Week 11: Bellman-Ford & MSTs

Algorithms & Data Structures

Thorben Klabunde

th-kl.ch

December 1, 2025

Agenda

Mini-Quiz

Assignment

Feedback Assignment 9

Common points:

- **Ex. 9.3:**
  - **a) Read carefully!** Many proved that a general topological order exists but not the one specified in the task.
  - **b)** Be very **precise regading the specific property** you want to show, here: the indices form a strictly increasing sequence.
  - **c) Don't forget the justification** (both for correctness and runtime)! And justification means explaining *why* **not** just *what* or *how*. In the **exam** this will lead to **deductions**.

- **Ex. 9.4:** Note that a **closed walk** $\neq$ **cycle**. If you want to apply the argument that there can be no odd cycles by Thm. 1, we need to **decompose the closed Eulerian walk** into a disjoint set of cycles (which is possible, as shown in the lecture). This is an **important step** since Thm. 1 does not state anything regarding closed walks.

Remember to check the detailed feedback on Moodle! Reach out if you have any questions regarding the corrections.

Exercise 10.3 *Driving on highways.*

In order to encourage the use of train for long-distance traveling, the Swiss government has decided to make all the m highways between the n major cities of Switzerland one-way only. In other words, for any two of these major cities C_1 and C_2, if there is a highway connecting them it is either from C_1 to C_2 or from C_2 to C_1, but not both. The government claims that it is however still possible to drive from any major city to any other major city using highways only, despite these one-way restrictions.

(a) Model the problem as a graph problem. Describe the set of vertices V and the set of edges E in words. Reformulate the problem description as a graph problem on the resulting graph.

(b) Describe an algorithm that checks the correctness of the government's claim in time $O(n + m)$. Argue why your algorithm is correct and why it satisfies the runtime bound.

Exercise 10.5 *Strongly connected vertices* **(1 point)**.

Let $G = (V, E)$ be a directed graph with n vertices and m edges. We say two distinct vertices $v, w \in V$ are *strongly connected* if there exists both a directed path from v to w, and from w to v.

Describe an algorithm which finds a pair $v, w \in V$ of strongly connected vertices in G, or decides that no such pair exists. The runtime of your algorithm should be at most $O(n + m)$. You are provided with the number of vertices n, and the adjacency list Adj of G.

Hint: *Use DFS as a subroutine.*

Recap: Dijkstra

Dijkstra's Algorithm: Quick Review

Last week: Dijkstra's algorithm for weighted graphs with non-negative edge costs

The Idea:

- Maintain set S of *finalized* vertices
- Greedily select closest unprocessed vertex
- Add to S and relax outgoing edges
- Use priority queue (min-heap) for efficiency

Key Invariant:

When v^* is extracted from heap:

$$d[v^*] = d(s, v^*)$$

(distance is **final**)

Runtime:

$$O((|V| + |E|) \cdot \log |V|)$$

Why it works:

- **Non-negative costs** ensure: any path leaving S only gets more expensive
- $\implies$ Closest vertex in heap has final distance
- $\implies$ Greedy choice is safe!

Dijkstra's Algorithm: Implementation (Java Style)

```
1:  function DIJKSTRA(G, s)
2:      dist[·] ← ∞;    dist[s] ← 0
3:      PQ.add( new Node(s, 0) )
4:
5:      while PQ is not empty do
6:          (u, d) ← PQ.poll()
7:
8:          // 1. Check for stale entry
9:          if d > dist[u] then
10:             continue
11:         end if
12:
13:         // 2. Relax Edges
14:         for edge (u, v) with weight w do
15:             if dist[u] + w < dist[v] then
16:                 dist[v] ← dist[u] + w
17:                 PQ.add( new Node(v, dist[v]) )
18:             end if
19:         end for
20:     end while
21: end function
```

The "Lazy" Logic

Why duplicates? If we find a shorter path to v, we add a new pair (v, d_{new}) to the PQ.
The old pair (v, d_{old}) remains in the PQ but sinks to the bottom.

The "Stale" Check (Line 7): When we eventually pop (v, d_{old}), we see that:

$$d_{old} > \texttt{dist}[v]$$

So we ignore it.

Bellman-Ford

The Challenge: How to Handle Negative Edges?

Key observation about Dijkstra:

Dijkstra's Strong Guarantee

When vertex v is extracted from heap, we know $d[v] = d(s, v)$ is **final**.

We never need to reconsider v again!

With negative edges, we can't make this strong guarantee.

Weaker Guarantee: ℓ-good bounds

Instead of finalizing vertices one-by-one, we'll track:
"For which path lengths have we found optimal distances?"

After ℓ iterations: $d[v]$ is correct for all paths with $\leq \ell$ edges.

New strategy: Incrementally increase path length guarantees until we've covered all possible paths.

Understanding ℓ-Good Bounds

Definition: ℓ-good bound

For $\ell \in \mathbb{N}_0$, define:

$$S_{\leq \ell} := \{v \in V \mid \exists \text{ path from } s \text{ to } v \text{ with } \leq \ell \text{ edges}\}$$

A distance estimate $d[v]$ is ℓ-**good** if:

$$d[v] = \begin{cases} d(s,v) & \text{if } v \in S_{\leq \ell} \\ \geq d(s,v) & \text{otherwise} \end{cases}$$

i.e., $d[v]$ is exact for paths using $\leq \ell$ edges, and an upper bound otherwise

Base case ($\ell = 0$): $S_{\leq 0} = \{s\}$, $d[s] = 0$, $d[v] = \infty$ for $v \neq s$

Idea: Build up from 0-good to 1-good to 2-good, ...

When can we stop?

Improving Bounds: The Relaxation Step

How to go from ℓ-good to $(\ell + 1)$-good?

Recurrence Relation

For any vertex v, the shortest path using $\leq \ell + 1$ edges either:

1. Uses $\leq \ell$ edges (already covered), or
2. Uses exactly $\ell + 1$ edges: comes from some u via edge (u, v)

Therefore:

$$d^{(\ell+1)}(s, v) = \min\left\{d^{(\ell)}(s, v), \min_{u \to v}\{d^{(\ell)}(s, u) + c(u, v)\}\right\}$$

Implementation: *Relax all edges*
For each edge $(u, v) \in E$:

$$d[v] \leftarrow \min\{d[v], d[u] + c(u, v)\}$$

Key property: Relaxing all edges once improves from ℓ-good to $(\ell + 1)$-good!

How Many Edges in a Shortest Path?

Key Observation

In a graph with $n = |V|$ vertices (without negative cycles):
Any shortest simple path has at most $n - 1$ edges.

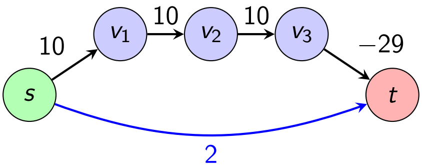
Why?

1. A simple path visits each vertex at most once
2. With n vertices, at most $n - 1$ edges connect them
3. If a path has $\geq n$ edges, it must revisit a vertex $\implies$ contains a cycle
4. Without negative cycles, we can remove the cycle to get a shorter path

Conclusion: If $d[v]$ is $(n-1)$-good for all v, then $d[v] = d(s, v)$ is optimal!
We just need to ensure our bounds are $(n-1)$-good.

Bellman-Ford: Visual Example

Example showing why we need $n-1$ iterations:



Two paths:

- Blue: cost 2 (1 edge)
- Black: cost $10+10+10-29=1$ (4 edges)

Iterations ($n=5$):

ℓ	s	v_1	v_2	v_3	t
0	0	∞	∞	∞	∞
1	0	10	∞	∞	2
2	0	10	20	∞	2
3	0	10	20	30	2
4	0	10	20	30	**1**

Note:

- After $\ell=1$: Find path with 1 edge (cost 2)
- After $\ell=2,3$: Build longer path
- After $\ell=4$: Discover optimal path (4 edges, cost 1)!

$\Longrightarrow$ Only at iteration $n-1$ do we guarantee optimality!

Bellman-Ford: Pseudocode & Runtime

function BELLMAN-FORD$(G = (V, E, c), s \in V)$
    $d[s] \leftarrow 0$, $d[v] \leftarrow \infty$ for all $v \neq s$

    for $\ell = 1$ to $|V| - 1$ **do** ▷ Build ℓ-good bounds
        for each edge $(u, v) \in E$ **do**
            if $d[u] + c(u, v) < d[v]$ **then**
                $d[v] \leftarrow d[u] + c(u, v)$

    return d

Runtime Analysis:

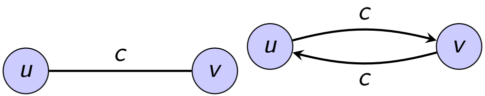
- Outer loop: $|V| - 1$ iterations
- Inner loop: $|E|$ edge relaxations
- **Total:** $O(|V| \cdot |E|)$

Handling Undirected Graphs

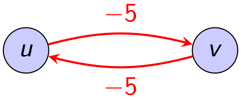
How do we handle undirected graphs?

Replace each undirected edge $\{u, v\}$ with **two directed edges**:

- (u, v) with cost $c(u, v)$
- (v, u) with cost $c(v, u) = c(u, v)$



Problem with negative edges:



This creates a **negative cycle**!

- $u \to v \to u$ has cost $-5 + (-5) = -10$
- Can traverse infinitely, cost $\to -\infty$
- Shortest paths undefined!

Critical limitation: Bellman-Ford (and any shortest path algorithm) **cannot** be used on undirected graphs with negative edge weights!

A single negative edge creates a negative cycle.

Detecting Negative Cycles

Can we detect if a negative cycle exists?

Algorithm Extension

After running $n - 1$ iterations of Bellman-Ford:

1. Run **one more** iteration (the n-th iteration)
2. Check if **any** distance updates occur
3. If yes $\implies$ negative cycle exists
4. If no $\implies$ no negative cycle reachable from s

Your Turn: Proving Negative Cycles

The Claim: If an update occurs in iteration $|V|$, there must be a negative cycle.

Guided Proof (Fill in the logic):

1. **Path Length:** Bellman-Ford guarantees that after k iterations, we know the shortest paths using at most k edges.
   Therefore: After n iterations:

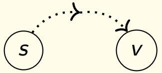
2. **Vertex Count:** A path with n edges touches _______ vertices.

3. **The Conflict:** The graph only has n vertices. By the **Pigeonhole Principle**, what must happen?

4. **The Cost:** If the cycle had positive weight, would this longer path be shorter than the simple path?

Time: 5 Minutes

Discuss with your neighbor!



Path with n edges?

Minimum Spanning Trees

Definitions: Trees and Spanning Trees

1. What is a Tree?

A **Tree** is a graph that is:

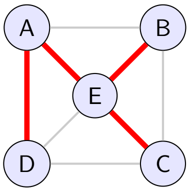
- **Connected** (1 component)
- **Acyclic** (No cycles)

Recall: A tree with V vertices always has exactly $|V| - 1$ **edges**.

2. What is a Spanning Tree?

Given a graph $G = (V, E)$, a **spanning tree** $T = (V, E')$ is a subgraph such that:

- T contains **all** vertices of G ("Spanning").
- T is a valid Tree.



Gray = Graph G

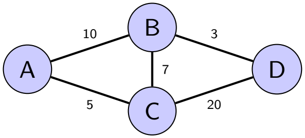
Red = Spanning Tree T

The Minimum Spanning Tree Problem

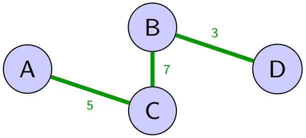
Problem Definition

Given: Undirected graph $G = (V, E)$ with edge weight function $c : E \to \mathbb{R}$
Find: A spanning tree T of minimum total weight:

$$w(T) = \sum_{e \in T} c(e)$$



Graph G



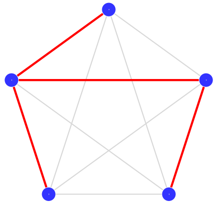
MST: $w = 15$

The Naive Approach: Brute Force?

Algorithm:

1. Enumerate **all** spanning trees of graph G.
2. Compute the total weight for each tree.
3. Return the tree with the minimum weight.

Input: K_5 (Complete Graph)



One possible tree (red)

The Bottleneck: Cayley's Formula

For a complete graph K_n, the number of spanning trees is:

$$T(n) = n^{n-2}$$

We need a better way...

The search space is too large. We need a way to build the tree **step-by-step** without looking back. $\rightarrow$ **Greedy Algorithms**.

Greedy Algorithms: When Do They Work?

Recall: A greedy algorithm makes locally optimal choices, hoping they lead to a globally optimal solution.

General Properties for Greedy Algorithms

1. Optimal Substructure
The optimal solution incorporates optimal solutions to subproblems.

2. Greedy Choice Property
A locally optimal choice leads to a globally optimal solution.

Examples that work:

- Dijkstra's algorithm
- **Minimum Spanning Trees!**

Examples that don't:

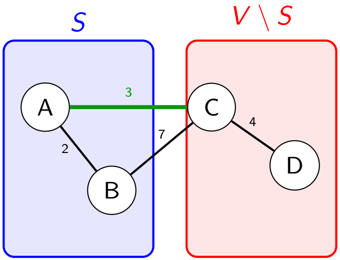
- Knapsack problem
- Longest path problem

Greedy Choice Property: The Cut Property

Key insight: We can always safely include certain edges in the MST.

Definition: Cut

A **cut** $(S, V \setminus S)$ is a partition of vertices into two non-empty sets.
An edge (u, v) **crosses the cut** if $u \in S$ and $v \in V \setminus S$ (or vice versa).



Green edge: minimum-weight crossing edge

Cut Property

For any cut $(S, V \setminus S)$, let e be a **minimum-weight edge** crossing the cut.

Then e belongs to **some** MST of G.

Intuition:

- Any spanning tree must cross the cut
- Might as well use the cheapest crossing edge!

Proof Strategy: The Exchange Argument

The Goal: Prove the Greedy Choice g is part of *some* optimal solution.

The Logic Flow:

1. **Assume** there exists an Optimal Solution (OPT) that *does not* contain our greedy choice g.
2. **Identify** an element x in OPT that "conflicts" with g.
3. **Swap** them: Build a new solution OPT'.

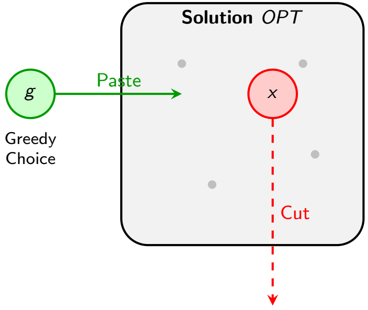
$$OPT' = (OPT \setminus \{x\}) \cup \{g\}$$

4. **Compare Costs:** Since g was the greedy choice, it is "better" (or equal) to x:

$$\text{cost}(g) \leq \text{cost}(x)$$

Therefore:

$$\text{cost}(OPT') \leq \text{cost}(OPT)$$



Conclusion:
OPT' is valid and costs no more than OPT. Thus, a solution with g is optimal.

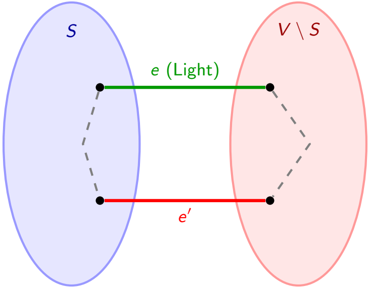
Proof: The Cut Property

Lemma (Cut Property)

For any cut $(S, V \setminus S)$, any minimum-weight edge $e = \{u, v\}$ crossing the cut (with $u \in S, v \notin S$) is in some MST.

Exchange Argument:

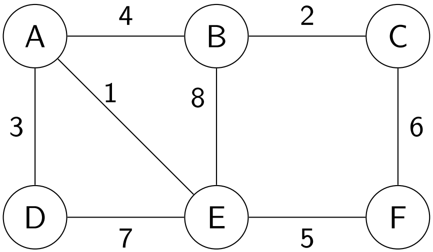
1. Assume MST T **does not** contain e.
2. **The Cycle:** Adding e to T creates a cycle.
3. **The Crossing:** This cycle must cross the cut at least twice.
   - Once at e (by definition).
   - Once at some other edge e' (already in T).
4. **The Swap:** Since e is the lightest crossing edge:
$$w(e) \leq w(e')$$
5. **Conclusion:** Create $T' = (T \setminus \{e'\}) \cup \{e\}$.
$$w(T') = w(T) - w(e') + w(e) \leq w(T)$$



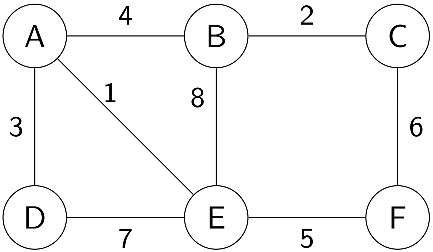
Result: Since e is a min.-weight edge crossing the cut, the tree can't have gotten worse!

The Cut Property gives us a greedy choice. But how do we use it?

1. Prim's Algorithm



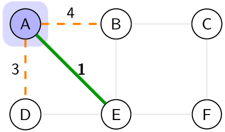
2. Boruvka's Algorithm



Prim's Algorithm: Iterative Expansion

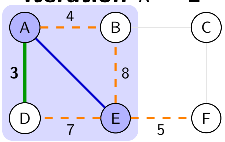
Formal Definition: Maintain a set $S \subset V$. At each step, select the edge (u, v) with minimum weight such that $u \in S$ and $v \notin S$.

Iteration $k = 1$



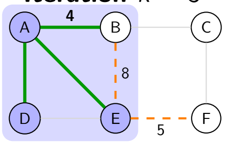
$$S = \{A\}$$
$$\text{Cut} = \{(A, B), (A, D), (\mathbf{A}, \mathbf{E})\}$$
$$\min = w(A, E) = 1$$

Iteration $k = 2$



$$S = \{A, E\}$$
$$\text{Cut} = \{(A, B), (\mathbf{A}, \mathbf{D}), (E, B) \dots\}$$
$$\min = w(A, D) = 3$$

Iteration $k = 3$



$$S = \{A, E, D\}$$
$$\text{Cut} = \{(\mathbf{A}, \mathbf{B}), (E, F), (D, F) \dots\}$$
$$\min = w(A, B) = 4$$

Invariant Maintenance

Claim: At every step, the set of edges selected T is a subset of some MST.

Proof: The edge $e = \text{argmin}_{(u,v) \in \text{Cut}(S)} w(u, v)$ is a safe edge by the **Cut Property** and doesn't add a cycle. Adding it preserves the MST invariant.

Prim's Algorithm: Implementation Complexity

Key Insight: Prim's is essentially **Dijkstra's Algorithm**, but measuring distance from the *tree* (S), not the *source* (s).

c

function $\textsc{Prim}(G, s)$
  // Init: $O(V)$
  $Q \leftarrow \text{PriorityQueue}(V)$
  $\texttt{key}[v] \leftarrow \infty, \forall v; \quad \texttt{key}[s] \leftarrow 0$

  while $Q \neq \emptyset$ **do** ▷ Loop $|V|$ times
  $u \leftarrow \texttt{ExtractMin}(Q)$
  for $v \in \text{Adj}[u]$ **do** ▷ Loop $|E|$ times total
    if $v \in Q$ **and** $w(u, v) < \texttt{key}[v]$ **then**
      $\texttt{key}[v] \leftarrow w(u, v)$
      $\texttt{parent}[v] \leftarrow u$
      $\texttt{DecreaseKey}(Q, v)$
    end if
  end for

  return $\{(v, \texttt{parent}[v])\}$
end function=0

Runtime (Binary Heap)

1. **Vertex Operations:**
   - We call `ExtractMin` once for every vertex.
   - Cost: $|V| \times O(\log |V|)$
2. **Edge Operations:**
   - We call `DecreaseKey` at most once for every edge.
   - Cost: $|E| \times O(\log |V|)$

Total Time:

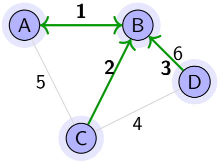
$$O((|V| + |E|) \log |V|)$$

Boruvka's Algorithm: The Parallel Approach

Why pick one edge at a time? Let's pick **one edge per component** simultaneously!

The Phase Strategy:

1. **Input:** Start with n components (each vertex is separate).
2. **Scan:** For *every* component C, find the cheapest edge leaving C.
3. **Add:** Add *all* these edges to the MST at once.
4. **Merge:** Contract connected components.
5. Repeat until only 1 component remains.



Phase 1: Everyone picks closest neighbor

Why it guarantees progress: Every component adds at least one edge. The number of components reduces by at least half (50%) in every phase!

Boruvka's Algorithm: Implementation Analysis

```
function BORŮVKA(G = (V, E, c))
    T ← ∅
    𝒞 ← {{v} | v ∈ V}                          ▷ Init: |V| components

    while |𝒞| > 1 do                           ▷ Phase Loop
        // 1. Scan Edges (O(|V| + |E|))
        for each component C ∈ 𝒞 do
            Find min-weight edge e_C leaving C
        end for

        // 2. Merge (O(|V| + |E|))
        T ← T ∪ {all found e_C}
        Merge components connected by T
        Update 𝒞
    end while
    return T
end function
```

Complexity Logic

1. Inner Work (Per Phase): We scan edges to find minimums and merge components.

$$O(|V| + |E|)$$

2. Number of Phases: Every phase reduces the number of components by at least half.

$$|C_{new}| \leq \frac{1}{2}|C_{old}|$$

Max phases: $O(\log(|V|))$

Total Runtime:

$$O((|V| + |E|)\log(|V|))$$

Summary: Shortest Path Algorithms

Algorithm	Input / Constraints	Technique	Runtime
BFS	Unweighted	Graph Traversal	$O(\|V\| + \|E\|)$
DAG-SP	Weighted, **No Cycles**	Dynamic Prog.	$O(\|V\| + \|E\|)$
Dijkstra	Weighted, **Non-negative**	**Greedy** (P. Queue)	$O((\|V\| + \|E\|) \log \|V\|)$
Bellman-Ford	General Weights	**Dynamic Prog.**	$O(\|V\| \cdot \|E\|)$

Which one should I use?

- **Default choice: Dijkstra** (fastest for standard maps).
- **If negative edges exist:** Must use **Bellman-Ford** (slower, but safe).
- **If DAG:** Exploit topological ordering and use DP!
- **Bellman-Ford Bonus:** Can **detect** negative cycles (returns False).

Summary: Minimum Spanning Trees (MST)

Core Principle: All MST algorithms use the **Cut Property** (Greedy). They only differ in *which* cut they pick next.

Algorithm	Growth Strategy	Data Structure	Runtime
Prim	**Serial:** Grow 1 tree (Vertex-centric)	Priority Queue	$O((\lvert V\rvert + \lvert E\rvert) \log \lvert V\rvert)$
Borůvka	**Parallel:** Grow all components at once	Adjacency Scan	$O((\lvert V\rvert + \lvert E\rvert) \log \lvert V\rvert)$

Questions?

Additional Practice

Exercise: Trip Planning in New York City

You are organizing a day trip to New York City with your friends to see the sights. However, your friends are quite particular about when they want to visit each attraction. The trip must follow these constraints:

- The day is divided into three time slots:
  - **Morning:** 9:00–12:00
  - **Noon:** 12:00–15:00
  - **Evening:** 15:00–18:00
- Each sight has specific opening hours given by a function $t : V \to \{1, \ldots, 24\} \times \{1, \ldots, 24\}$ that returns $(t_{\text{open}}, t_{\text{close}})$ for each sight. Sights are open in contiguous blocks of multiples of 3 hours (e.g., a sight might be open 9–12, or 9–15, or 12–18, but not 10–13).
- You must **change sights**, one during each time slot, based on when your friends want to see them and when the sights are open.
- Travel time between any two sights u and v is given by $c : V \times V \to \mathbb{N}$ (in minutes).
- You start and end at your hotel H. You can assume all sights are reachable within their time slots, and that you spend enough time at each sight to fill its entire time slot.

Task: Design an algorithm to find the route that minimizes total travel time. Your algorithm should run in $O((n+m)\log n)$ time, where n is the number of sights and $m \leq n^2$ is the number of edges in your graph.