COMP 382: Reasoning about Algorithms

Review Session

Prof. Maryam Aliakbarpour

co-instructors: Prof. Anjum Chida & Prof. Konstantinos Mamouras

December 4, 2025

Today's Lecture

Greedy Algorithms

What is a Greedy Algorithm?

- Solves an **optimization problem**:
  - Maximize or minimize an objective
  - Subject to constraints
- Builds the solution **step by step**
- At each step:
  - Choose what looks **best locally**
  - Once a choice is made, it is not changed later (**irrevocable**).

Key question: *Do locally optimal choices lead to a global optimum?*

Proof of Correctness: General Recipe

The proof of correctness for a greedy algorithm has two main steps:

- The produced solution is **valid** given the constraints of the problem.
- The final solution is **optimal**.

Proof of Correctness: General Recipe

The proof of correctness for a greedy algorithm has two main steps:

- The produced solution is **valid** given the constraints of the problem. Usually easy.
- The final solution is **optimal**. This is the main hurdle.

Proof of Correctness: General Recipe

The proof of correctness for a greedy algorithm has two main steps:

- The produced solution is **valid** given the constraints of the problem. Usually easy.
- The final solution is **optimal**. This is the main hurdle.

Common Techniques for Proving Optimality: The core idea is a **comparison with an optimal solution**.

- **Greedy Stays Ahead:** Show that the greedy choice is always "better" than or equal to the optimal choice at every step, leading to an equally good final result.
- **Exchange Argument:** Show that any differences between a supposed optimal solution and the greedy solution can be "exchanged" to make the optimal solution more like the greedy one, without making it worse.

Proving Optimality

To prove our "Farthest-First" strategy is optimal, we'll use a classic technique: **Greedy Stays Ahead**.

1. **Setup:** Let's define two refueling strategies:
   - $G = \{g_1, g_2, \ldots, g_k\}$: The sequence of stops chosen by our **Greedy** algorithm.
   - $O = \{o_1, o_2, \ldots, o_m\}$: The sequence of stops in any **Optimal** solution.

Proving Optimality

To prove our "Farthest-First" strategy is optimal, we'll use a classic technique: **Greedy Stays Ahead**.

1. **Setup:** Let's define two refueling strategies:
   - $G = \{g_1, g_2, \ldots, g_k\}$: The sequence of stops chosen by our **Greedy** algorithm.
   - $O = \{o_1, o_2, \ldots, o_m\}$: The sequence of stops in any **Optimal** solution.

2. **Goal:** We want to prove that our greedy solution makes the minimum number of stops, i.e., $k = m$.

Proving Optimality

To prove our "Farthest-First" strategy is optimal, we'll use a classic technique: **Greedy Stays Ahead**.

1. **Setup:** Let's define two refueling strategies:
   - $G = \{g_1, g_2, \ldots, g_k\}$: The sequence of stops chosen by our **Greedy** algorithm.
   - $O = \{o_1, o_2, \ldots, o_m\}$: The sequence of stops in any **Optimal** solution.

2. **Goal:** We want to prove that our greedy solution makes the minimum number of stops, i.e., $k = m$.

3. **Method: Greedy Stays Ahead.** We will show that the greedy choice is always at least as close to the destination as the optimal choice at every step.

The Exchange Argument: A General Technique

This is a powerful method for proving a greedy algorithm is correct. The core idea is to show that any other solution can be transformed into the greedy one without increasing the cost.

1. Start with an assumed optimal solution, σ^*, that is different from the greedy one, σ.

The Exchange Argument: A General Technique

This is a powerful method for proving a greedy algorithm is correct. The core idea is to show that any other solution can be transformed into the greedy one without increasing the cost.

1. Start with an assumed optimal solution, σ^*, that is different from the greedy one, σ.

2. Find a place where σ^* differs from the greedy choice. This is an "inversion".

The Exchange Argument: A General Technique

This is a powerful method for proving a greedy algorithm is correct. The core idea is to show that any other solution can be transformed into the greedy one without increasing the cost.

1. Start with an assumed optimal solution, σ^*, that is different from the greedy one, σ.
2. Find a place where σ^* differs from the greedy choice. This is an "inversion".
3. Perform a small, local **exchange** to make σ^* look more like σ.

The Exchange Argument: A General Technique

This is a powerful method for proving a greedy algorithm is correct. The core idea is to show that any other solution can be transformed into the greedy one without increasing the cost.

1. Start with an assumed optimal solution, σ^*, that is different from the greedy one, σ.
2. Find a place where σ^* differs from the greedy choice. This is an "inversion".
3. Perform a small, local **exchange** to make σ^* look more like σ.
4. Show that this exchange does not increase the cost (and often improves it).

The Exchange Argument: A General Technique

This is a powerful method for proving a greedy algorithm is correct. The core idea is to show that any other solution can be transformed into the greedy one without increasing the cost.

1. Start with an assumed optimal solution, σ^*, that is different from the greedy one, σ.
2. Find a place where σ^* differs from the greedy choice. This is an "inversion".
3. Perform a small, local **exchange** to make σ^* look more like σ.
4. Show that this exchange does not increase the cost (and often improves it).
5. Argue that the greedy solution is at least as good as any optimal solution.

Huffman's Greedy Algorithm

The core insight: The two symbols with the smallest frequencies must be siblings at the lowest level of the optimal tree.

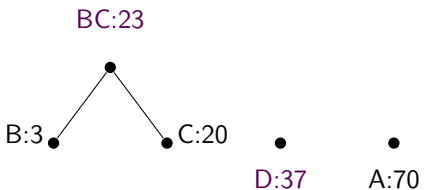
The Greedy Strategy

1. Identify the two symbols with the lowest frequencies.
2. Join them as children of a new parent node. This parent's frequency is the sum of its children's frequencies ($f_i + f_j$).
3. Remove the original two symbols from the list and add this new parent node.
4. Repeat this process until only one node remains—the root of the tree.

Huffman Algorithm: An Example

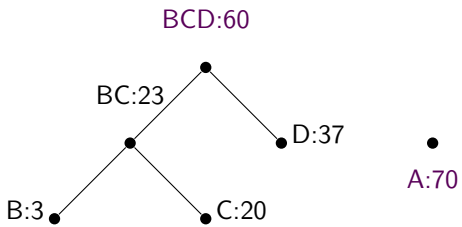
Symbol	Frequency
A	70 million
B	3 million
C	20 million
D	37 million

Huffman Algorithm: An Example



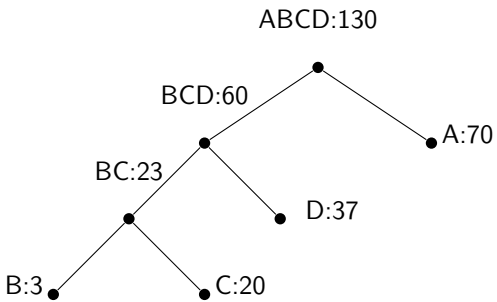
Symbol	Frequency
A	70 million
B	3 million
C	20 million
D	37 million

Huffman Algorithm: An Example



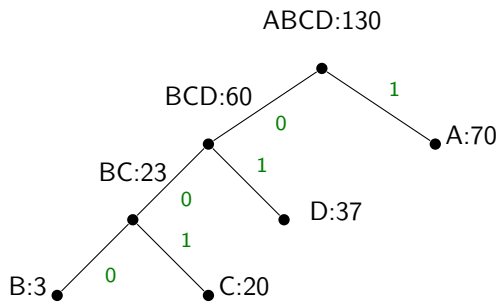
Symbol	Frequency
A	70 million
B	3 million
C	20 million
D	37 million

Huffman Algorithm: An Example



Symbol	Frequency
A	70 million
B	3 million
C	20 million
D	37 million

Huffman Algorithm: An Example

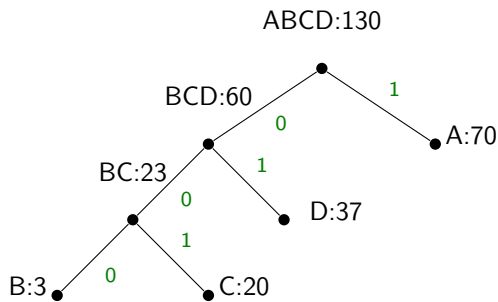


Symbol	Frequency
A	70 million
B	3 million
C	20 million
D	37 million

The Huffman algorithm gives us the following code:

A→1 B→000 C→001 D→01

Huffman Algorithm: An Example



Symbol	Frequency
A	70 million
B	3 million
C	20 million
D	37 million

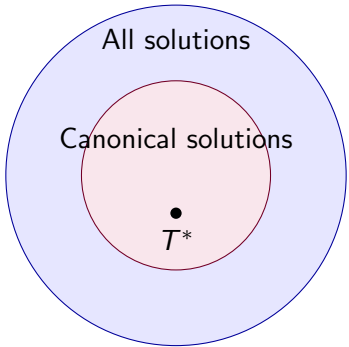
The Huffman algorithm gives us the following code:

A→1 B→000 C→001 D→01

Total length: 213 million bits ↓ %18 reduction in the size

Canonical Solutions

If a globally optimal solution lies within a restricted set, then we may focus solely on that restricted set and still obtain a globally optimal solution.
We choose this restricted set carefully so that it simplifies the proof.



Possible Questions

- **Huffman Coding Concepts**

Possible Questions

- **Huffman Coding Concepts**
  - Understanding its implementation

Possible Questions

- **Huffman Coding Concepts**
  - Understanding its implementation
  - Understanding the relationship between symbol frequency and codeword depth/length (similar to Question 1.1 from Quiz 3).

Possible Questions

- **Huffman Coding Concepts**
  - Understanding its implementation
  - Understanding the relationship between symbol frequency and codeword depth/length (similar to Question 1.1 from Quiz 3).

- Conceptual questions about **canonical format**: what a canonical format means and how optimization changes under this restriction.

Possible Questions

- **Huffman Coding Concepts**
  - Understanding its implementation
  - Understanding the relationship between symbol frequency and codeword depth/length (similar to Question 1.1 from Quiz 3).

- Conceptual questions about **canonical format**: what a canonical format means and how optimization changes under this restriction.

- **A New Greedy-Algorithm Problem**

Possible Questions

- **Huffman Coding Concepts**
  - Understanding its implementation
  - Understanding the relationship between symbol frequency and codeword depth/length (similar to Question 1.1 from Quiz 3).

- Conceptual questions about **canonical format**: what a canonical format means and how optimization changes under this restriction.

- **A New Greedy-Algorithm Problem**
  - Identify what constitutes a single **step** (or choice) in the problem.

Possible Questions

- **Huffman Coding Concepts**
  - Understanding its implementation
  - Understanding the relationship between symbol frequency and codeword depth/length (similar to Question 1.1 from Quiz 3).

- Conceptual questions about **canonical format**: what a canonical format means and how optimization changes under this restriction.

- **A New Greedy-Algorithm Problem**
  - Identify what constitutes a single **step** (or choice) in the problem.
  - Propose the natural **greedy strategy**.

Possible Questions

- **Huffman Coding Concepts**
  - Understanding its implementation
  - Understanding the relationship between symbol frequency and codeword depth/length (similar to Question 1.1 from Quiz 3).

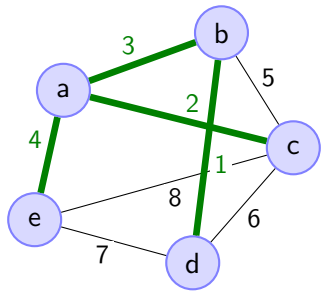
- Conceptual questions about **canonical format**: what a canonical format means and how optimization changes under this restriction.

- **A New Greedy-Algorithm Problem**
  - Identify what constitutes a single **step** (or choice) in the problem.
  - Propose the natural **greedy strategy**.
  - Compare the greedy choice with the optimal one using an **exchange argument** or **greedy stays ahead** proof.

Minimum Spanning Trees

Definition

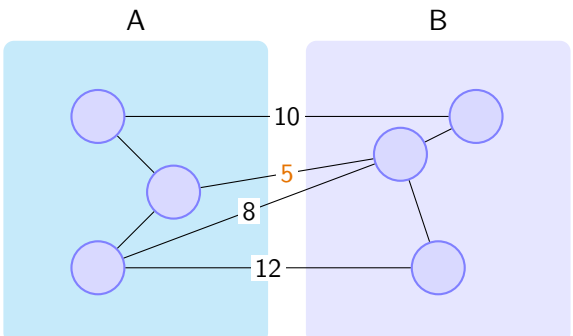
Problem. Given connected $G = (V, E)$ with weights w, find a spanning tree T minimizing $\sum_{e \in T} w(e)$.



The Cut Property (safe-to-include)

The Cut Property

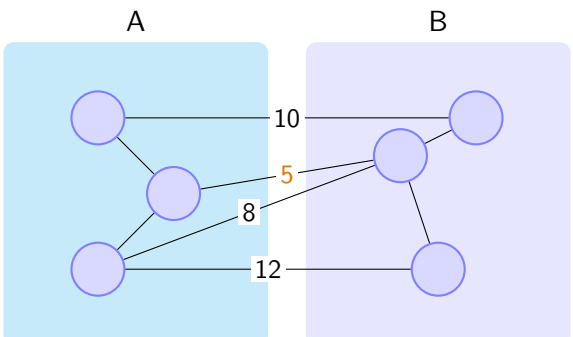
Assume all edge costs are distinct.
Let e be the **cheapest edge** crossing *any* cut (A, B).
Then e **must** belong to the Minimum Spanning Tree.



The Cut Property

Why is this true? If an MST **didn't** use e, it would have to use some other, more expensive edge f to cross that cut. We could swap f for e and get a **cheaper** tree!

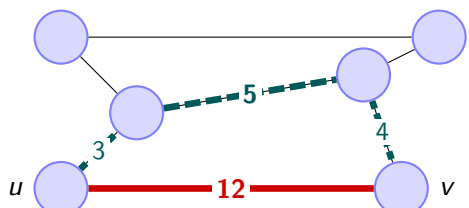
This is a contradiction.



The Cycle Property (safe-to-exclude)

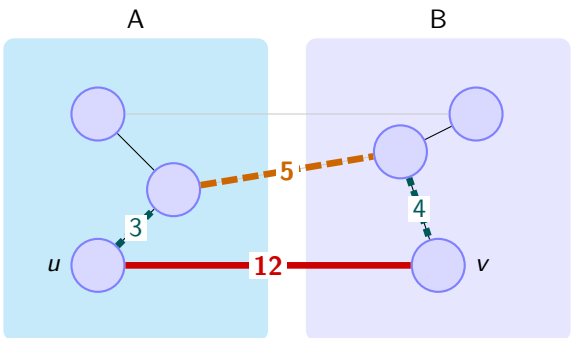
The Cycle Property

On any cycle C, the heaviest edge is **unsafe**. There is an MST that does not use this edge. If the heaviest edge is unique, it appears in no MST.



Cycle Property: Exchange Argument

Key Idea. In any cycle, the **heaviest** edge f cannot belong to an MST. Remove f: the graph splits into two components with endpoints u and v. The rest of the cycle is a $u - v$ path, so it must **cross the cut**. That crossing edge e in the cycle satisfies $w(e) < w(f)$. Swapping f for e yields a strictly lighter spanning tree.



Exchange Argument with Two Trees

Setup. Let T be an MST and let S be any spanning tree. Pick an edge $e \in S \setminus T$. Add e to T; this creates a unique cycle C in $T \cup \{e\}$.

Exchange Argument with Two Trees

Setup. Let T be an MST and let S be any spanning tree. Pick an edge $e \in S \setminus T$. Add e to T; this creates a unique cycle C in $T \cup \{e\}$.

Removing e from S disconnects S into two components. The endpoints of e are connected in T by a unique path, and that path must cross this cut. Hence, along that T–path there is an edge that can reconnect the two components of $S \setminus \{e\}$; call this edge f.

Exchange Argument with Two Trees

Setup. Let T be an MST and let S be any spanning tree. Pick an edge $e \in S \setminus T$. Add e to T; this creates a unique cycle C in $T \cup \{e\}$.

Removing e from S disconnects S into two components. The endpoints of e are connected in T by a unique path, and that path must cross this cut. Hence, along that T–path there is an edge that can reconnect the two components of $S \setminus \{e\}$; call this edge f.

This gives rise to two new spanning trees:

$$T' = T + \{e\} - \{f\}, \qquad S' = S - \{e\} + \{f\}.$$

Comparing the weights of T' and S' to those of T and S leads to the key inequalities in many MST arguments.

Exchange Argument with Two Trees

Setup. Let T be an MST and let S be any spanning tree. Pick an edge $e \in S \setminus T$. Add e to T; this creates a unique cycle C in $T \cup \{e\}$.

Removing e from S disconnects S into two components. The endpoints of e are connected in T by a unique path, and that path must cross this cut. Hence, along that T–path there is an edge that can reconnect the two components of $S \setminus \{e\}$; call this edge f.

This gives rise to two new spanning trees:

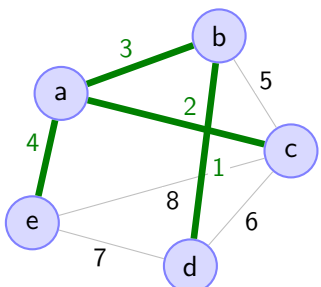
$$T' = T + \{e\} - \{f\}, \qquad S' = S - \{e\} + \{f\}.$$

Comparing the weights of T' and S' to those of T and S leads to the key inequalities in many MST arguments.

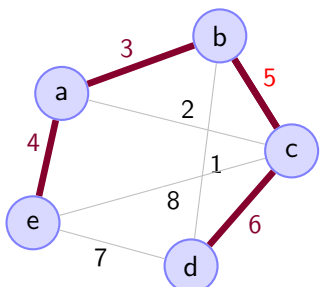
See the Second Best MST in Lab 5, and Question 3 in quiz 3.

Exchange argument with two trees

Tree T

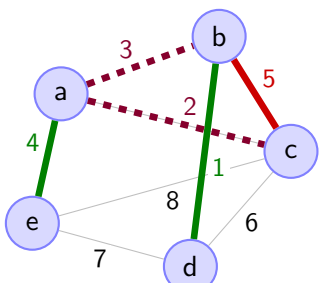


Tree S

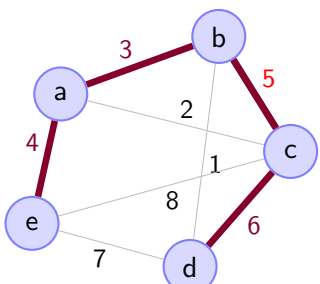


Exchange argument with two trees

$T \cup \{(b, c)\}$

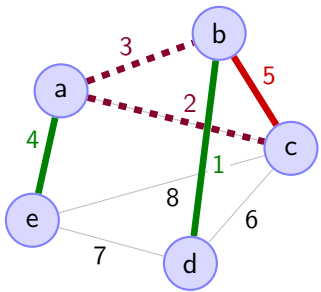


Tree S

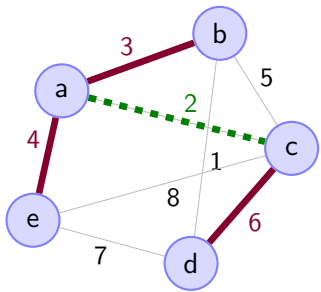


Exchange argument with two trees

$T \cup \{(b, c)\}$

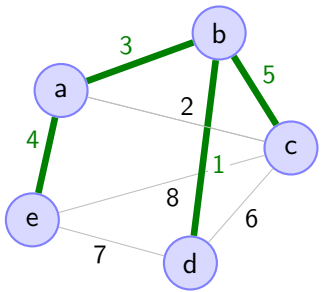


$S \setminus \{(b, c)\}$

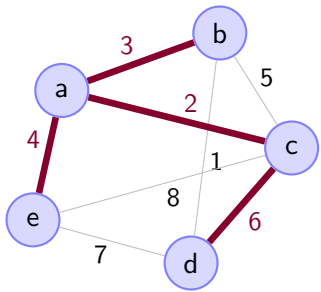


Exchange argument with two trees

Tree T'



Tree S'



Uniqueness of the MST

Theorem (Uniqueness Condition). A connected, weighted graph has a **unique** MST if and only if

every cut of the graph has a unique lightest edge crossing it.

Intuition.

- Ties among lightest cut edges give multiple safe choices $\rightarrow$ multiple MSTs.
- Ties among heaviest cycle edges give multiple edges that can be safely removed $\rightarrow$ multiple MSTs.

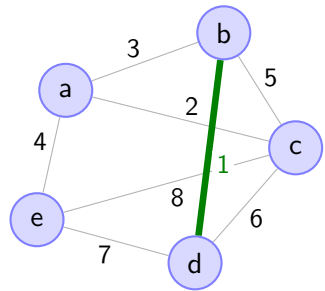
Kruskal's algorithm

Outline.

1. Sort edges by weight.
2. Scan in order; add if it connects two components.

Correctness: each added edge is lightest across some cut (Cut Property).

Data structure: Union-Find.



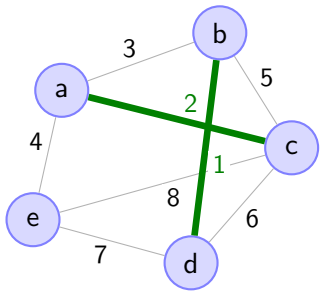
Kruskal's algorithm

Outline.

1. Sort edges by weight.
2. Scan in order; add if it connects two components.

Correctness: each added edge is lightest across some cut (Cut Property).

Data structure: Union-Find.



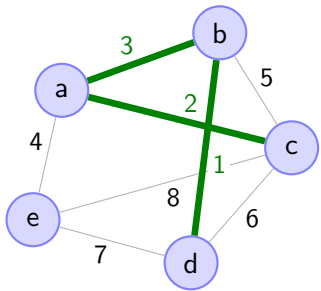
Kruskal's algorithm

Outline.

1. Sort edges by weight.
2. Scan in order; add if it connects two components.

Correctness: each added edge is lightest across some cut (Cut Property).

Data structure: Union-Find.



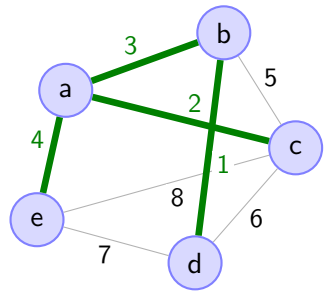
Kruskal's algorithm

Outline.

1. Sort edges by weight.
2. Scan in order; add if it connects two components.

Correctness: each added edge is lightest across some cut (Cut Property).

Data structure: Union-Find.



Possible Question

Conceptual focus: Determining whether an edge belongs to the MST, using the cut and cycle properties.

A partially built MST is given, and the question asks for the next step of the algorithm. (Comparable to Question 3 on Quiz 4.)

- Kruskal's algorithm
  - including the use of the union–find data structure
- Prim's algorithm
- Borůvka's algorithm

Max Flow and Min Cut

The Maximum Flow Problem

The goal: What is the maximum amount of "stuff" we can send from the source node **s** to the sink node **t**?

The Maximum Flow Problem

The goal: What is the maximum amount of "stuff" we can send from the source node **s** to the sink node **t**?

The maximum flow problem is to find a valid flow f that maximizes this value.

$$\max_{\text{valid } f} |f| \ .$$

Here, we define the **value** or the **size of a flow**, denoted by $|f|$, as the total net flow leaving the source node s, or going to the sink node t.

$$|f| := \underbrace{\sum_{u \in OUT(s)} f(s,u)}_{\text{outgoing flow from } s} - \underbrace{\sum_{u \in IN(s)} f(u,s)}_{\text{incoming flow to } s}$$

The Maximum Flow Problem

The goal: What is the maximum amount of "stuff" we can send from the source node **s** to the sink node **t**?

The maximum flow problem is to find a valid flow f that maximizes this value.

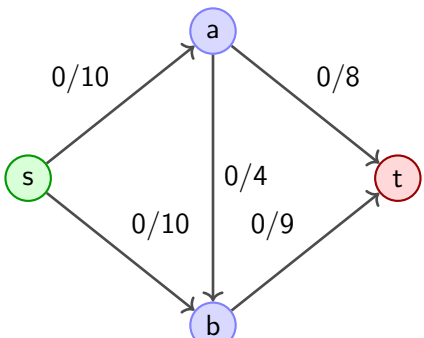
$$\max_{\text{valid } f} |f| \ .$$

Here, we define the **value** or the **size of a flow**, denoted by $|f|$, as the total net flow leaving the source node s, or going to the sink node t.

$$|f| := \underbrace{\sum_{u \in OUT(s)} f(s,u)}_{\text{outgoing flow from } s} - \underbrace{\sum_{u \in IN(s)} f(u,s)}_{\text{incoming flow to } s} = \underbrace{\sum_{u \in IN(t)} f(u,t)}_{\text{incoming flow to } t} - \underbrace{\sum_{u \in OUT(t)} f(t,u)}_{\text{outgoing flow from } t}$$

Ford-Fulkerson: An Example

Let's find the max flow for this network. Total flow so far: **0**.

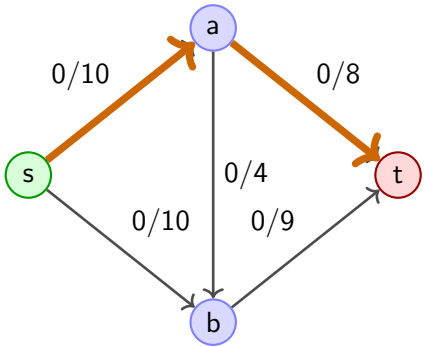


Example: Augmenting Path 1

Path: $s \to a \to t$.

Bottleneck: $\min(10, 8) = \mathbf{8}$.

Augmenting Path



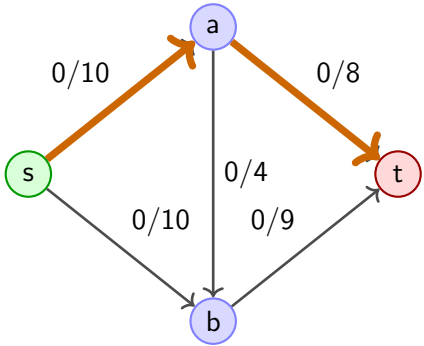
We push 8 units of flow.

Example: Augmenting Path 1

Path: $s \to a \to t$.

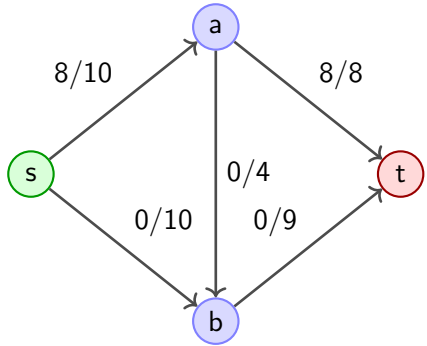
Bottleneck: $\min(10, 8) = \mathbf{8}$.

Augmenting Path



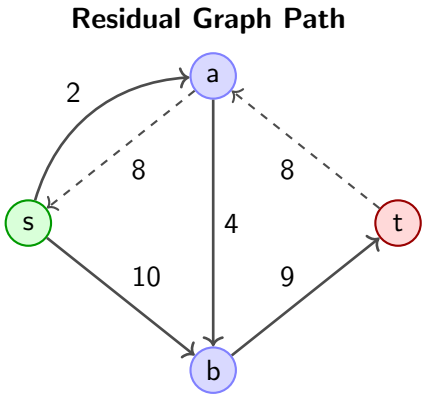
We push 8 units of flow.

Updated Flow



New total flow: **8**.

Example: Augmenting Path 2

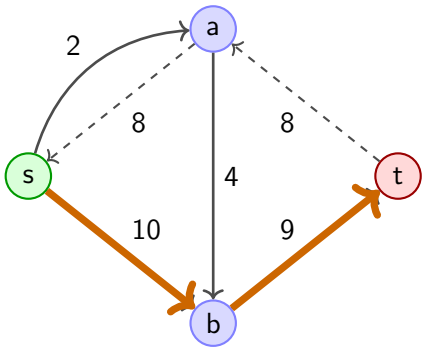


Example: Augmenting Path 2

Path: $s \to b \to t$.

Bottleneck: $\min(10, 9) = \mathbf{9}$.

Residual Graph Path



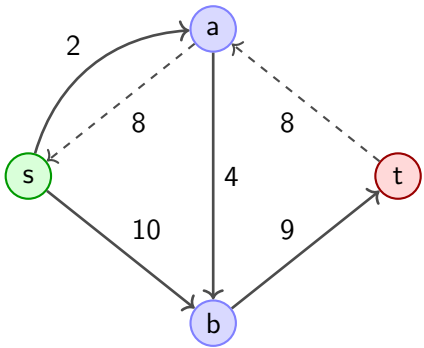
We push 9 units of flow.

Example: Augmenting Path 2

Path: $s \to b \to t$.

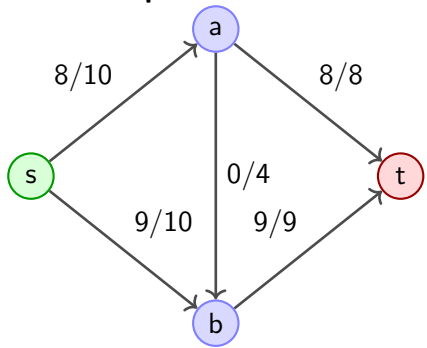
Bottleneck: $\min(10, 9) = \mathbf{9}$.

Residual Graph Path



We push 9 units of flow.

Updated Flow

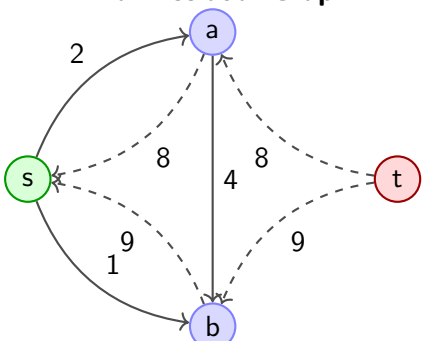


New total flow: $8 + 9 = \mathbf{17}$.

Example: Searching for another Path

After pushing 17 units of flow, let's search for another path in the residual graph.

Final Residual Graph



In the residual graph, there is no incoming edge to t.

No more augmenting paths can be found.

Example: No More Paths Found

Since there is no path from **s** to **t** in the final residual graph, the algorithm terminates.

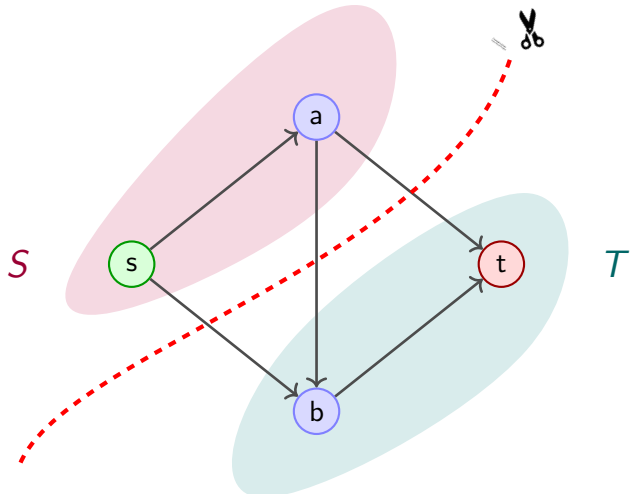
The total flow is the sum of the flows sent along the augmenting paths found:

$$\textbf{Maximum Flow} = 17$$

This result is guaranteed to be the maximum by the max-flow min-cut theorem.

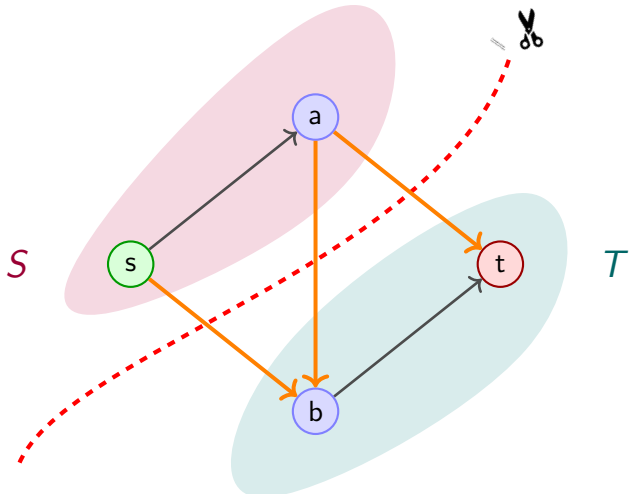
What Is an s-t Cut?

- An *s-t cut* is a **partition** of the vertices into two sets, S and T, such that the source $s \in S$ and the sink $t \in T$.



What Is an s-t Cut?

- An *s-t cut* is a **partition** of the vertices into two sets, S and T, such that the source $s \in S$ and the sink $t \in T$.
- The **capacity** (or the **size**) of the cut is the sum of capacities of all edges going from a node in S to a node in T.

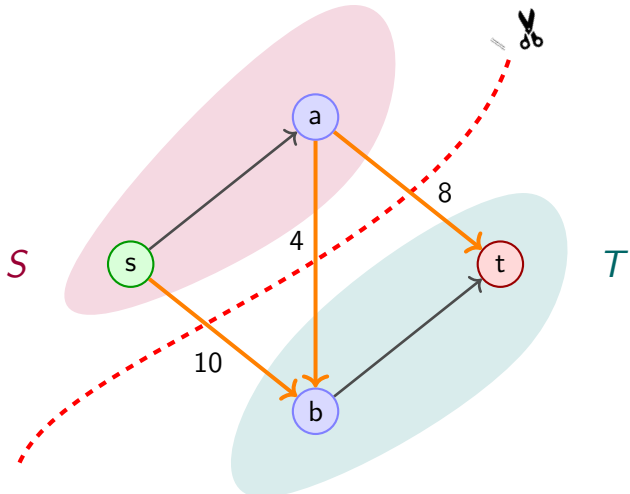


What Is an s-t Cut?

- An *s-t cut* is a **partition** of the vertices into two sets, S and T, such that the source $s \in S$ and the sink $t \in T$.
- The **capacity** (or the **size**) of the cut is the sum of capacities of all edges going from a node in S to a node in T.

The capacity is the sum of edges crossing from S to T (orange):

$$10 + 8 + 4 = 22.$$



The Max-Flow Min-Cut Theorem

Theorem

In any flow network, the value of the maximum (s, t)-flow is equal to the capacity of the minimum (s, t)-cut.

$$\max_{\text{flows } f} |f| = \min_{\text{cuts } (S,T)} \text{capacity}(S, T)$$

The Max-Flow Min-Cut Theorem

Theorem

In any flow network, the value of the maximum (s, t)-flow is equal to the capacity of the minimum (s, t)-cut.

$$\max_{\text{flows } f} |f| = \min_{\text{cuts } (S,T)} \text{capacity}(S, T)$$

Main Lemma: Weak Duality

Lemma (Weak Duality Lemma)

For any feasible (s,t)-flow f and any (s,t)-cut (S,T), the value of the flow is at most the capacity of the cut.

$$|f| \leq \textit{capacity}(S,T)$$

Important implication

$$\text{Weak duality} \quad \Rightarrow \quad \text{max flow} \leq \text{min cut}$$

Weak duality provides **certificates of optimality** when a feasible solutions are equal: they both must be optimal.

Potential Questions

Conceptual questions that probe understanding of how max-flow works and why.

- **Residual Graph:** What it represents, why we need reverse edges, and how residual capacity enables undoing previous choices.
- **Augmenting Paths:** Why pushing flow along any augmenting path increases the total flow, and how augmenting on a reverse edge "cancels" earlier flow.
- **Cuts and Min-Cut:** Definition of s–t cuts, cut capacity, and why the max-flow value equals the min-cut capacity (the dual view).
- **Integrality Property:** If all capacities are integers, then there exists an integral maximum flow. (Useful when flows encode matchings or assignments.)
- **Implementation of Ford–Fulkerson and Edmonds–Karp** and knowing the guaranteed polynomial time.
- **Uniqueness / Non-uniqueness:** Why the maximum flow value is unique but the flow itself may not be.
- **Linear Programming:** Standard LP questions (including duality)

Potential Questions

A new problem that can be solved by reducing it to a matching problem.

Common tricks used in such reductions:

- Assignment problems can be handled similarly to maximum matching, where capacities ensure that an integral solution assigns only one object to each resource.

Potential Questions

A new problem that can be solved by reducing it to a matching problem.

Common tricks used in such reductions:

- Assignment problems can be handled similarly to maximum matching, where capacities ensure that an integral solution assigns only one object to each resource.
- When a problem appears to have multiple sources (or sinks), introduce a single global source (or sink) and connect all individual ones to it.

Potential Questions

A new problem that can be solved by reducing it to a matching problem.

Common tricks used in such reductions:

- Assignment problems can be handled similarly to maximum matching, where capacities ensure that an integral solution assigns only one object to each resource.
- When a problem appears to have multiple sources (or sinks), introduce a single global source (or sink) and connect all individual ones to it.
- To force certain vertices to stay together in the graph, we can use "infinite" edge weights (as in the project constraints from Lab 7). A similar trick can be used to prevent those edges from appearing in the cut.

Potential Questions

A new problem that can be solved by reducing it to a matching problem.

Common tricks used in such reductions:

- Assignment problems can be handled similarly to maximum matching, where capacities ensure that an integral solution assigns only one object to each resource.
- When a problem appears to have multiple sources (or sinks), introduce a single global source (or sink) and connect all individual ones to it.
- To force certain vertices to stay together in the graph, we can use "infinite" edge weights (as in the project constraints from Lab 7). A similar trick can be used to prevent those edges from appearing in the cut.
- Vertex splitting: Convert a vertex capacity to an "in-node" and "out-node" joined by a capacity edge.

Potential Questions

A new problem that can be solved by reducing it to a matching problem.

Common tricks used in such reductions:

- Assignment problems can be handled similarly to maximum matching, where capacities ensure that an integral solution assigns only one object to each resource.
- When a problem appears to have multiple sources (or sinks), introduce a single global source (or sink) and connect all individual ones to it.
- To force certain vertices to stay together in the graph, we can use "infinite" edge weights (as in the project constraints from Lab 7). A similar trick can be used to prevent those edges from appearing in the cut.
- Vertex splitting: Convert a vertex capacity to an "in-node" and "out-node" joined by a capacity edge.
- Flow Decomposition: A feasible solution of the problem can be the decomposition of the flow into path flows and cycle flows.

Potential Questions

A new problem that can be solved by reducing it to a matching problem.

Common tricks used in such reductions:

- Assignment problems can be handled similarly to maximum matching, where capacities ensure that an integral solution assigns only one object to each resource.
- When a problem appears to have multiple sources (or sinks), introduce a single global source (or sink) and connect all individual ones to it.
- To force certain vertices to stay together in the graph, we can use "infinite" edge weights (as in the project constraints from Lab 7). A similar trick can be used to prevent those edges from appearing in the cut.
- Vertex splitting: Convert a vertex capacity to an "in-node" and "out-node" joined by a capacity edge.
- Flow Decomposition: A feasible solution of the problem can be the decomposition of the flow into path flows and cycle flows.

Potential Questions

A new problem that can be solved by reducing it to a matching problem.

Common tricks used in such reductions:

- Assignment problems can be handled similarly to maximum matching, where capacities ensure that an integral solution assigns only one object to each resource.
- When a problem appears to have multiple sources (or sinks), introduce a single global source (or sink) and connect all individual ones to it.
- To force certain vertices to stay together in the graph, we can use "infinite" edge weights (as in the project constraints from Lab 7). A similar trick can be used to prevent those edges from appearing in the cut.
- Vertex splitting: Convert a vertex capacity to an "in-node" and "out-node" joined by a capacity edge.
- Flow Decomposition: A feasible solution of the problem can be the decomposition of the flow into path flows and cycle flows.

Proof of correctness: show a bidirectional relationship.

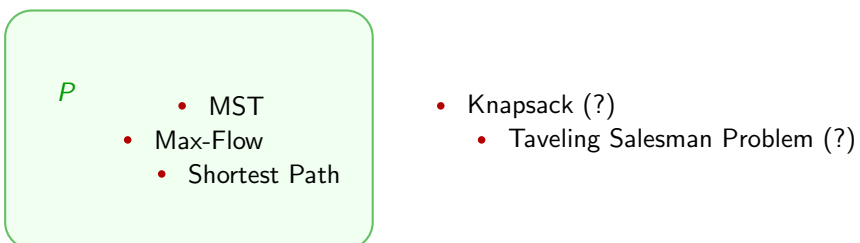
NP-completeness

P: Polynomial Time Solvable Problems

- Complexity theory classifies problems based on their *inherent difficulty*;
- Algorithms can be fast or slow, clever or naive, but our statements about the *problem itself*.
- A problem is polynomial time solvable if there is an algorithm that correctly solves it in $O(n^k)$ time, for some constant k, where n is the input length.
- still polynomial even $k = 10^{10}$.
- This is worst-case running time. (maximum running time over all possible inputs of size n)
- **P**: Problems solvable in **P**olynomial time (easy to **solve**).

P: Examples

- Typical examples:
  - Shortest paths (without nasty conditions like negative cycles).
  - Minimum spanning tree, maximum flow, bipartite matching, etc.
- Non-example: the standard dynamic programming for knapsack runs in $\Theta(nW)$ time, where W is the capacity; since the input size is only $\log W$, this is actually **pseudopolynomial**, not polynomial, in the input length.

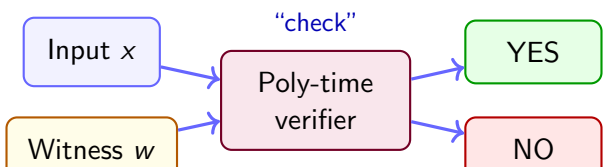


The Class NP

NP is the class of problems for which *solutions can be efficiently recognized*, even if we don't know how to find them efficiently.

A problem is in NP if:

- YES-instances have short **witnesses** (certificates) whose length is polynomial in the input size.
- We can verify a witness in polynomial time.



Cook–Levin Theorem

Key idea: Satisfiability (via CIRCUIT-SAT) can act as a *universal witness finder* for every problem in NP.

For any decision problem $A \in \text{NP}$ and any instance x of A:

- If x is a **YES**-instance, then there exists a witness w that convinces the verifier $V(x, w)$ to accept.
- If x is a **NO**-instance, then *no* witness can make the verifier accept.

The Cook–Levin theorem encodes this verifier behavior into a CIRCUIT-SAT formula. Given an instance x of A, we construct a Boolean formula Φ_x such that:

$$\Phi_x \text{ is satisfiable} \quad \Longleftrightarrow \quad \exists w : V(x, w) = \text{accept}.$$

Thus, CIRCUIT-SAT simulates the entire accepting computation of the verifier— it captures the witness *and* every step showing that the witness is correct.

NP-Hard and NP-Complete Problems

NP-Hard Problem

A problem B is **NP-hard** if for every problem $A \in \text{NP}$, A reduces to B.

NP-Hard and NP-Complete Problems

NP-Hard Problem

A problem B is **NP-hard** if for every problem $A \in \text{NP}$, A reduces to B.

NP-Complete Problem

A problem B is **NP-complete** if:

1. $B \in \text{NP}$, and
2. For every problem $A \in \text{NP}$, A reduces to B.

The NP-Completeness Recipe

To show a new problem B is NP-complete, start from a known NP-complete problem A. Show a polynomial-time reduction from A to B.

A is NP-complete: $\text{NP} \leq_p A$

A poly-time reduction from A to B: $A \leq_p B$

$$\Big\} \implies \text{NP} \leq_p B$$

The NP-Completeness Recipe

To show a new problem B is NP-complete, start from a known NP-complete problem A. Show a polynomial-time reduction from A to B.

A is NP-complete: $\text{NP} \leq_p A$

A poly-time reduction from A to B: $A \leq_p B$

$$\Big\} \implies \text{NP} \leq_p B$$

The NP-Completeness Recipe

To show a new problem B is NP-complete, start from a known NP-complete problem A. Show a polynomial-time reduction from A to B.

A is NP-complete: $\text{NP} \leq_p A$

A poly-time reduction from A to B: $A \leq_p B$

$\Longrightarrow \text{NP} \leq_p B$

B is NP-hard.

The NP-Completeness Recipe

To show a new problem B is NP-complete, start from a known NP-complete problem A. Show a polynomial-time reduction from A to B.

A is NP-complete: $\text{NP} \leq_p A$

A poly-time reduction from A to B: $A \leq_p B$

$\Longrightarrow \text{NP} \leq_p B$

B is NP-hard.

If we also show that B is in NP, then $\Longrightarrow$ B is NP-complete

What You Need to Know

- Understand the notions of **P**, **NP**, polynomial-time reductions, and the general recipe for proving NP-completeness.
- Be familiar with the **problems we defined**. You should know their general structure and which problems are "close enough" so that a reduction between them makes sense.
- These reductions are especially important to study carefully:
  - **3-SAT** to **Maximum Independent Set**
  - **Vertex Cover** to **Set Cover**
  - Conceptual message of Cook-Levin's theorem.

  Also review the reductions in the problem sets and in Lab 8.
- For the remaining reductions, you do *not* need to memorize the full proofs. However, you should know the problems themselves and the **general relationships** that motivate the reductions.

Potential Questions

- A new question that can be solved via reduction (to any of the previously known algorithms).
- Study the approximation algorithms (may be greedy algorithm). Vertex cover, TSP, set cover.
- For all such question follow the instruction given in homework 6.

Guidelines for Writing Algorithm Correctness Proofs

Goal: Show that the algorithm *always* produces a correct answer for *every* valid input.

General Structure:

- **Prove That the Algorithm Terminates.**
- **Follow a Logical Chain of Implications.** Write the proof so each claim follows explicitly:

$$p_1 \Rightarrow p_2 \Rightarrow p_3 \Rightarrow \cdots \Rightarrow q.$$

- Make every step justified — no intuition gaps. Use definitions, earlier lemmas, or properties of the algorithm (e.g., cut property, greedy choice, residual capacity).