Week 12: Kruskal & Union-Find

Algorithms & Data Structures

Thorben Klabunde

th-kl.ch

December 8, 2025

Agenda

1. Mini-Quiz
2. Assignment
3. Kruskal's Algorithm
4. Union-Find
5. Additional Practice

Mini-Quiz

Assignment

Feedback Assignment 10

Common points:

- **Ex. 10.1/2**: Well done, just be careful with the order, definitions and read the tasks carefully, making sure to answer every question.
- **Ex. 10.5**: Runtime and a correctness justification are required! In the exam, you will get deductions if you don't cover them (even if they are obvious). For these exercises, it can sometimes be easier to describe the algorithm in text to avoid mistakes in the pseudocode.

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

Ex. 11.1

Exercise 11.1 *Shortest paths with cheating* **(1 point)**.

Let $G = (V, E)$ be a weighted, directed graph with weights $c : E \to \mathbb{R}_{\geq 0}$. We consider a variation of the shortest path problem in G, where we are allowed to 'cheat' by setting a certain number of weights to 0. Formally, for $k \in \mathbb{N}$, we write C_k for the set of all weight functions $\gamma : E \to \mathbb{R}_{\geq 0}$ on G with $\gamma(e) \neq c(e)$ for at most k edges $e \in E$.[1]

Given $s, t \in V$, we wish to find a path $P = (v_1 = s, v_2, \ldots, v_\ell = t)$ in G which minimizes:

$$c_k(P) := \min_{\gamma \in C_k} \gamma(P), \text{ where } \gamma(P) := \sum_{i=1}^{\ell-1} \gamma\big((v_i, v_{i+1})\big).$$

We call such a path a 'shortest path from s to t with k cheats.'

Ex. 11.1

(b) Describe an algorithm which finds the length of a shortest path from s to t with k cheats in time $O\big((k|V|)^2\big)$.

Address the following elements:

1) The graph that you use. This includes: its vertex set and edge set; whether edges are directed or not; whether vertices or edges are weighted and if so, their weight.

Solution:

We set $V' = \{v^{(\ell)} : v \in V,\ \ell \in \{0, 1, 2, \ldots, k\}\}$, which has size $|V'| = (k+1) \cdot |V|$. We define E' and $c' : E' \to \mathbb{R}_{\geq 0}$ by:

$$\forall\, (v, w) \in E,\ 0 \leq \ell \leq k: \quad (v^{(\ell)}, w^{(\ell)}) \in E' \text{ with } c'((v^{(\ell)}, w^{(\ell)})) = c((v, w)), \tag{T1}$$

$$\forall\, (v, w) \in E,\ 0 \leq \ell \leq k-1: \quad (v^{(\ell)}, w^{(\ell+1)}) \in E' \text{ with } c'((v^{(\ell)}, w^{(\ell+1)})) = 0, \tag{T2}$$

$$\forall\, 0 \leq \ell \leq k-1: \quad (t^{(\ell)}, t^{(\ell+1)}) \in E' \text{ with } c'((t^{(\ell)}, t^{(\ell+1)})) = 0. \tag{T3}$$

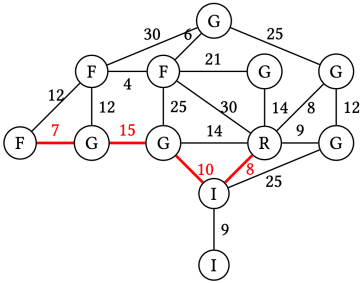
The first set of edges (T1) represents normal edge-traversal in G, the second set (T2) represents cheating, and the final set (T3) ensures that there is always a path from $s^{(0)}$ to $t^{(k)}$ in G' if there is a path from s to t in G.

Ex. 11.2

You are given a graph $G = (V, E)$ representing the towns of Switzerland. Each vertex V corresponds to a town, and there is an (undirected) edge $\{v_1, v_2\} \in E$ if and only if there exists a direct road going from town v_1 to town v_2. Additionally, there is a function $w : E \to \mathbb{N}$ such that $w(e)$ corresponds to the number of hours needed to hike over road e, and a function $\ell : V \to \{\mathrm{G}, \mathrm{F}, \mathrm{I}, \mathrm{R}\}$ that maps each town to the language that is spoken there[2]. For simplicity, we assume that only one language is spoken in each town.

Alice asks you to find an algorithm that returns the walking duration (in hours) of the shortest hike that goes through at least one town speaking each of the four languages.

For example, consider the following graph, where languages appear on vertices:



Ex. 11.2

(a) Suppose we know the order of languages encountered in the shortest hike. It first goes from an R vertex to an I vertex, then immediately to a G vertex, and reaches an F vertex in the end, after going through zero, one or more additional G vertices. In other terms, the form of the path is RIGF or RIG...GF. In this case, describe an algorithm which finds the shortest path satisfying the condition, and explain its runtime complexity. Your algorithm must have complexity at most $O((|V| + |E|) \log |V|)$.

Address the following elements:

1) The graph that you use. This includes: its vertex set and edge set; whether edges are directed or not; whether vertices or edges are weighted and if so, their weight.

$$
\begin{aligned}
E' = &\{\{v_s, (v,1)\} \mid \{u,v\} \in E, \ell(v) = \mathrm{R}\} \\
&\cup \{\{(u,1),(v,2)\} \mid \{u,v\} \in E, \ell(v) = \mathrm{I}\} \\
&\cup \{\{(u,2),(v,3)\} \mid \{u,v\} \in E, \ell(v) = \mathrm{G}\} \\
&\cup \{\{(u,3),(v,3)\} \mid \{u,v\} \in E, \ell(v) = \mathrm{G}\} \\
&\cup \{\{(u,3),(v,4)\} \mid \{u,v\} \in E, \ell(v) = \mathrm{F}\} \\
&\cup \{\{(v,4), v_d\} \mid v \in V\}
\end{aligned}
$$

$$
w'(\{u',v'\}) = \begin{cases} 0 & \text{if } u' = v_s \text{ or } v' = v_d \\ w(\{u,v\}) & \text{if } u' = (u,i) \text{ and } v' = (v,j) \end{cases}
$$

Ex. 11.2

(b) Now we don't make the assumption in (a). Describe an algorithm which finds the shortest path satisfying the condition. Briefly explain your approach and the resulting runtime complexity. Your algorithm must have complexity at most $O((|V| + |E|) \log |V|)$. Address the following elements:

1) The graph that you use. This includes: its vertex set and edge set; whether edges are directed or not; whether vertices or edges are weighted and if so, their weight.

Consider the vertex set V' above, as well as the following edge set E' and weight function w':

$$
\begin{aligned}
E' = \{\{v_s, (v, ((\ell(v) == \mathrm{G}), (\ell(v) == \mathrm{F}), (\ell(v) == \mathrm{I}), (\ell(v) == \mathrm{R})))\} \mid v \in V\} \\
\cup \{\{(v, (1,1,1,1)), v_d\} \mid v \in V\} \\
\cup \{\{(u, (g,f,i,r)), (v, (1,f,i,r))\} \mid (g,f,i,r) \in \{0,1\}^4, \{u,v\} \in E, \ell(v) = \mathrm{G}\} \\
\cup \{\{(u, (g,f,i,r)), (v, (g,1,i,r))\} \mid (g,f,i,r) \in \{0,1\}^4, \{u,v\} \in E, \ell(v) = \mathrm{F}\} \\
\cup \{\{(u, (g,f,i,r)), (v, (g,f,1,r))\} \mid (g,f,i,r) \in \{0,1\}^4, \{u,v\} \in E, \ell(v) = \mathrm{I}\} \\
\cup \{\{(u, (g,f,i,r)), (v, (g,f,i,1))\} \mid (g,f,i,r) \in \{0,1\}^4, \{u,v\} \in E, \ell(v) = \mathrm{R}\}
\end{aligned}
$$

$$
w'(\{u',v'\}) = \begin{cases} 0 & \text{if } u' = v_s \text{ or } v' = v_d \\ w(\{u,v\}) & \text{if } u' = (u,(g,f,i,r)) \text{ and } v' = (v,(g,f,i,r)) \end{cases}
$$

Ex. 11.4

Exercise 11.4 *Driving from Zurich to Geneva* **(1 point)**.

Bob is currently in Zurich and wants to visit his friend that lives in Geneva. He wants to travel there by car and wants to use only highways. His goal is to get to Geneva as cheap as possible. He has a map of the cities in Europe and which ones are connected by highways (in both directions). For each highway connecting two cities he knows how much fuel he will need for this part (depending on the length, condition of the road, speed limit, etc.) and how much this will cost him. This cost might be different depending on the direction in which he travels. Furthermore, for some connections between two cities, he has the option to take a passenger with him that will pay him a certain amount of money. Again this might be different depending on the direction he travels. We assume that this option is only available to him between cities directly connected by a highway and that the passengers want to travel the direct road and would not agree to making a detour. Also Bob has a small car, so he can only take at most one passenger with him. It is possible that he gains more money from this than he has to pay for the fuel between two given cities but we assume that he has no way to gain an infinite amount of money, i.e. there is no round-trip from any city that earns him money.

Ex. 11.4

(a) Model the problem as a graph problem.

Address the following elements:

1) The graph that you use. This includes: its vertex set and edge set; whether edges are directed or not; whether vertices or edges are weighted and if so, their weight.

Solution:

The graph $G = (V, E, w)$ is defined as follows: V is the set of cities on his map of Europe. There are directed edges between the cities (in both directions) if they are connected by a highway. The weight of any edge is the difference of the cost he needs to pay for the fuel for this highway and the money that a passenger would pay him (if available, otherwise it is just the cost he needs pay).

2) The algorithm that you apply to this graph. You can use the algorithms covered in the lecture material as subroutines, and you can use their running time bounds without proof.

Solution:

We apply Bellman-Ford starting at v_{Zurich} corresponding to Zurich since edge weights are possibly negative.

Kruskal's Algorithm

Recap: MST Algorithms So Far

What we've seen:

Prim's Algorithm:

- Grow a single tree from a starting vertex
- Always add the cheapest edge leaving the current tree
- Uses: Priority Queue
- Runtime: $O((|V| + |E|) \log |V|)$

Boruvka's Algorithm:

- Grow all components simultaneously
- Each component adds its cheapest outgoing edge
- Uses: Component tracking
- Runtime: $O((|V| + |E|) \log |V|)$

Different approach: What if we process edges in a **global** order and only pick edges that are safe?

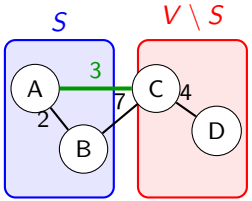
Kruskal's Algorithm: The Idea

New Approach: Instead of growing trees, look at min-weight edges and ensure that no cycles are introduced!

Kruskal's Greedy Strategy

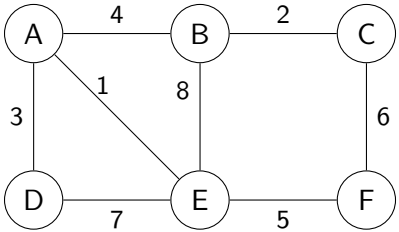
1. Sort all edges by weight (cheapest first)
2. Go through edges in order
3. For each edge (u, v):
   - If adding it doesn't create a cycle $\rightarrow$ **add it to MST**
   - Otherwise $\rightarrow$ **skip it**
4. Stop when we have $|V| - 1$ edges

- **Recall: Minimum Cut Property** states that any minimum weight edge that crosses a cut must be in some MST.



Kruskal: Manual Walkthrough

Min-Cut Property gives us a greedy choice. How does Kruskal use it to select the edges?



Sorted Edges:

Edge	Weight
{A, E}	1
{B, C}	2
{D, A}	3
{A, B}	4
{E, F}	5
{C, F}	6
{D, E}	7
{B, E}	8

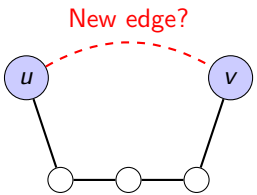
The Challenge: Detecting Cycles Efficiently

Bottleneck: How do we check if adding an edge creates a cycle?

Naive Approach (BFS / DFS):

- Traverse the graph starting from u.
- **Worst Case:** Must explore the entire connected component to find v (or prove it's unreachable).
- Cost per edge: $O(|V|)$
- **Total Cost:**
  $O(|E| \cdot |V|)$

  Too Slow for dense graphs!



Need to search entire component!

Notice: We don't need full path information!
We only need to know if u and v are in the same connected component.
This can be answered as a **Union-Find** problem much more efficiently!

Union-Find

Union-Find Data Structure

Goal: Maintain a partition of elements.

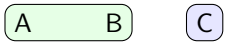
Required Operations

1. **make(x):** Create a new set containing only x
2. **find(x):** Return the *representative* of the set containing x
3. **union(x, y):** Merge the sets containing x and y

For Kruskal's:

- Each vertex starts in its own set
- `find(`u`) == find(`v`)` $\implies$ same component
- If different: `union(`u`, `v`)` to merge them

Initial: 3 separate sets



After union(A, B): 2 sets

Array-Based Representation

Idea: Use an array `rep[]` where each element stores its component representative.

Structure:

- Array `rep[1..n]`
- `rep[v]` = representative of v's component
- Elements with same `rep` are in same component

Operations:

- **find(v):** Return `rep[v]`
- **union(u, v):** Change all reps matching one to match the other

Example:

	A	B	C	D	E
`rep[]`:	A	B	C	D	E

Initially: each element is its own representative

	A	B	C	D	E
`rep[]`:	A	C	C	A	A

After `union(A, D)` and `union(D, E)`

Invariant: find(u) = find(v) $\iff$ u and v are in the same component

Naive Union-Find: Operation Costs

Implementation Idea: Array `rep[]` where `rep[v]` stores component's rep of v.

Individual Operation Complexities

- **find(v):** Return `rep[v]` $\Theta(1)$
- **union(u, v):** Scan entire array, change all entries matching `rep[u]` to `rep[v]` $\Theta(n)$

Why is union $\Theta(n)$?

- Must scan all n positions in the array
- Check if `rep[i]` matches the old component rep
- Update matching entries to new component rep
- Cannot avoid scanning the entire array — we don't know which entries need updating!

Bottleneck: Every union operation costs $\Theta(n)$ time $\implies$ $\Theta(n^2)$ in total.

Union: Improvement 1

Idea: Maintain a list of members for each component.

Additional Data Structure

members[r]: List of all vertices in the component with representative r

For example: If `rep[A]` $=$ `rep[B]` $=$ `rep[C]` $=$ A, then:

$$\texttt{members[A]} = [A, B, C]$$

Invariant:

`members[rep[u]]` is a list of all nodes in the connected component of u

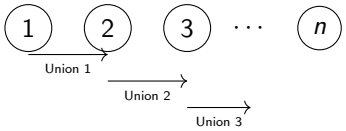
Runtime: $O(|\texttt{members}[r_u]|) = O(|\text{component of } u|)$
No longer need to scan all n positions!

Members List: Does This Solve Our Problem?

Question: Is $O(|\text{component of } u|)$ per union good enough?

Answer: Not quite... Consider the worst case:

Worst-case sequence:



Build a chain: union(1,2), union(2,3), ..., union($n-1$,n)

Cost Analysis:

i-th union call:
After $i-1$ unions, we have a component of size i.
Union cost: $O(i)$

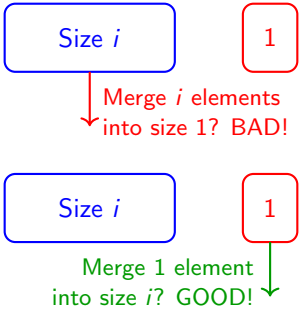
Total cost:

$$
\begin{aligned}
& O(1+2+3+\cdots+(n-1)) \\
&= O\left(\frac{n(n-1)}{2}\right) \\
&= O(n^2)
\end{aligned}
$$

Problem: Still $O(n^2)$ total cost in the worst case...

Improvement 2: Weighted Union

Why did the worst case happen?
We kept merging the *larger* component into the *smaller* one!



Weighted Union Rule: Always merge the **smaller** component into the **larger** one.
This ensures each element is moved at most $\log n$ times!

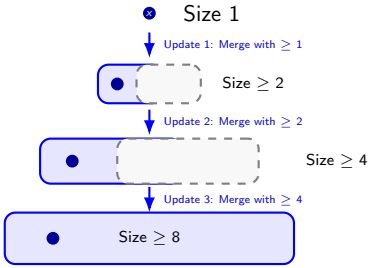
Weighted Union: Runtime

Observe: How many times can a single element's representative change?

Update Chain:

- Consider a specific element x.
- x's representative is only updated if x's set is the **smaller** one in a union.
- Therefore, after the union, the new set size is at least **double** the old size.

Visualized



Component size: After k-th update: size $\geq 2^k$

Since max size is N:

$$2^k \leq n \implies k \leq \log_2 n$$

Conclusion: Total complexity is $O(n \log n)$ because each of the n elements is moved at most $\log n$ times.

Union: Implementation

```
1  void union(ArrayList<LinkedList> mems,
2             int[] rep, int u, int v) {
3
4      // 1. OPTIMIZATION: Ensure u is smaller
5      if (mems.get(u).size() > mems.get(v).size()) {
6          int swap = u; u = v; v = swap;
7      }
8
9      // 2. MERGE: Update smaller (u) into larger (v)
10     LinkedList<Integer> uList = mems.get(u);
11     LinkedList<Integer> vList = mems.get(v);
12
13     // Update representative for u itself
14     rep[u] = v;
15     vList.add(u);
16
17     // Update representatives for all members of u
18     for (Integer z : uList) {
19         rep[z] = v;      // The "Work"
20         vList.add(z);    // Move member
21     }
22 }
```

Note: Lines 6 and 18 guarantee we only touch the minority of elements.

Complexity Analysis

1. Single Union Cost

- Worst case: $O(n)$
- *(Happens if we merge two sets of size $n/2$)*

2. Aggregate Cost (Kruskal's)

- How many times does a *specific element's* `rep` change?
- Every time it changes, its set size **doubles**.
- Max set size is n.
- Therefore: $\leq \log_2 n$ updates per element.

Total Time: $O(n \log n)$

Kruskal: Pseudocode

```
1:  function KRUSKAL(G = (V, E, c))
2:      T ← ∅                                              ▷ MST edges
3:
4:      // Sort edges by weight
5:      Sort E by increasing weight: e_1, e_2, ..., e_|E|
6:
7:      // Process edges in order
8:      for each edge (u, v) ∈ E (in sorted order) do
9:          if find(u) ≠ find(v) then
10:             T ← T ∪ {(u, v)}                           ▷ Add to MST
11:             union(u, v)                                ▷ Merge components
12:         end if
13:         if |T| = |V| − 1 then
14:             break                                      ▷ MST complete
15:         end if
16:     end for
17:     return T
18: end function
```

Your Turn: Proving Kruskal's Correctness

Claim: Let A be the set of edges selected by Kruskal's. Then $A \subseteq T$ for some MST T.

Proof. Proceed by induction on $|A|$:

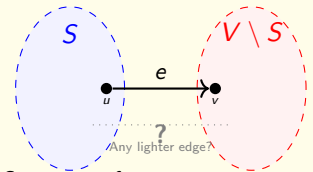
B.C.: $|A| =$ _____

I.H.: Assume $A \subseteq T$ after selecting _____ edges.

Inductive Step: Let $e = (u, v)$ be the next edge Kruskal adds and let S be the set of vertices in u's conn.-comp.

1. **Cut:** What can we say about e? *Does v belong to S?*
2. **Minimality:** We know Kruskal adds edges in _____ order. (*Could there be an edge e' crossing $(S, V \setminus S)$ with $w(e') < w(e)$?*)
3. **Conclusion:** By the **Cut Property**:

Visualize the Step



Kruskal: Runtime

Kruskal's Complexity

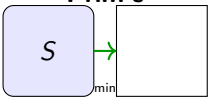
Sorting edges:	$O(m \log m) = O(m \log n)$
Union-Find:	$O(m + n \log n)$
Total:	$O(m \log n + m + n \log n)$
	$= O((m + n) \log n)$

Note: Since $m \leq n^2$, we have $\log m \leq 2 \log n$

Summary: Complete MST Picture

Three Greedy Algorithms, One Principle: The Cut Property

Prim's

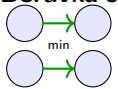


Grow one tree
Add min edge
leaving tree

Kruskal's

Sort all edges
Add min edge
not creating cycle

Boruvka's



All components
add min edge
simultaneously

Key Takeaways:

- All three algorithms are correct by the Cut Property
- Runtime: $O(|V| + (|E|) \log |V|)$ for all three
- Kruskal's requires Union-Find for efficiency

Additional Practice

Theory Task T4. / 12 P

1. Consider the following problem. The Swiss government is negotiating a deal with Elon Musk to build a tunnel system between all major Swiss cities. They put their faith into you and consult you. They present you with a map of Switzerland. For each pair of cities it depicts the cost of building a bidirectional tunnel between them. The Swiss government asks you to determine the cheapest possible tunnel system such that every city is reachable from every other city using the tunnel network (possibly by a tour that visits other cities on the way).

   i) Model the problem as a graph problem. Describe the set of vertices, the set of edges and the weights in words. What is the corresponding graph problem?

ii) Use an algorithm from the lecture to solve the graph problem. State the name of the algorithm and its running time in terms of $|V|$ and $|E|$ in Θ-notation.

2. Now, the Swiss tunneling society contacts the government and proposes to build the tunnel between Basel and Geneva for half of Musk's cost. Thus, the government contacts you again. They want you to solve the following problem: Given the solution of the old problem in a) and an edge for which the cost is divided by two, design an algorithm that updates the solution such that the new edge cost is taken into account. *In order to achieve full points, your algorithm must run in time* $\mathcal{O}(|V|)$.

   Hint: You are only allowed to use the *solution* from 1., i.e. the set of tunnels in the chosen tunnel system. You are not allowed to use any intermediate computation results from your algorithm in 1.

   i) Describe your algorithm (for example, via pseudocode). A high-level description is enough.

ii) Prove the correctness of your algorithm and show that it runs in time $\mathcal{O}(|V|)$.

Your Turn: MST "Bottlenecks"

Define the *bottleneck capacity* of a path as the maximum weight of any edge on that path.

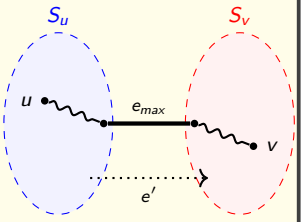
Claim: The path between any two nodes u, v in an MST minimizes the bottleneck capacity.
Guided Proof:

1. Let P_{MST} be the path in the MST between u and v. Let e_{max} be the heaviest edge in P_{MST}. Removing e_{max} creates a Cut (S_u, S_v) because ...

   What do we know about $w(e_{max})$ compared to other edges crossing this cut?

2. Assume for contradiction there is another path P_{alt} in G whose bottleneck is **strictly less** than $w(e_{max})$. Then:

Time: 5-8 Minutes



Does P_{alt} have to cross the gap?