COMP 382: Reasoning about Algorithms

Max Flows and Its Applications

Prof. Maryam Aliakbarpour

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

November 6, 2025

Today's Lecture

1. **Flow Decomposition**

2. **Reductions**

3. **Bipartite Matching**

4. **Edge-Disjoint Paths**

5. **Vertex-Disjoint Paths**

Reading:

- Chapter 10 and Chapter 11 of the *Algorithms* book [Erickson, 2019]

Content adapted from the same chapters in [Erickson, 2019].

1. Flow Decomposition

Flow Components: Path and Cycle Flows

Every flow is a combination of these two fundamental unit flows.

1. Path Flow (Unit Flow)

For a directed path P from s to t:

- **Value:** $|P| = 1$.
- **Definition:** The unit flow $P : E \to \mathbb{R}$ is defined as:

$$P(u \to v) = \begin{cases} 1 & \text{if } u \to v \in P \\ -1 & \text{if } v \to u \in P \\ 0 & \text{otherwise} \end{cases}$$

Flow Components: Path and Cycle Flows

Every flow is a combination of these two fundamental unit flows.

2. Cycle Flow (Circulation)

For a directed cycle C:

- **Value:** $|C| = 0$.
- **Definition:** The unit flow $C : E \to \mathbb{R}$ is defined as:

$$C(u \to v) = \begin{cases} 1 & \text{if } u \to v \in C \\ -1 & \text{if } v \to u \in C \\ 0 & \text{otherwise} \end{cases}$$

Flow Linearity:

For now, ignore the capacities...

- A flow is essentially a function mapping an edge to a number.
- It also consistently maps the edge in the opposite direction to the negative of that number.

$$f(u, v) = -f(v, u)$$

- Any linear combination of (s, t)-flows is also an (s, t)-flow.

$$\text{If } h = \alpha f + \beta g$$

  - shorthand for: $h(u, v) = \alpha f(u, v) + \beta g(u, v)$
- The size of the flow is also preserved:

$$|h| = \alpha|f| + \beta|g|$$

The Flow Decomposition Theorem

Every flow is a combination of these two fundamental unit flows: Paths and Cycles.

Theorem

Every **non-negative** (s, t)-flow f can be written as a **positive linear combination** of directed (s, t)-paths and directed cycles.

Applications: Many practical problems (e.g., transportation, communication, logistics) need a list of specific routes used by the flow, not just edge capacities. This theorem, and its associated algorithm, allow us to convert a flow solution to a path-based representation.

Proof Idea: Flow Decomposition (Induction I)

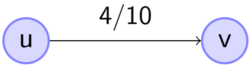
The proof uses induction on $\#\mathbf{f}$, the number of edges carrying non-zero flow.

1. **Base Case:** If $\#f = 0$, the flow is trivially decomposed.

Proof Idea: Flow Decomposition (Induction I)

The proof uses induction on $\#\mathbf{f}$, the number of edges carrying non-zero flow.

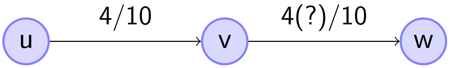
1. **Base Case:** If $\#f = 0$, the flow is trivially decomposed.
2. **Inductive Step (Finding a Flow Structure):**
   - Since $\#f > 1$ an edge (u, v) exists with positive flow.



Proof Idea: Flow Decomposition (Induction I)

The proof uses induction on $\#\mathbf{f}$, the number of edges carrying non-zero flow.

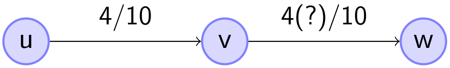
1. **Base Case:** If $\#f = 0$, the flow is trivially decomposed.
2. **Inductive Step (Finding a Flow Structure):**
   - Since $\#f > 1$ an edge (u, v) exists with positive flow.
   - By the *Flow Conservation Property* (for any node $v \in V \setminus \{s, t\}$), if flow enters v, it must also leave v.



Proof Idea: Flow Decomposition (Induction I)

The proof uses induction on $\#\mathbf{f}$, the number of edges carrying non-zero flow.

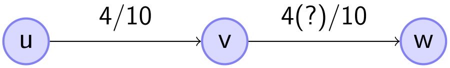
1. **Base Case:** If $\#f = 0$, the flow is trivially decomposed.
2. **Inductive Step (Finding a Flow Structure):**
   - Since $\#f > 1$ an edge (u, v) exists with positive flow.
   - By the *Flow Conservation Property* (for any node $v \in V \setminus \{s, t\}$), if flow enters v, it must also leave v.
   - This property guarantees that as long as we are not at s or t, we can *always extend the walk* to an outgoing edge with positive flow.



Proof Idea: Flow Decomposition (Induction I)

The proof uses induction on $\#\mathbf{f}$, the number of edges carrying non-zero flow.

1. **Base Case:** If $\#f = 0$, the flow is trivially decomposed.
2. **Inductive Step (Finding a Flow Structure):**
   - Since $\#f > 1$ an edge (u, v) exists with positive flow.
   - By the *Flow Conservation Property* (for any node $v \in V \setminus \{s, t\}$), if flow enters v, it must also leave v.
   - This property guarantees that as long as we are not at s or t, we can *always extend the walk* to an outgoing edge with positive flow.
   - The walk must eventually either reach s/t (forming an $s \to t$ Path) or visit a vertex twice (forming a Cycle).



Proof Idea: Flow Decomposition

Once a path or cycle structure is found, we apply the recursive step.

3. **Decompose and Recurse:**
   - Let S be the found structure (Path P or Cycle C).
   - Determine the bottleneck flow $F = \min_{e \in S} f(e)$.
   - Construct a new flow $f' = f - F \cdot S$.

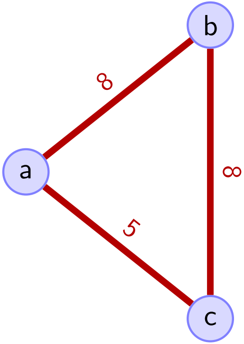
Proof Idea: Flow Decomposition

Once a path or cycle structure is found, we apply the recursive step.

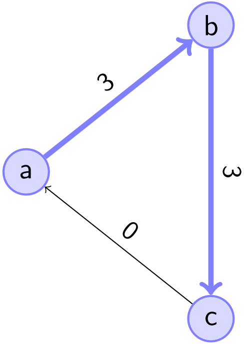
3. **Decompose and Recurse:**
   - Let S be the found structure (Path P or Cycle C).
   - Determine the bottleneck flow $F = \min_{e \in S} f(e)$.
   - Construct a new flow $f' = f - F \cdot S$.
4. **Conclusion:**
   - Subtracting F units empties **at least one edge** in S, so the new flow $\#f' < \#f$.
   - By the inductive hypothesis, f' is decomposed. Adding back $F \cdot S$ completes the decomposition of f.

$$f = f' + F \cdot S$$

Removing Flow Component



Cycle C: $a \to b \to c \to a$.
Bottleneck $F = \min(8, 8, 5) = \mathbf{5}$.



Flow after removing $5 \cdot C$.

Implications of Decomposition Theorem

- The proof also immediately translates directly into an algorithm.
  - The total number of paths and cycles in the decomposition is at most $|E|$, the number of edges in the network.
  - Finding a cycle or a path takes $O(|V|)$ (why not $O(|E|)$?)
  - The total time for decomposition is $O(|V| \cdot |E|)$.

- Any circulation ($|f| = 0$) can be decomposed into a weighted sum of cycles; no paths are necessary.

- Any acyclic (s, t)-flow can be decomposed into a weighted sum of (s, t)-paths; no cycles are necessary.

Flow of size $|f| \Rightarrow |f|$ Paths (Integral Case)

Goal. From an *integral* (s,t)-flow f of value $|f|$, produce exactly $|f|$ unit $s \to t$ paths whose sum equals f.

Key observations.

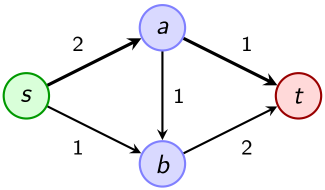
- **Decomposition:** $f = (\text{paths}) + (\text{cycles})$; cycles carry 0 value and can be removed.
- **Integrality:** With integral capacities, we can take a max flow that is integral.

Idea. Make f acyclic, then repeatedly extract unit $s \to t$ paths until no flow remains.

Greedy Extraction of $|f|$ Unit Paths

Algorithm.

1. **Acyclicify:** While a directed cycle exists in the support of f, subtract its bottleneck flow.
2. **Repeat $|f|$ times:**
   2.1 From s, follow any edge with $f(e) > 0$ until t.
   2.2 Record P_i and set $f(e) \leftarrow f(e) - 1$ for all $e \in P_i$.



Bold edges: extracted path P_i
Labels: current $f(e)$ before subtracting 1

Why it works.

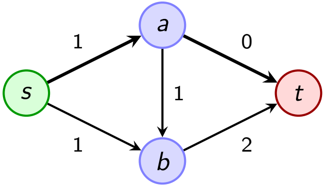
- Acyclic positive flow lies on $s \to t$ paths.
- Each subtraction preserves feasibility and integrality.

Running time: $O(|E||V|)$.

Greedy Extraction of $|f|$ Unit Paths

Algorithm.

1. **Acyclicify:** While a directed cycle exists in the support of f, subtract its bottleneck flow.
2. **Repeat $|f|$ times:**
   2.1 From s, follow any edge with $f(e) > 0$ until t.
   2.2 Record P_i and set $f(e) \leftarrow f(e) - 1$ for all $e \in P_i$.



Bold edges: extracted path P_i
Labels: current $f(e)$ after subtracting 1

Why it works.

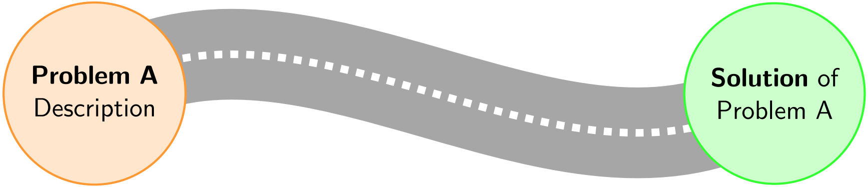
- Acyclic positive flow lies on $s \to t$ paths.
- Each subtraction preserves feasibility and integrality.

Running time: $O(|E||V|)$.

2. Reductions

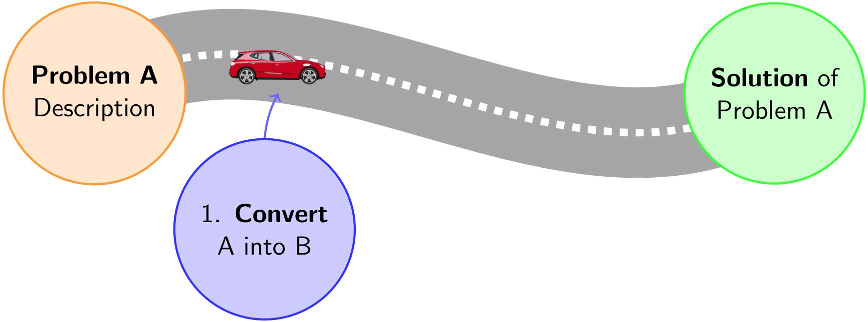
Solving New Problems by Reusing Old Ones

Solving Problem A by Reduction to Problem B



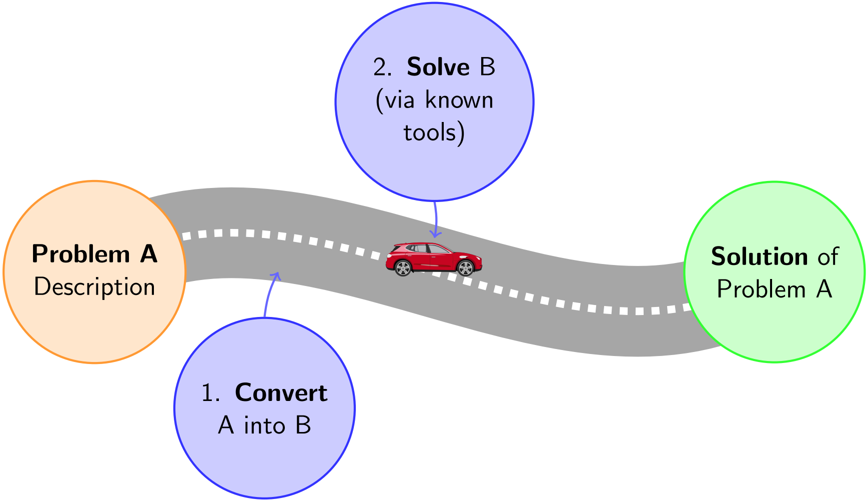
Car Image Designed by Image Sarovar

Solving Problem A by Reduction to Problem B



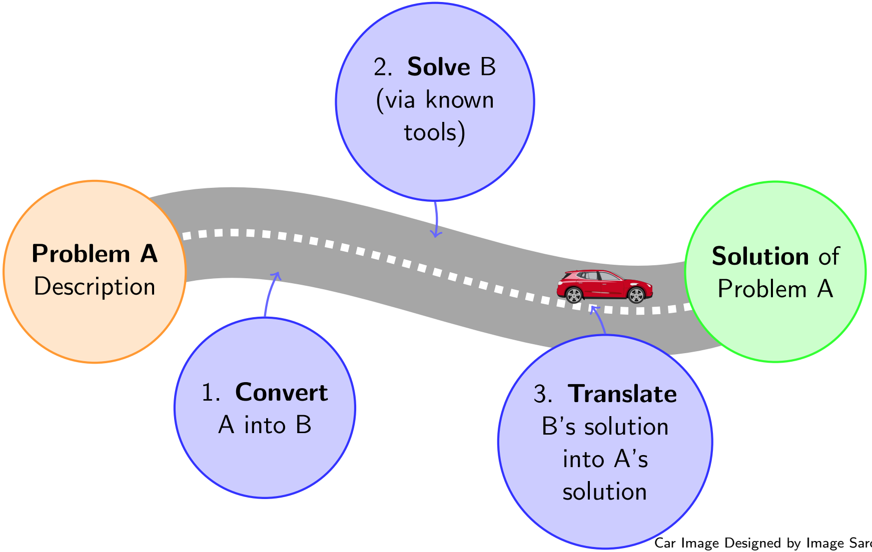
Car Image Designed by Image Sarovar

Solving Problem A by Reduction to Problem B



Car Image Designed by Image Sarovar

Solving Problem A by Reduction to Problem B



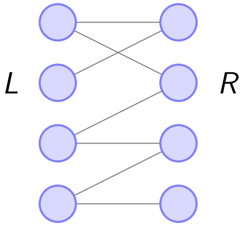
Car Image Designed by Image Sarovar

3. Bipartite Matching

Reducing bipartite matching to max-flow

The Bipartite Matching Problem

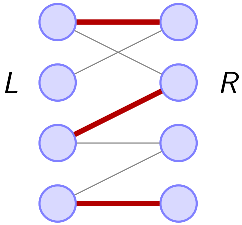
A **bipartite graph** is a graph where vertices can be divided into two disjoint sets, L and R, such that every edge connects a vertex in L to one in R.



The Bipartite Matching Problem

A **bipartite graph** is a graph where vertices can be divided into two disjoint sets, L and R, such that every edge connects a vertex in L to one in R.

A **matching** is a set of edges with no common vertices.

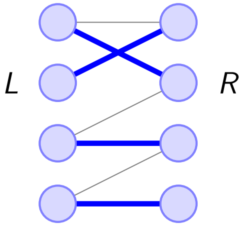


The Bipartite Matching Problem

A **bipartite graph** is a graph where vertices can be divided into two disjoint sets, L and R, such that every edge connects a vertex in L to one in R.

A **matching** is a set of edges with no common vertices.

Goal: Find the **maximum matching** - the matching with the largest possible number of edges.



Example: Assigning Jobs

Imagine we have a set of applicants and a set of available jobs. An edge exists if an applicant is qualified for a job.

Problem: How do we hire the maximum number of applicants, assigning each to a single job they are qualified for?

Applicants (L)

- Alice
- Bob
- Carol

Jobs (R)

- </> Coder
- Designer
- Analyst

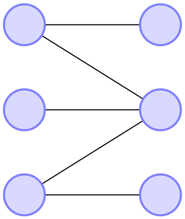
This is a maximum bipartite matching problem.

Convert Bipartite Matching to Max Flow

From Matching to Max Flow: The Construction

We are given a bipartite graph G for which we would like to find the maximum matching.

We convert the bipartite graph G into a flow network G'.

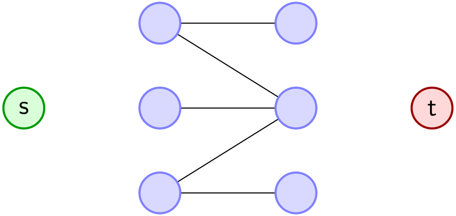


From Matching to Max Flow: The Construction

We are given a bipartite graph G for which we would like to find the maximum matching.

We convert the bipartite graph G into a flow network G'.

1. Create a **source** s and a **sink** t.

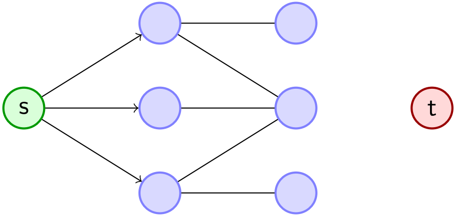


From Matching to Max Flow: The Construction

We are given a bipartite graph G for which we would like to find the maximum matching.

We convert the bipartite graph G into a flow network G'.

1. Create a **source** s and a **sink** t.
2. Add edges from s to every vertex in L.

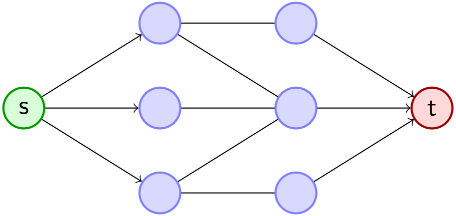


From Matching to Max Flow: The Construction

We are given a bipartite graph G for which we would like to find the maximum matching.

We convert the bipartite graph G into a flow network G'.

1. Create a **source** s and a **sink** t.
2. Add edges from s to every vertex in L.
3. Add edges from every vertex in R to t.

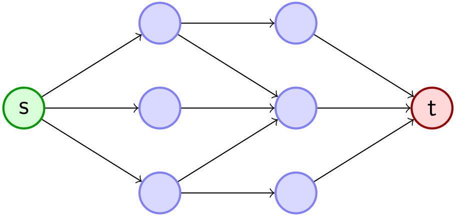


From Matching to Max Flow: The Construction

We are given a bipartite graph G for which we would like to find the maximum matching.

We convert the bipartite graph G into a flow network G'.

1. Create a **source** s and a **sink** t.
2. Add edges from s to every vertex in L.
3. Add edges from every vertex in R to t.
4. Direct original edges from L to R.

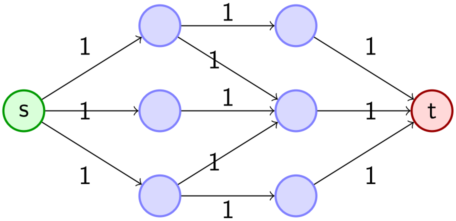


From Matching to Max Flow: The Construction

We are given a bipartite graph G for which we would like to find the maximum matching.

We convert the bipartite graph G into a flow network G'.

1. Create a **source** s and a **sink** t.
2. Add edges from s to every vertex in L.
3. Add edges from every vertex in R to t.
4. Direct original edges from L to R.
5. Assign **capacity 1** to ALL edges.



Why This Works: The Core Intuition

Key Idea

The value of the maximum flow in the constructed network G' is equal to the size of the maximum matching in the original bipartite graph G.

- Because all capacities are 1, the Ford-Fulkerson algorithm will produce an integer-valued flow (either 0 or 1 on each edge).

Why This Works: The Core Intuition

Key Idea

The value of the maximum flow in the constructed network G' is equal to the size of the maximum matching in the original bipartite graph G.

- Because all capacities are 1, the Ford-Fulkerson algorithm will produce an integer-valued flow (either 0 or 1 on each edge).
- A flow of 1 along a path $s \to u \to v \to t$ corresponds to selecting the edge (u, v) for our matching.

Why This Works: The Core Intuition

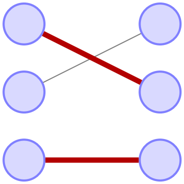
Key Idea

The value of the maximum flow in the constructed network G' is equal to the size of the maximum matching in the original bipartite graph G.

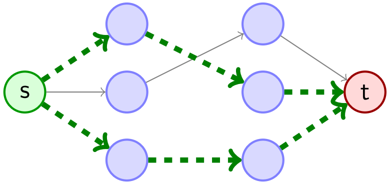
- Because all capacities are 1, the Ford-Fulkerson algorithm will produce an integer-valued flow (either 0 or 1 on each edge).
- A flow of 1 along a path $s \to u \to v \to t$ corresponds to selecting the edge (u, v) for our matching.
- The capacity constraints enforce the matching rules:
  - Edge $s \to u$ (cap 1): Vertex $u \in L$ is in at most one matched edge.
  - Edge $v \to t$ (cap 1): Vertex $v \in R$ is in at most one matched edge.

Example: Matching to Flow

A matching in G corresponds to a valid flow in G'.



A Matching of Size 2



A Flow of Value 2

Augmenting Paths vs. Alternating Paths

The Ford-Fulkerson algorithm's search for an **augmenting path** in the flow network G' has a direct parallel in the original bipartite graph G.

An augmenting path in G' corresponds to an **alternating path** in G.

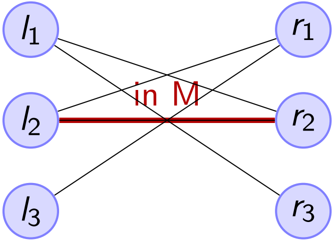
- An alternating path starts at an unmatched vertex in L.
- It ends at an unmatched vertex in R.
- It alternates between edges **not in** the current matching and edges **in** the current matching.

Finding and using an alternating path increases the size of the matching by one, just as an augmenting path increases the flow value by one.

Example: An Alternating Path

Flipping the edges along the alternating path gives a larger matching.

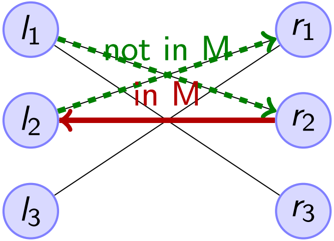
- Initial Matching: $\{(l_2, r_2)\}$



Example: An Alternating Path

Flipping the edges along the alternating path gives a larger matching.

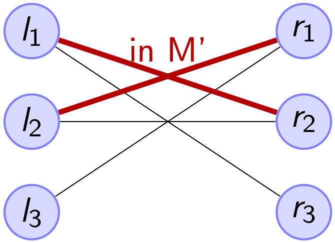
- Initial Matching: $\{(l_2, r_2)\}$
- Alternating Path: $l_1 \to r_2 \to l_2 \to r_1$



Example: An Alternating Path

Flipping the edges along the alternating path gives a larger matching.

- Initial Matching: $\{(l_2, r_2)\}$
- Alternating Path: $l_1 \to r_2 \to l_2 \to r_1$
- New Matching: $\{(l_1, r_2), (l_2, r_1)\}$



Algorithm Summary & Complexity

To find a maximum bipartite matching:

1. Construct the flow network G' from the bipartite graph G. This takes $O(V + E)$ time.
2. Compute the maximum flow from s to t in G'.
   - The value of the max flow, $|f^*|$, is the size of the maximum matching.
   - Using the standard Ford-Fulkerson algorithm, this takes $O(|f^*|E)$.
   - Since $|f^*| \leq V$, the complexity is $O(VE)$.
3. The set of edges from L to R with flow equal to 1 forms the maximum matching.

More advanced algorithms like Hopcroft-Karp can find maximum matchings in $O(E\sqrt{V})$ time.

Summary

- The **Maximum Bipartite Matching** problem is a fundamental problem with many applications (e.g., assignments, scheduling).
- It can be elegantly solved by reducing it to a **Maximum Flow** problem.
- The key is to construct a special flow network where all edge capacities are 1.
- The value of the max flow in this network equals the size of the max matching.
- The concept of an **augmenting path** in flow analysis corresponds directly to an **alternating path** in matching theory.

Edge-Disjoint Paths

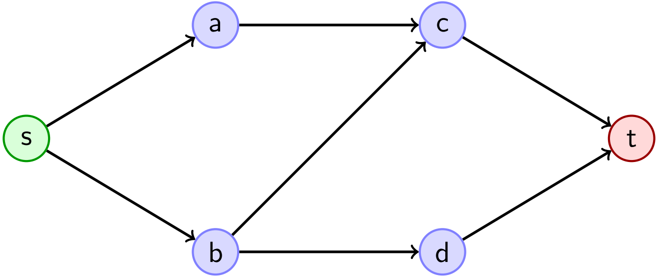
In directed graphs

The Edge-Disjoint Path Problem

Given a directed graph $G = (V, E)$ and two vertices s and t.

Problem: Find the *maximum* number of paths from s to t that are *edge-disjoint*.

- A set of paths is edge-disjoint if no two paths share an edge.
- Paths *are* allowed to share vertices.

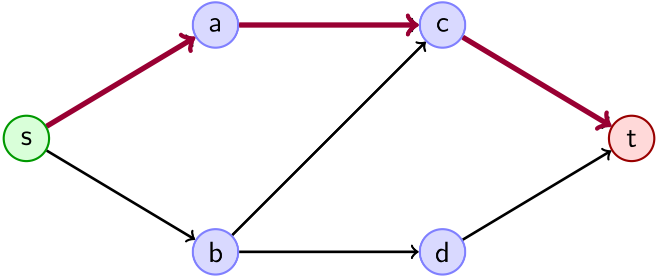


The Edge-Disjoint Path Problem

Given a directed graph $G = (V, E)$ and two vertices s and t.

Problem: Find the *maximum* number of paths from s to t that are *edge-disjoint*.

- A set of paths is edge-disjoint if no two paths share an edge.
- Paths *are* allowed to share vertices.

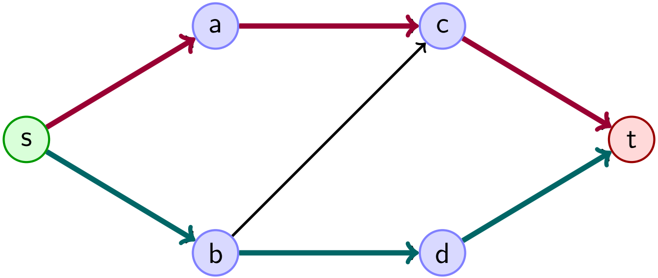


The Edge-Disjoint Path Problem

Given a directed graph $G = (V, E)$ and two vertices s and t.

Problem: Find the *maximum* number of paths from s to t that are *edge-disjoint*.

- A set of paths is edge-disjoint if no two paths share an edge.
- Paths *are* allowed to share vertices.

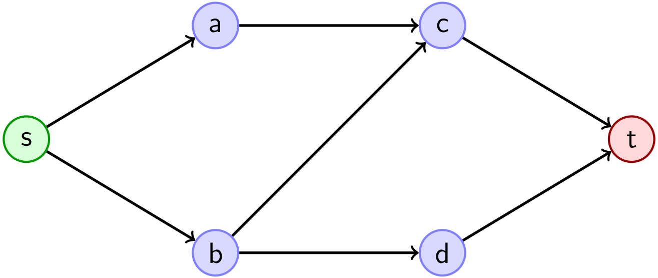


These two paths are edge-disjoint.

From Edge-Disjoint Paths to Max Flow

We can reduce this path problem to a max-flow problem:

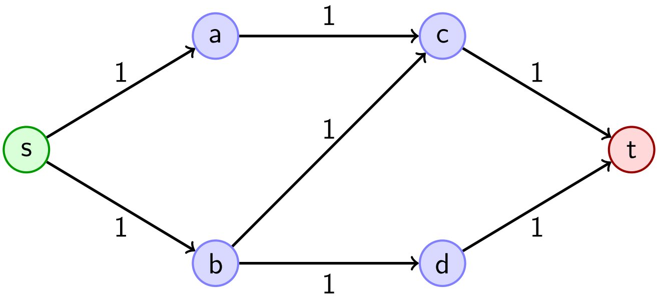
1. Take the original graph $G = (V, E)$, and create a flow network $G' = (V, E, s, t, c)$.



From Edge-Disjoint Paths to Max Flow

We can reduce this path problem to a max-flow problem:

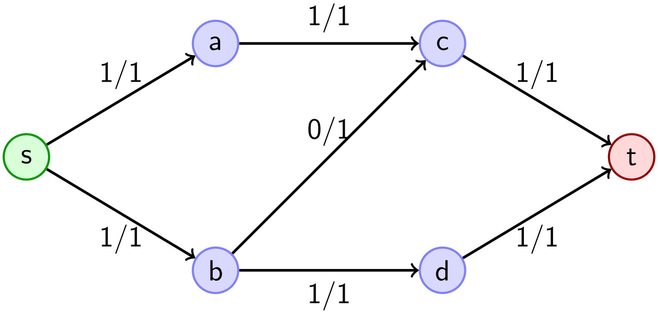
1. Take the original graph $G = (V, E)$, and create a flow network $G' = (V, E, s, t, c)$.
2. Assign a capacity of $\mathbf{1}$ to *every* edge $e \in E$.



From Edge-Disjoint Paths to Max Flow

We can reduce this path problem to a max-flow problem:

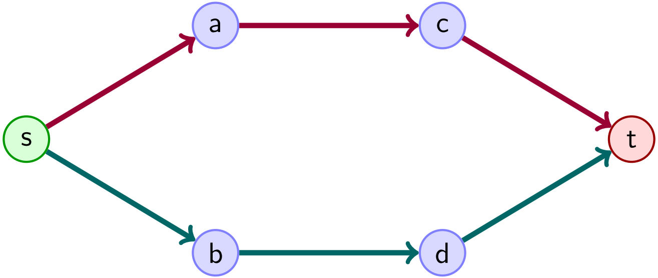
1. Take the original graph $G = (V, E)$, and create a flow network $G' = (V, E, s, t, c)$.
2. Assign a capacity of $\mathbf{1}$ to *every* edge $e \in E$.
3. Compute the maximum (s, t)-flow in G'.



From Edge-Disjoint Paths to Max Flow

We can reduce this path problem to a max-flow problem:

1. Take the original graph $G = (V, E)$, and create a flow network $G' = (V, E, s, t, c)$.
2. Assign a capacity of $\mathbf{1}$ to *every* edge $e \in E$.
3. Compute the maximum (s, t)-flow in G'.
4. Compute the path decomposition of the max flow



From Edge-Disjoint Paths to Max Flow

Running Time: The max flow value $|f^*|$ is at most $V - 1$ (the capacity of the cut $(\{s\}, V \setminus \{s\})$). Using Ford-Fulkerson, the time is $O(|f^*|E) = O(VE)$ time.

From Edge-Disjoint Paths to Max Flow

Running Time: The max flow value $|f^*|$ is at most $V - 1$ (the capacity of the cut $(\{s\}, V \setminus \{s\})$). Using Ford-Fulkerson, the time is $O(|f^*|E) = O(VE)$ time.

Proof of Correctness: Why does this algorithm work?

- If k edge-disjoint paths exist $\Rightarrow$ A valid flow of size k exists.
- If flow of size k exists $\Rightarrow$ We can construct k edge-disjoint paths.

Equivalence: Paths to Flow

Claim: A set of k edge-disjoint paths from s to t can be converted into a valid (s, t)-flow of value k.

How: Push 1 unit of flow along each of the k paths.

- *Capacity Constraint:* Since the paths are edge-disjoint, each edge is used at most once. The flow on any edge is either 0 or 1, which does not exceed its capacity of 1.
- *Flow Conservation:* This holds at every vertex $v \notin \{s, t\}$.

The total flow leaving s (and entering t) is exactly k.

The max-flow in that graph is at least k:

$$\textit{Max Flow Value} \geq \textit{Max Number of Edge-Disjoint Paths}$$

Equivalence: Flow to Paths

Claim: An integer-valued (s, t)-flow f of value k can be decomposed into k edge-disjoint paths from s to t.

How:

- Since all capacities are integers (they are all 1), the Ford-Fulkerson algorithm (and others) guarantees an integer-valued max flow. Every edge will have flow 0 or 1.
- By the *Flow Decomposition Theorem*, any valid s-t flow can be decomposed into a set of paths and cycles.
- The value of the flow, k, is exactly the number of s-t paths in this decomposition.
- Since each edge has capacity 1, no edge can be used by more than one path.

$$\textit{Max Flow Value} \leq \textit{Max Number of Edge-Disjoint Paths}$$

Edge-Disjoint Paths

In undirected graphs

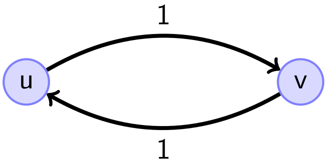
Edge-Disjoint Paths in Undirected Graphs

Problem: Find the max number of edge-disjoint paths from s to t in an *undirected* graph G.

Reduction:

1. Create a new *directed* graph G'.
2. For each undirected edge $\{u, v\}$ in G, add two directed edges to G':
   - (u, v) with capacity 1
   - (v, u) with capacity 1
3. ...

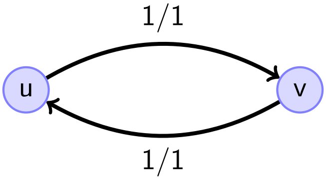
Undirected Edge



Becomes Two Directed Edges

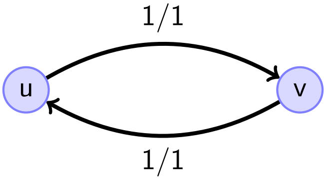
Edge-Disjoint Paths in Undirected Graphs

This situation is problematic because the effective capacity of edge (u, v) becomes 2, allowing two distinct paths to share the same edge.



Edge-Disjoint Paths in Undirected Graphs

This situation is problematic because the effective capacity of edge (u, v) becomes 2, allowing two distinct paths to share the same edge.



Solution: If the flow saturates both (u, v) and (v, u), this forms a cycle. We can remove this cycle from the flow without changing the total value. Thus, we can find an acyclic max flow, and the resulting paths in G' correspond to edge-disjoint paths in G.

Vertex-Disjoint Paths

The Vertex-Disjoint Path Problem

Given a directed graph $G = (V, E)$ and two vertices s and t.

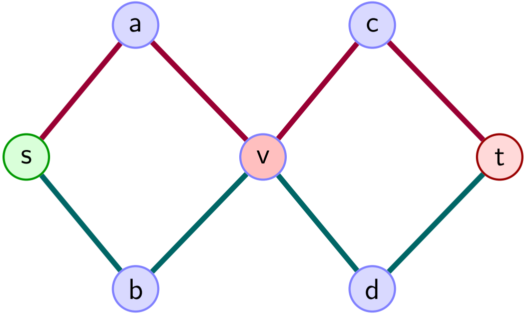
Problem: Find the *maximum* number of paths from s to t that are *vertex-disjoint*.

- A set of paths is vertex-disjoint if no two paths share an intermediate vertex (i.e., any vertex other than s or t).

The Vertex-Disjoint Path Problem

Not Vertex-Disjoint (Shares vertex v)

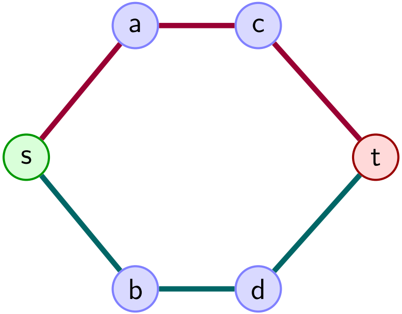
$s \to a \to v \to c \to t$ and $s \to b \to v \to d \to t$



The Vertex-Disjoint Path Problem

Vertex-Disjoint

$s \to a \to v \to c \to t$ and $s \to b \to v \to d \to t$



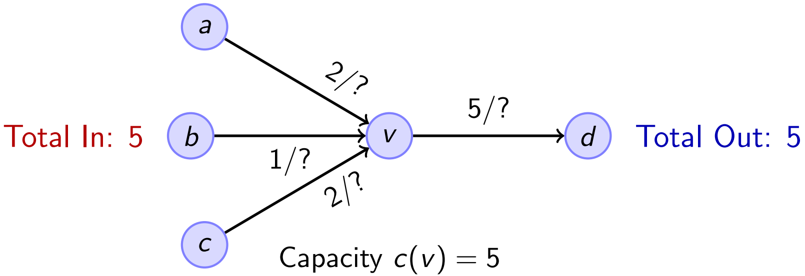
A New Tool: Vertex Capacities

To solve this, we first introduce a more general problem: what if *vertices* have capacities?

We can add a constraint for each vertex $v \notin \{s, t\}$:

$$\sum_{u \in V} f(u, v) \leq c(v)$$

The total flow *into* vertex v is at most its capacity $c(v)$.



The Reduction: Vertex Splitting

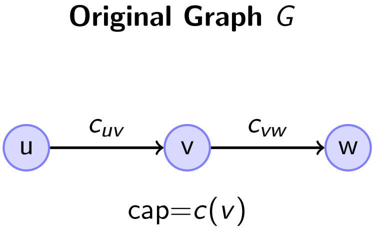
We can reduce a vertex-capacity problem to a standard max-flow problem using *vertex splitting*.

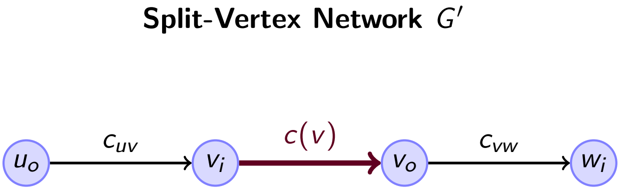
For *each* vertex v with a capacity $c(v)$ (and $v \notin \{s, t\}$):

1. Replace v with two new vertices: v_i and v_o.
2. Add a new directed edge (v_i, v_o) with capacity $c(v)$.
3. For every original edge (u, v), create a new edge (u_o, v_i).
4. For every original edge (v, w), create a new edge (v_o, w_i).

(For s and t, we just use $s = s_o$ and $t = t_i$).

The Reduction: Vertex Splitting





Any flow passing *through* v in G must now pass *through the edge* (v_i, v_o) in G', which enforces the capacity constraint.

Putting It All Together

Now we can solve the vertex-disjoint path problem:

1. We want to find paths where each intermediate vertex is used at most **once**.
2. This is a max-flow problem where all intermediate vertices $v \notin \{s, t\}$ have a capacity of $c(v) = 1$.
3. We also want paths to be edge-disjoint, so we can set all *edge* capacities to 1 as well.

The Algorithm:

1. For every vertex $v \notin \{s, t\}$, apply the vertex-splitting reduction:
   - Create v_i and v_o.
   - Add edge (v_i, v_o) with capacity **1**.
2. For every original edge (u, v):
   - If $u = s$, add edge (s, v_i) with capacity 1.
   - If $v = t$, add edge (u_o, t) with capacity 1.
   - Otherwise, add edge (u_o, v_i) with capacity 1.
3. Compute the max (s, t)-flow in this new network G'.

Why The Reduction Works

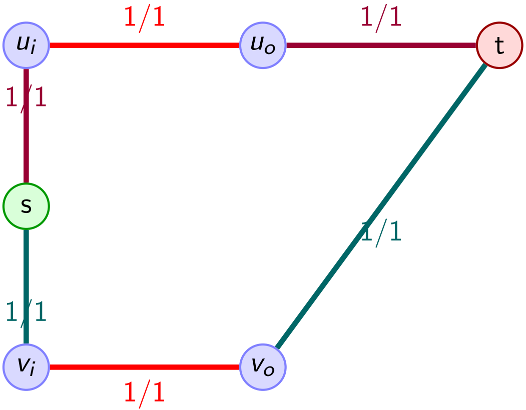
The first direction will trivially hold. If we have k-vertex disjoint path, we can push k units of flow.

Why The Reduction Works

The first direction will trivially hold. If we have k-vertex disjoint path, we can push k units of flow.

For the other direction, we compute the max flow in the new network G', where all edges have capacity 1.

- The max flow will be integer-valued, k.
- By flow decomposition, this corresponds to k paths from s to t.
- Because the "original" edges (like (u_o, v_i)) have capacity 1, no two paths can share an original edge.
- Because the "vertex" edges (like (v_i, v_o)) have capacity 1, no two paths can share an intermediate vertex.



A flow of value 2 corresponds to 2 vertex-disjoint paths. □

References

Erickson, J. (2019).
Algorithms.
Self-published.