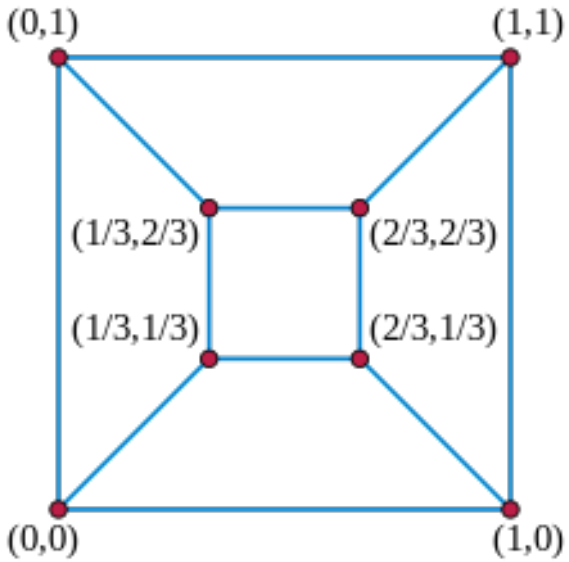
Minimal non-planar graphs K_5 and $K_{3,3}$.



A subdivision of K_5.

Drawing a graph: Tutte's Spring Theorem

In [Tutte, 1963] special maps of planar graphs to $\mathbb{R}^2$ were defined such that:
1. The convex hull of the points is a *face* of the planar graph, that is, the vertices and the edges of the hull are in the image of V and E, respectively.
2. Each *inner* vertice is the **baricenter** of its neighbours in the graph.
If the outer polygon is fixed, this condition on the interior vertices determines their position uniquely as the solution to a system of linear equations. Solving the equations geometrically produces a planar embedding.
Tutte's **Spring theorem**, [Tutte, 1963] states that this unique solution has always, as a planar graph, zero interior intersections, and moreover that every face of the resulting planar embedding is convex!



Adjacency matrix and paths

• We already pointed out that using adjacency matrix needs $|E|^2$ size, which is a strong drawback vs. adjacency lists provided $|E| \ll n^2$. Does it have advantages?
• An obvious advantage is that checking whether vertices $i, j \in E$ are connected with an edge has $O(1)$ complexity for adjacency matrix and has $O(E)$ complexity for adjacency lists, in worst case.
• It is also important to note that the powers of adjacency matrix encode the (directed) paths on the (directed) graph:

Lemma 3

Let $A[i,j], 1 \leq i,j \leq |V|$ be the adjacency matrix of a graph $G = (V, E)$. Then for every $p = 1, 2, \ldots$ the entry $A^p[i,j]$ is equal to the number of paths from i to j of length p.

Proof: Apply induction on p. For $p = 1$ the statement is the definition of A. Assume the statement is proven for p. Every path of length $p+1$ from i to j is one of $A^p[i,k]$ paths from i to k augmented wih one of $A[k,j]$ egdes from k to j, and this for every $k \in V$, totally $\sum_{k \in V} A^p[i,k]A[k,j] = A^{p+1}[i,j]$ paths. □

Breadth First Search vs Depth First Search

• Recall that in the previous lecture on Binary Trees, important algorihms are based on Depth First Search, a graph walk that goes recursively from a node to one of its children, then to one of grand children until reachs a leaf.
• A similar approach is possible in arbitrary graphs, directed or not, having cycles or not, and it will be considered in the next lecture.
• For a general graph, by lack of well defined layered structure DFS is used in specific cases, but in most cases another policy turns out to be ubiquituos:

Definition 4

Breadth First Search is a method of graph walk that **a)** gets an initial vertex as input **b)** supports a queue of vertices to be visited **c)** iteratively Dequeue the head of queue vertex and **d)** Handles its neighbors adding some of them to the queue.

Remark: Definition 4 is in fact quite broad and embraces many specific algorithms solving different problems, polynomial and even NP-hard. In such a generality it plays in Graph Algorithm the role of universal method that requires specific queue structure in **b)** and handling in **d)** for each specific problem.

BFS for finding graph connected components

• Connected components are an important invariant of **undirected** graphs w.r.t. isomorphism and a useful, although simple, part of more complicated algorithms.
• Finding connected components of a graph is a standard operation that might be thought of as assignment of the connected component index to every vertex $c : V \to \mathbb{N}$ such that the image of V is a segment $[1, C]$ and vertices $i, j \in V$ belong to the same connected component if and only if $c[i] = c[j]$.
Finding Connected Components: Input: a graph G with $V = \{1, \ldots, n\}$ and an adjacency array $A[v]$ assigned to each $v \in V$. Output: the number C of connected components and $1 \times n$ array c with the connected component indices.

1. Initialize: array c with 0 values, FIFO queue Q, $C := 0$;
2. for$(v := 1 : n)$//v starts the next component
3. if $(c[v] = 0)$
4. $C := C + 1$; $c[v] := C$; $Q.Enqueue(v)$;
5. while$(Q.IsEmpty = 0)$
6. $x = Q.Dequeue$;
7. for $(y \in A[x]$ and $c[y] = 0)$
8. $c[y] := C$; $Q.Enqueue(y)$;
9. endif
10. endfor return C, c

BFS for finding graph connected components: analysis

• In the proper sense of Definition 4, BFS is the content of lines **4-8**. It starts with a vertex v that hasn't been included to the previous components, adds v to the simple FIFO queue, then Dequeue x and handles those of x's neighbors y, not yet reached in the previous steps ($c[y]$ is still 0) by setting $c[y] := C$ and Enqueue y.

Theorem 5

Finding connected component is correct and of $O(|V| + |E|)$ complexity.

Proof: Correctness: every vertex gets a positive component index in lines **2-4**, the vertices, reached by BFS indeed belong to the same component. Conversely, assume that u is not reached by BFS starting from v. Consider a path from v to u in the graph and let u_1, u_2 be subsequent on this path such that u_1 is reached and u_2 is not reached by BFS. Since all vertices reached by BFS are put to the queue in line **8**, at some step Dequeue in line **6** returns u_1, then u_2 will be found in $A[u_1]$ with $c[u_2] = 0$ (u_2 couldn't be reached by BFS on previous components). So u_2 will be reached by BFS as well, contradiction. For complexity, notice that the vertex loop in line **2** yields $O(|V|)$ and the BFS checks in **7-8** every edge at most 2 times.

Minimal Spanning Tree

Definition 6

Let $G(V, E)$ be a weighted connected undirected graph with positive weights of the edges, $w : E \to \mathbb{R}_{>0}$. A subgraph $H(V_1, E_1) \subseteq G$ is called a Minimal Spanning Tree if **a)** $V_1 = V$, **b)** H is connected, and **c)** H has the minimal sum $\sum_{e \in E_1} w(e)$ among all subgraphs with **a,b**.

Lemma 7

An MST is always a tree, that is, has no cyclic paths and $|E_1| = |V| - 1$

Proof: Indeed, if a cyclic path existed, we might remove any edge of the path to get a connected subgraph with a smaller total weight. That a connected graph without cycles contains $|V| - 1$ edges is well known and proven by induction. □

• The existence of a MST follows from finitness of the set of all subgraph and connectedness of the graph. MST may be not unique.

Remark: The definition of MST generalizes to undirected graphs with negative weights, if the subgraph is required to be a tree by definition.

MST and graph cut

Definition 8

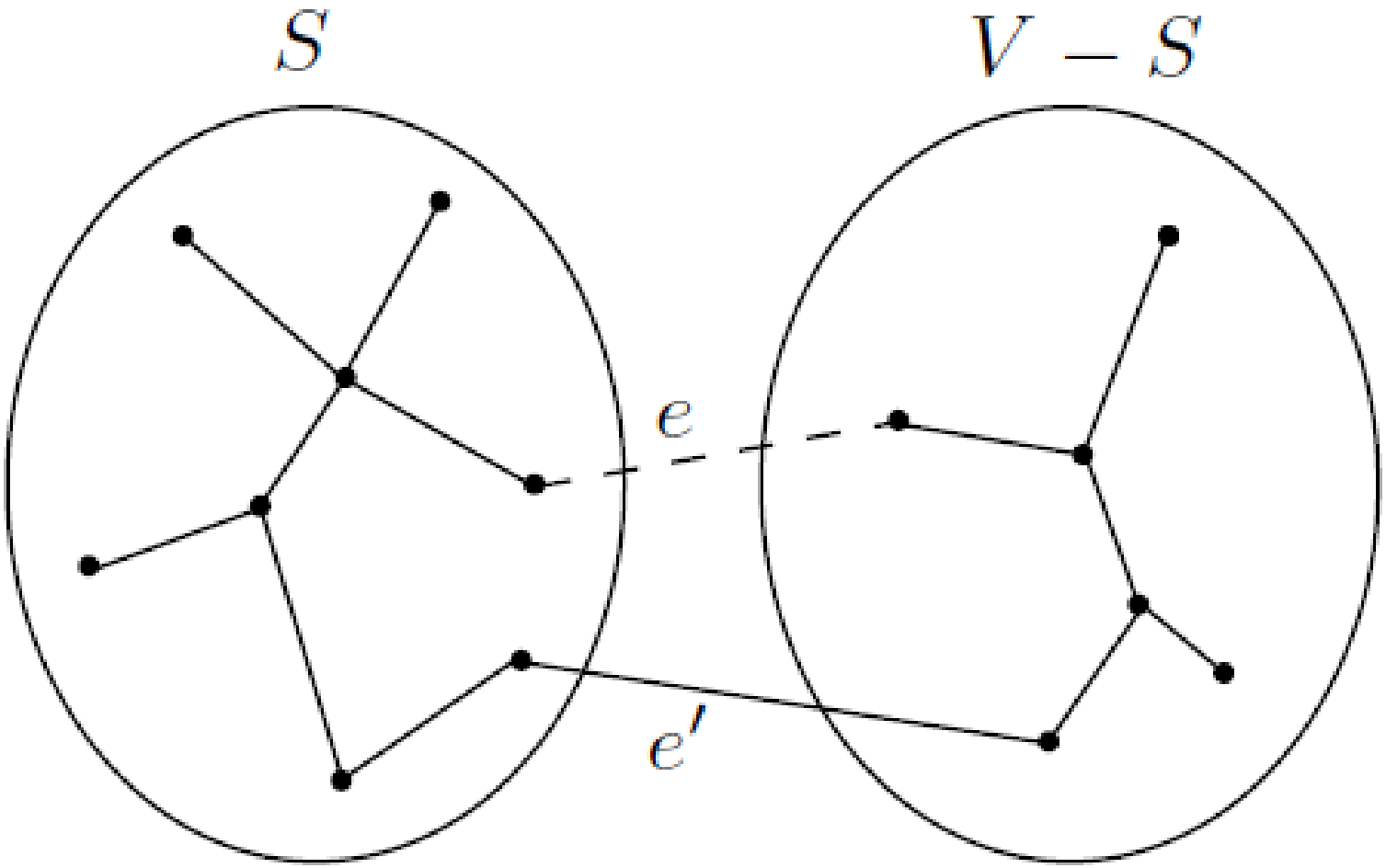
A graph cut is a decomposition of V into $V_0 \subseteq V$ and $V_1 = V \setminus V_0$. For every graph cut an edge is called 0-edge if both vertices belong to V_0 and 1-edge, symmetrically, and 0-1 edge if connecting a vertex in V_0 with a one in V_1.

Theorem 9

Let $S \subseteq E$ be a set of edges belonging to some MST T and assume that S consists of 0- and 1-edges for some cut $V = V_0 \sqcup V_1$. Let e be a 0-1 edge with the minimal weight, out of all 0-1 edges. Then $S \cup \{e\}$ also belongs to a MST.

Proof: Since T is a spanning tree, both vertices of e are connected with a path $P \subseteq T$. Then P contains a 0-1 edge e', and by the choice of e, $w(e) \leq w(e')$. Take a graph $T_1 = (T \setminus \{e'\}) \cup \{e\}$. T_1 is connected because T is connected and both vertices of e' remain connected in T_1 with the the path $(P \setminus \{e'\}) \cup \{e\}$. T_1 contains $|V| - 1$ edges, same as T, hence, T_1 is a spanning tree. Since $w(T_1) \leq w(T)$, T_1 is an MST as well. Since S contains no 0-1 edges, $S \subseteq T_1$.

MST and graph cut



Different approach to building an MST

• Theorem 9 makes it clear that an MST may be built by a sequential algorithm such that every next edge is selected as the one with the minimal weight out of all 0-1 edges for some graph cut. Possible approaches differ in how to define the cut.
• **Kruskal algorithm** starts with the edge of the minimal weight, then at every step adds the edge of the minimal weight out of all edges such that adding this edge does not generate a loop. Indeed, no loop condition guarantees that both vertices of the edge belong to different connected components of the previously selected subgraph, so we may put these components to V_0 and V_1 add put other connected components to either V_0 or V_1 and get that the previously selected edges are either 0 or 1-edges. So by Theorem 9 such a strategy builds an MST.
• **Prim algorithm** starts with some root vertex, r, and adds the edge of the minimal weight incident to r. Then at every step it adds the edge of the minimal weight out of those connecting the the vertices included to the previous tree and those, which aren't yet included. Here, Theorem 9 also applies and proves that the result will be an MST.

Tree and Rooted Tree

- In the context of MST and other problems of this lecture, a Tree is a subgraph in the graph (undirected for MST and or arbitrary, for Shortest Path Problem) that has no cyclic undirected subgraphs (forgetting the direction if existed).
- The above property implies that in a tree a path from one node to another is unique, if exists (may not exist due to direction).
- In all cases of this lecture the tree will have a root vertex and consist of vertices reachable from the root by a (unique!) path in the subtree.
- In pevious lectures we consdiered rooted trees, in particular Binary Search Trees.
- To handle a subtree in the above sense - a subtree $T \subseteq G$ with a root $r \in T$, such that every $v \in V$ is reachable from r by a unique path in T - we reuse the concept of rooted tree, using very small data: an array *prev* of size $|V|$, where $prev[v]$ contains the parent node in T thought of as a rooted tree.
- Children nodes data is typically not necessary to store.

Prim Algorithm: implementation

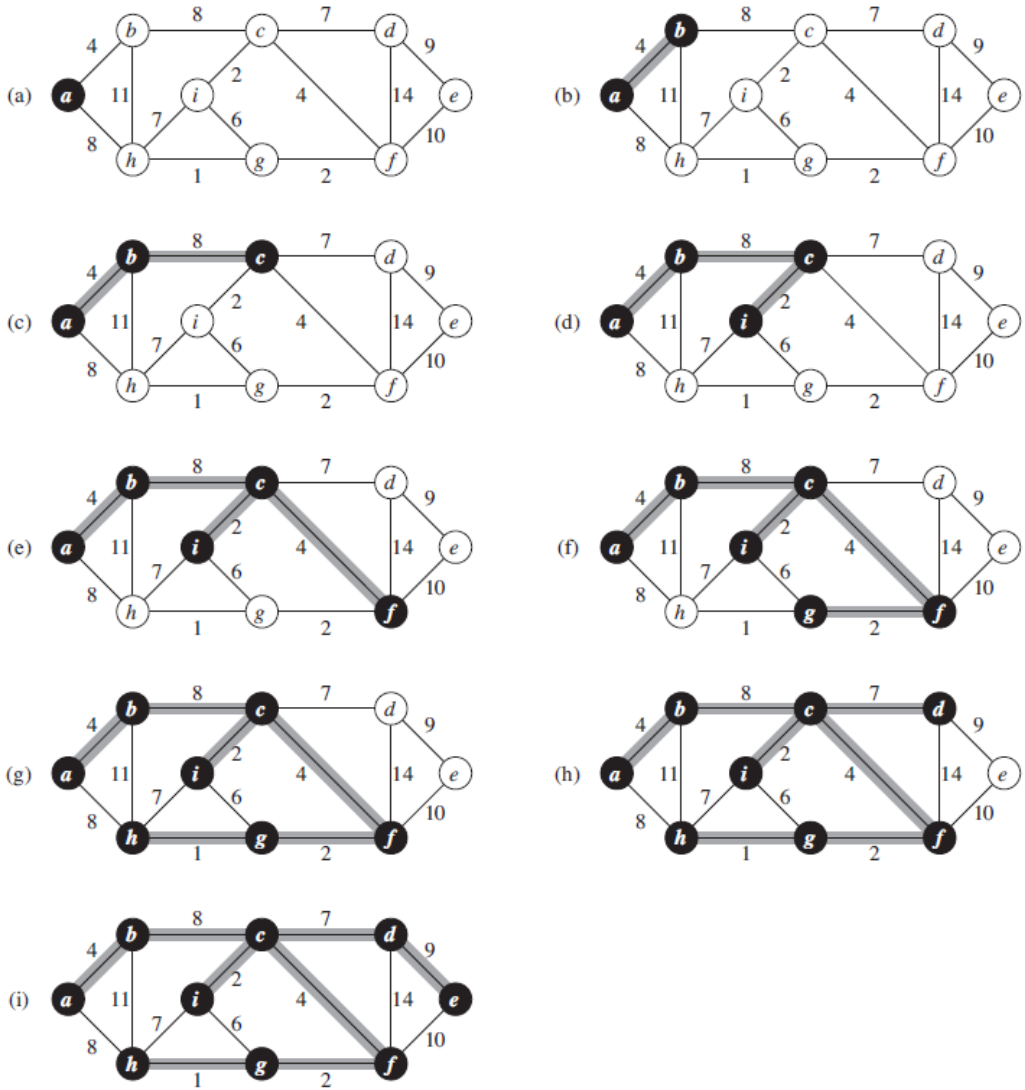
Thoughout the algorithm, we support auxillary data as follows. Per vertex arrays:
- $d[v]$ the minimal weight of edges $u - v$, where u belongs to the current tree.
- $prev[v]$ is the parent in the tree, such that $w(prev[v], v) = d[v]$.
- Flag $InTree[v]$ that is equal to 1 if and only if v is included.
- Priority queue, Q, a Minimum Heap with $d[v]$ as priority.

Prim Algorithm: Input: a weighted graph $G(V, E, w)$ with adjacency lists $A[v]$ and a root vertex, r. Output: the array $prev$ containing MST structure.

1. for $v \in V$ set $d[v] := \infty$; $prev[v] := NULL$; $InTree[v] := 0$;
2. $d[r] := 0$;
3. $Q := BuildHeap(d)$;//only the root r has a finite d
4. while $(Q \neq \emptyset)$
5. $v := Q.ExtractMin$; $InTree[v] := 1$;
6. for $(u \in A[v]$ and $InTree[u] = 0)$//update d of v's neighbors
7. if $(w(u, v) < d[u])$
8. $prev[u] := v$; $d[u] := w(u - v)$; $Q.DecreaseKey(u, w(u - v))$;

Remark: The function $Hash.DecreaseKey$ in Min-Heap is similar to $Insert$ and uses $\log(|Q|)$ complexity.

Prim Algorithm: illustration



Prim Algorithm: analysis

• Prim Algorithm is a BFS in the sense of definition 4. The priority queue contains all vertices from the beginning and the next vertex added to the tree is extracted from the queue, so the queue size decreases at each step.

Theorem 10

The Prim algorithm builds an MST correctly and uses $O(|E|\log(|V|))$ complexity.

Proof: Correctness of the Prim algorithm follows from Theorem 9. There are exactly $|V|$ steps of while loop line **4** and each call of *ExtractMin*, line **5**, takes $O(\log(|Q|)$ so the total complexity of these calls is $O(|V|\log(|V|))$. But the support of d values and changes in heap, lines **6-8** cost more. Throughout the algorithm, each edge is considered at most once in line **8** and the call of *DecreaseKey* uses $O(\log(|Q|))$ compexity, so totally $O(|E|\log(|V|))$. □

Remark: The above implementation is optimal for sparse graphs. For dense graphs, such that $|E| \in \Theta(|V|^2)$ one may use **FIFO simple queue**. Then *ExtractMin* in **5** replaces with $O(|Q|)$ search for minimum, but the total complexity of **6-8** without *DecreaseKey* is $O(|E|)$, so totally, $O(|V|^2)$.

Shortest Path Problem in Graph

• Path in a graph is an important concept and, although the set of all paths in the graph is of exponential size ($n!$ in a full undirected graph), in most applications, the concept of *length* is defined such that finding the shortest paths is polynomial.
• Typically the length is defined as a weight, $w : E \to \mathbb{R}$ so that the path length is the sum of $w(e)$ for e in the path. For non-weighted graph, $w \equiv 1$, so that path length is just the number of edges in it.
• Finding the shortest path in the graph is one of fundaments of Computer science, to which reduce numerous problems from different application domains.
The problem statement, in itself, may vary according to the following parameters:
a) For $w \geq 0$ the problem is well-posed but if some edges have negative weights, the definition becomes non-intuitive and algorithms more complicated.
b) In many cases, one finds not one path from v to u, but a family of paths, from v to each vertex in its connected component.
c) Moreover, there might be several shortest paths between given v, u.
d) In many cases, paths are subject to some constraints, this may make the problem NP-hard so that approximately shortest feasible paths are acceptable.
e) In geometric routing the graph itself is computed while in path search.
BFS applies to all above as a specific algorithm or at least, as a concept.

Existence of the Shortest Paths

Lemma 11

Let $G(V, E)$ be a finite graph weighted with some weight $w : E \to \mathbb{R}$, $u, v \in V$. Then two alternative cases may occur:

1. *If v is reachable from u, then there is a shortest path from u to v.*

2. *G has a cyclic path of negative length.*

Proof: If a negative cycle exists, then passing it several times we may get length that tends to $-\infty$ and for u, v on the cycle there is no a shortest path. Alternatively, if every cycle has non-negative length, then cutting out this cycle does not increase the path length, so, we may look for the shortest path in the subset of paths without cycles, and there are finitely many of these. □

Remark: From now on, we assume to have case **1.** of Theorem 11.

Lemma 12

Let $\delta(u, v)$ denote the shortest path length from u to v if v is reachable, and ∞, otherwise. Then for any $q \in V$ holds: $\delta(u, v) \leq \delta(u, q) + \delta(q, v)$

Proof: is obvious from the definition of δ, including ∞ cases. □

Shortest Paths Properties

Lemma 13

Let $u = q_1, q_2, \ldots, q_k = v$ be a shortest path. Then for every $1 \leq i < j \leq k$ the path $q_i, q_{i+1}, \ldots, q_j$ is a shortest path as well.

Lemma 14

If $u \neq r$ then there is $v \in V$ such that $\delta(r, u) = \delta(r, v) + w(v, u)$.

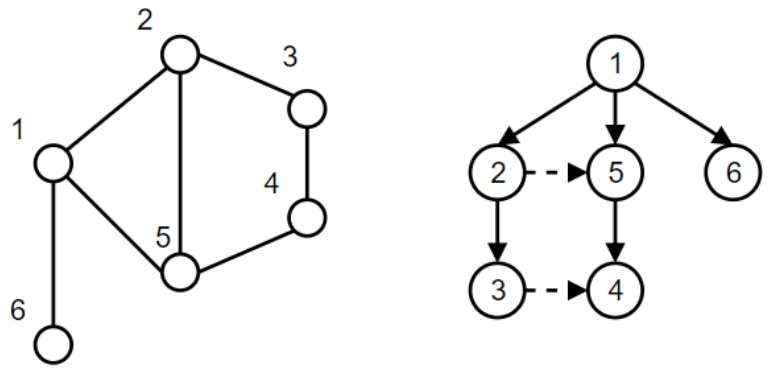
Proof: Take a shortest path from r to u of length $\delta(r, u)$ and let v be the last vertex on the path before u. By Lemma 13, the piece $r - v$ is a shortest path to v of length $\delta(r, v)$, hence, $\delta(r, v) + w(v, u) = \delta(r, u)$. □

Definition 15

Lemma 13 and 14 imply that the shortest paths can be stored with, for every $v \in V$, the previous vertex $prev[v]$ in the shortest path. It means that the shortest path from r to v is the one to $prev[v]$ extended by $prev[v] - v \in E$. These $|V| - 1$ edges (because $Prev[r] = 0$) constitute the **Shortest Path Tree**.

Paths with minimal number of edges

- For a non-weighted graph G, path length is just the number of edges.
- Fix a root vertex $r \in V$ and look for the shortest paths from r to all $v \in V$.
- Agorithm *BFSTraverse* enumerates $v \in V$ by the growing distance from r.
- For every $v \in V$ reachable from r, *BFSTraverse* finds a shortest path and stores all these as a rooted tree using the array *prev* of size $|V|$ such that $prev[u]$ is that vertex v from Lemma 14 such that $\delta(r, v) + w(v, u) = \delta(r, u)$.
- The algorithm supports a *color* of a vertex, $col[v]$ such that initially all vertices are *white*, then become *grey*, finally become *black*.



Undirected graph BFS tree

BFS Traverse

BFSTraverse: Input a graph $G(V, E)$ with the arrays $A[v], v \in V$ of neigbours, to which one can go from v and a root vertex, r. Output, the arrays $d, prev$ with the distance and the shortest path from r to each reachable $v \in V$. Auxillary: color array col with white, grey, and black, FIFO queue Q.

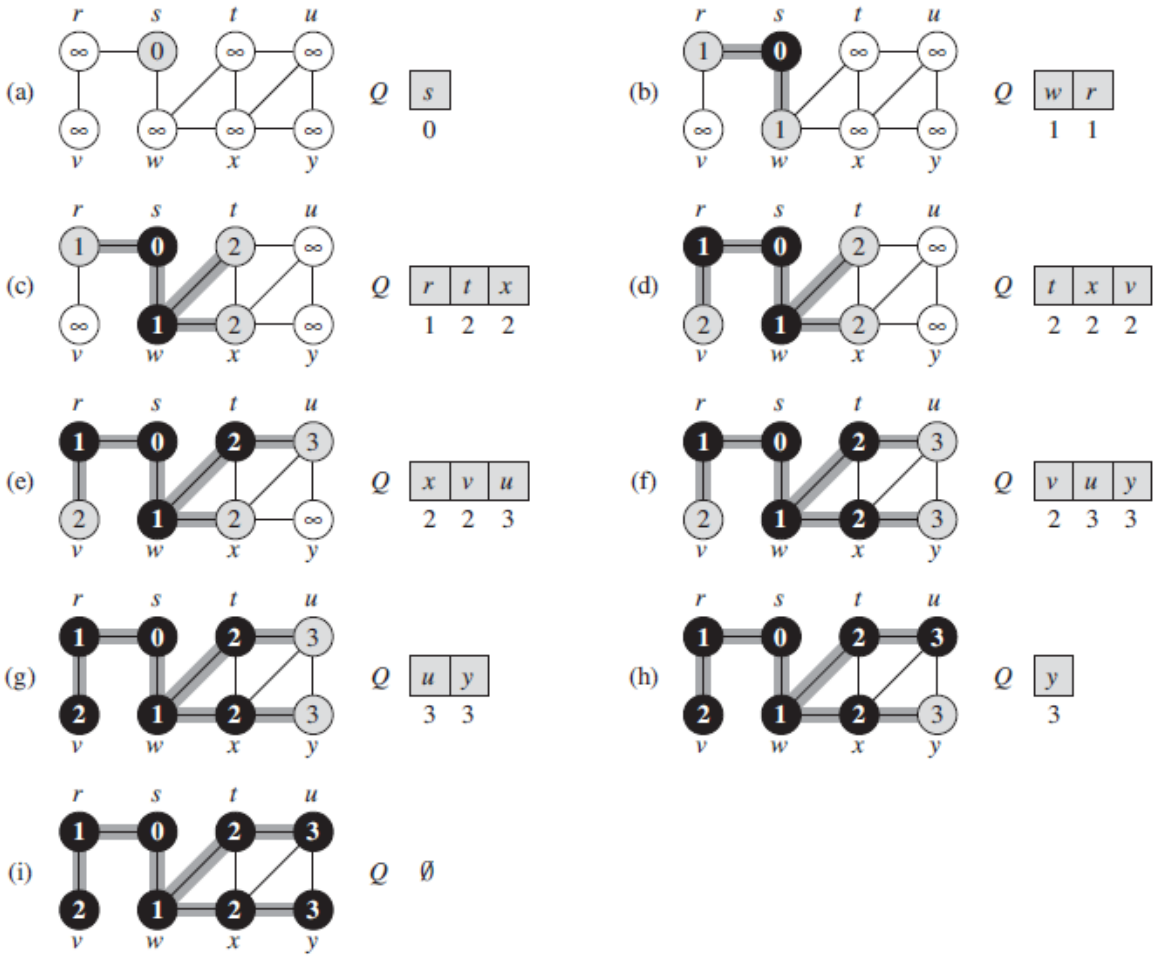
1. for $v \in V$ $d[v] := \infty; prev[v] := 0; col[v] := white;$
2. $d[r] := 0; col[r] := grey; Q.Enqueue(r);$//initialization
3. while $(Q.IsEmpty = 0)$
4. $\quad v := Q.Dequeue;$
5. $\quad$ for $u \in A[v]$ such that $col[u] = white$
6. $\quad\quad d[u] := d[v] + 1; prev[u] := v; col[u] = grey; Q.Enqueue(u);$
7. $\quad col[v] := black;$
8. endwhile//now $d[v] = \delta(r, v), col[v] =$black for all $v \in V$
9. return $d, prev$

Lemma 16

Assignment $d[u] := d[v] + 1$ *in line* **6** *makes* $d[u] \geq \delta(r, u)$.

Proof: By induction follows from triangle inequality Lemma 12. □

BFS Traverse Example



BFS Traverse Properties

Lemma 17

1. *Vertices appear in the queue in the ascending order of the distance* $\delta(r, v)$.
2. *Assignment* $d[u] := d[v] + 1$ *makes* $d[u] = \delta(r, u)$.

Proof: Applying induction on δ we prove both statements. There is a unique vertex with $\delta(r, v) = 0$, this is r and it appears the first in the queue. By Lemma 16 r's neighbors have $\delta \leq 1$ and not 0, so $d = 1$ is the correct δ. Now assume that we proved that all vertices with $\delta \leq k$ appear in the queue ordered by δ and their white neighbors got correct $d = \delta = k + 1$. Then these neighbors appear in the queue before all vertices with $\delta > k + 1$ and by Lemma 14 every vertex with $\delta = k + 1$ appeared. By Lemma 16 the neighbors of vertices with $\delta = k + 1$ have $\delta \leq k + 2$. Since the white neighbors haven't been in the queue, these may not have $\delta \leq k + 1$ otherwise, by Lemma 14 and induction assumption would be put to the queue beforehand. □

BFS Traverse Correctness and Complexity

Theorem 18

BFSTraverse returns correct distances and shortest paths in $O(|V| + |E|)$.

Proof: Notice that d changes for every vertex 2 times, first in the initialization, second, in the assignment in line **6**, so the correctness of d is proven in Lemma 17.2. The correctness of the path is proven by induction: assume that we proved it for $\delta \leq k$. By Lemma 17, vertices $u \in V$ with $\delta(r, u) = k + 1$ are assigned with their δ values when discovered as a white neighbor of $v \in V$ with $\delta(r, v) = k$. Hence, the shortest path to v is augmented with the edge $v - u$, hence, has length $\delta(r, u)$, therefore, is the shortest one. Complexity: the while cycle has $|V|$ steps as every vertex is added one time and all step operations except of line **5-6** take $O(1)$, so this yields $O(|V|)$. The steps **5-6** consider every edge twice, and this yields $O(|E|)$. □

Shortest paths on a weighted graph

• Now we switch to a more general problem of shortest paths on graphs weighted with $w : E \to \mathbb{R}$ and we recall the assumption that the case 1 of Lemma 11 holds: no negative cycles exist.

• Similar to *BFSTraverse* we use the per vertex arrays, $d, prev$ such that at every moment $d[v] \geq \delta(r, v)$ holds and $prev[v]$ is the previous vertex in the so far shortest path from r. Similar to *BFSTraverse*, for the sake of proof, we also support color of the vertex, $col[d]$, first white, then black.

• Arrays $d, prev, col$ are initialized by *Initialize*:

Initilalize(G,r): for $v \in V$
$d[v] := \infty$;
$prev[v] := 0$;
$col[v] := white$;
$d[r] := 0$;

• Then both d and $prev$ are updated by *Relax* procedure:

Relax(v,u): if $(d[u] > d[v] + w(v, u))$
$d[u] := d[v] + w(v, u)$; $prev[u] := v$;
return 1
return 0

Of course, *Relax* is based on the triangle inequality, Lemma 12.

Dijkstra Algorithm

Now assume that w is non-negative: $w : E \to \mathbb{R}_{\geq 0}$. An obvious consequence of positivity is that $\delta(r, v)$ **does not decrease on a shorest path**.

Dijkstra Algorithm: Input: weighted graph $G(V, E, w)$ and root vertex $r \in V$. Output: d and $prev$. Throughout the algorithm we use a Min-Heap as a priority queue.

1. $Initialize(G, r)$;
2. $Q = BuildHeap(d)$;
3. while $(Q.IsEmpty = 0)$
4. $\quad v := Q.ExtractMin$;
5. $\quad col[v] := black$;
6. $\quad$ for $(u \in A[v])$
7. $\quad\quad$ if $(Relax(v, u))$
8. $\quad\quad\quad Q.DecreaseKey(u, d[u])$;
9. return $d, prev$;

Remark: As we will see in Theorem 19 below, the vertex u in **8** is white, so it hasn't been extracted from the heap, hence, *DecreaseKey* makes sense. Hence, we might decrease the number of calls of **7** if checking $col[u] = white$ in line **6**.

Dijkstra Algorithm correctness

Theorem 19

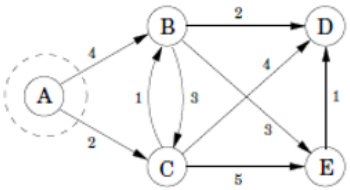
1. *Vertices become black in the ascending order of d.*
2. *When a vertex becomes black, holds $d[v] = \delta(r, v)$.*
3. *Vertex u in line* **8** *is always white.*

Proof: Let $v_0 = r, v_1, \ldots, v_k, \ldots$ be the order, in which the vertices get black and let $d_0[u], u \in V$ be the value of $d[u]$ in the very moment of u getting black; in fact, we are going to prove $d_0[u] = \delta(r, u)$. **1.** Assume the converse, then for some $k > 1$ holds $d_0[v_k] < d_0[v_{k-1}]$. However, in the moment, when v_{k-1} gets black an inequality $d[v_k] \geq d[v_{k-1}]$ holds, because v_{k-1} was extracted as the minimum. Then $Relax(v_{k-1}, u)$ was called for v_{k-1}'s neighbors and either v_k hasn't been changed or $d[v_k] := d_0[v_{k-1}] + w(v_{k-1}, v_k) \geq d_0[v_{k-1}]$, in contradiction to the assumed inequality $d[v_k] < d_0[v_{k-1}]$ immediately after that. **2.** Assume the converse for some k: $d_0[v_k] > \delta(r, v_k)$. Then, for some $l > k$ the call $Relax(v_l, v_k)$ would change $d[v_k]$ from $d_0[v_k]$ to a smaller value. However, this would contradict to $d_0[v_l] \geq d_0[v_k]$ proved in **1.** **3** is equivalent to **2**. □

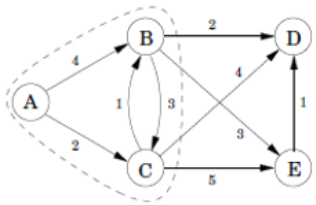
Dijkstra Algorithm complexity and example

Theorem 20

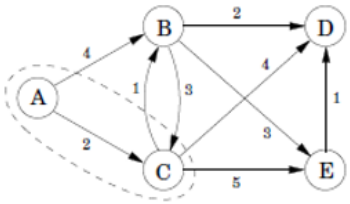
Dijkstra algorithm has complexity $O((|V| + |E|)\log(|V|))$.



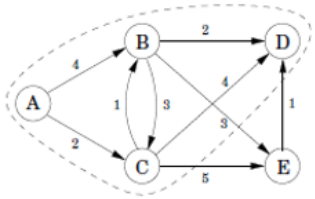
A: 0	D: ∞
B: 4	E: ∞
C: 2	



A: 0	D: 5
B: 3	E: 6
C: 2	



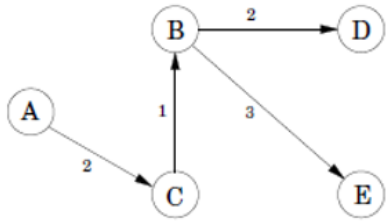
A: 0	D: 6
B: 3	E: 7
C: 2	



A: 0	D: 5
B: 3	E: 6
C: 2	

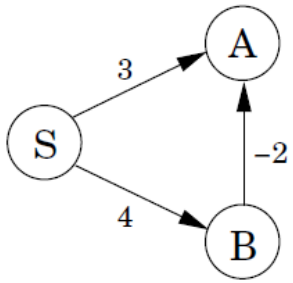
The steps of Dijkstra algorithm with A as root vertice and values of d.

Shortest path tree



Negative edges matter

• We mentioned that the positivity of weights implies that $\delta(r, v)$ grows monotonously along shortest paths, implying the nice property Theorem 19.1
• If having negative edges, we lose this nice property and Dijkstra does not need to work anymore. Consider the example:



• The shortest paths are: $S - B$ of length 4 and $S - B - A$ of length 2.
• If applying Dijkstra to such a graph, the first vertex that gets black color is A with $d[A] = 3$ at that step and this is not the correct $\delta(S, A) = 2$.
• Notice that if we want to deal with a normal situation such that a shortest path exists from every $u \in V$ to every reachable $v \in V$, then by Lemma 11 cyclic paths with negative length should be excluded.
• In particular, the graph may **not be undirected**! Indeed, an undirected graph with a negative edge $v \leftrightarrow u$ has a negative cycle $v \to u \to v$.

Bellman-Ford Algorithm

Bellman-Ford: Input: a weighted directed graph $G(V, E, w)$, a route vertex, r. Output: if negative cycle, return 0. Otherwise, return 1 and the arrays $d, prev$.

1. *Initialize*(G, r);
2. for $(t := 1 : |V| - 1)$
3. for $(u \to v \in E)$
4. *Relax*(u, v);
//at this step d and $prev$ are ready to be output, if no negative cycles
5. for $(u \to v \in E)$
6. if $(Relax(u, v))$
7. return 0 //negative cycle
8. return 1,$d, prev$

- Bellman-Ford algorithm correctness and complexity are due to the following

Theorem 21

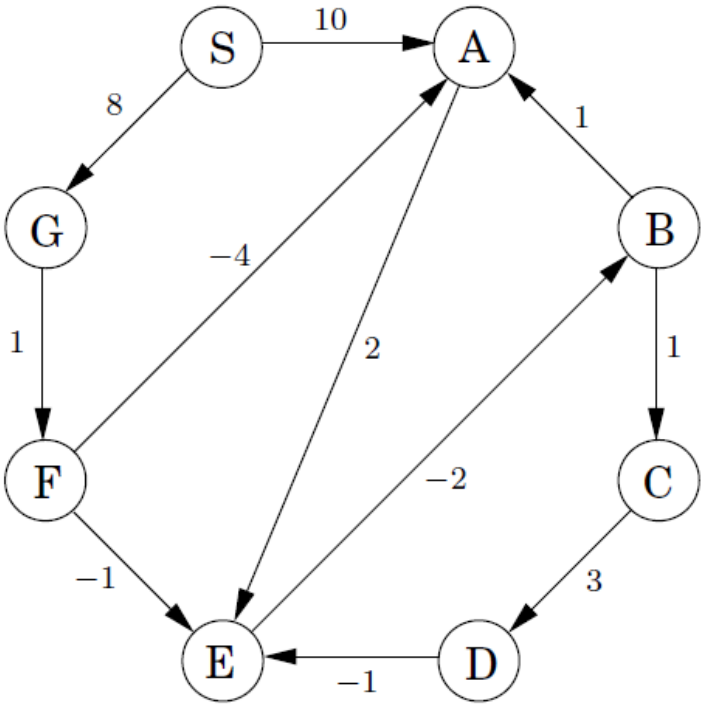
1. *If no negative cycles, then after* $|V| - 1$ *t-iterations, for every* $v \in V$ *reachable from* r *holds* $d[v] = \delta(r, v)$ *and a shortest path from* r *to* v *is in the tree.*
2. *A negative cycle exists iff Relax at line* **6** *makes a change for at least one edge.*
3. *Bellman-Ford is in* $\Theta(|V||E|)$.

Bellman-Ford Algorithm Correctness

Proof: 1. Let's call a step of passing line **4** *final* if after this step $d[v]$ didn't decrease anymore. A vertex v may be involved in several final steps $u \to v$, out of which the first one also finalised the parent $prev[v]$ in the path tree. **Claim:** every reachable vertex v is Relaxed by a final step while in first $|V| - 1$ t-iterations. Indeed, let $r = v_0, v_1, \ldots, v_k = v$ be a shortest path without (zero length) cycles guaranteed by Lemma 11. Notice that $k \leq |V| - 1$ because the path can't pass across any vertex twice. Then for $t = 1$ the step $Relax(v_0, v_1)$ is final. For $t = 2$ the step $Relax(v_1, v_2)$ is final, etc. Therefore, after k t-iterations $Relax(v_{k-1}, v_k)$ will be final. Applying induction, one may prove there exists a path from r to v in the tree, and it is a shortest one, by the Claim.
2. Due to **1**, if at least one *Relax* changes $d[v]$ after $|V| - 1$ loops, it means there are negative loops. Conversely, if there is a negative loop, then there is no a shortest path for every $u, v \in V$ such that $u \to v$ occurs in that negative cycle. Then some *Relax* while $t \geq |V|$ must return 1 and change d. However, if this didn't happen with any edge while $t = |V|$, this would not happen in the further step either. The proof of **3** is straighforward. □

Remark: Bellman-Ford is not a BFS.

Bellman-Ford Algorithm Example



	Iteration							
Node	0	1	2	3	4	5	6	7
S	0	0	0	0	0	0	0	0
A	∞	10	10	5	5	5	5	5
B	∞	∞	∞	10	6	5	5	5
C	∞	∞	∞	∞	11	7	6	6
D	∞	∞	∞	∞	∞	14	10	9
E	∞	∞	12	8	7	7	7	7
F	∞	∞	9	9	9	9	9	9
G	∞	8	8	8	8	8	8	8

Bellman-Ford Algorithm: an application example with $r = S$ and intermediate values of d for every vertex after each step of t iteration, line **2**.

Conclusion

• Graphs are a natural framework for many interesting algorithmic problems.
• Graph presentation may differ: adjacency list is optimal for sparse graphs, which are typical. But adjacency matrices have interesting applications to path search.
• Drawing a graph such that the picture is well understandable is a separate math problem that includes such brilliant ideas as Tutte's Spring Theorem.
• In this lecture we mostly concentated on Bread First Search as a concept and gave several examples of math problems solved by BFS.
• In particular we gave a broad picture of Shortest Path problem that is the origin of BFS. We saw how to solve it for non-weighted graphs (BFSTraverse), for positively weighted graphs (Dijkstra), and for directed graphs without negative cycles (Bellman-Ford).
• The domain of applications for BFS as a concept is much wider than the given well-known example, and it includes almost all, if not all Graph problems.
• The point that we'd like to emphasize is that BFS as a concept is a universal tool that waits for creative minds able to adjust it to the problem you deal with.
• To some extent, this makes BFS in Graph algorithms similar to Neural Networks in Machine Learning.

References


Kuratowski, Kazimierz, *Sur le problème des courbes gauches en topologie*, *Fund. Math.* **15**, 271–283


Tutte, W. T., *How to draw a graph*, *Proceedings of the London Mathematical Society*, Third Series, **13**, 743–767.


Williamson, S. G., *Depth-first search and Kuratowski subgraphs*, *J. ACM*, **31** (4), 681–693.