Week 9 — A*, All-Pairs Shortest Paths, Maximum Flow & Bipartite Matching

Félix Chavelli felix.chavelli@inria.fr

April 22, 2026 · Semester 2

PSL
UNIVERSITÉ PARIS

Today's Agenda

Motivation & Recap

A* Search

All-Pairs Shortest Paths

Maximum Flow

Bipartite Graphs & Maximum Matching

Summary & What's Next

Part 1

Motivation & Recap

Last Week: MST & Single-Source Shortest Paths

What we covered:

- MST: Kruskal ($\mathcal{O}(E \log V)$) and Prim ($\mathcal{O}(E \log V)$)
- SSSP: Bellman-Ford ($\mathcal{O}(VE)$), Dijkstra ($\mathcal{O}(E \log V)$)
- Relaxation as universal building block
- Negative-weight edges vs. negative cycles

Today: four new topics!

- A* Search: heuristic-guided shortest path to a specific target
- All-Pairs Shortest Paths: distances between *every* pair of vertices
- Maximum Flow: how much can we push through a network?
- Bipartite Matching: optimally pair items from two groups

Why These Three Topics?

A* Search

- GPS navigation & route planning
- Video game pathfinding (NPCs)
- Robot motion planning
- Puzzle solving (15-puzzle, Rubik's cube)

All-Pairs Shortest Paths

- Network diameter computation
- Centrality measures in social networks
- Pre-compute routing tables
- Transitive closure of relations

Maximum Flow

- Network bandwidth optimization
- Supply chain & logistics
- Image segmentation (graph cuts)
- Airline scheduling

Bipartite Matching

- Job/internship assignment
- Scheduling interviews (candidates ↔ time slots)
- Resource allocation (servers ↔ tasks)
- Organ donor matching programs
- Student ↔ project allocation

Key Idea

Maximum flow is a **meta-algorithm**: many combinatorial problems *reduce* to max flow!

Part 2

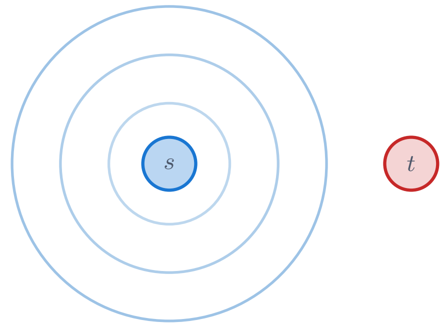
A* Search

Dijkstra's Limitation

Dijkstra finds shortest paths from s to *all* vertices.

But what if we only need $s \to t$?

Dijkstra expands outward like a **growing circle**: no notion of *where* t is.



Dijkstra: explores uniformly

Key Idea

Can we **guide** the search *toward* the target?

$\Rightarrow$ **A***: use a *heuristic* to estimate remaining cost!

Heuristic Functions

Heuristic Function

$h : V \to \mathbb{R}_{\geqslant 0}$ estimates the cost from v to target t.

Used to *guide* the search toward t instead of expanding uniformly.

Common heuristics for grid pathfinding:

- **Manhattan distance** (4-directional moves): $h(v) = |x_v - x_t| + |y_v - y_t|$
- **Euclidean distance** (any-angle moves): $h(v) = \sqrt{(x_v - x_t)^2 + (y_v - y_t)^2}$
- $h \equiv 0$ (no information): A* degenerates into Dijkstra

Key Idea

Two key properties make heuristics useful: **admissibility** (guarantees optimality) and **consistency** (guarantees no reopening).

Admissible Heuristics

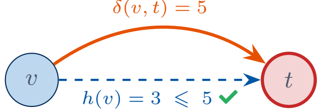
Admissibility

h is **admissible** if it *never overestimates* the true cost:

$$h(v) \leqslant \delta(v,t) \quad \text{for all } v \in V.$$

Intuition: an optimistic guess.

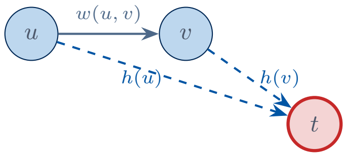
- Manhattan distance on a 4-grid is admissible: any actual path cannot be shorter than the straight-line L^1 distance.
- Euclidean distance is admissible whenever edge weights $\geqslant$ Euclidean length.
- $h \equiv 0$ is trivially admissible.



Consistent Heuristics

Consistency (monotonicity)

For every edge (u, v) and target t: $h(u) \leqslant w(u, v) + h(v)$ and $h(t) = 0$. (Triangle inequality on h.)



Manhattan / Euclidean on uniform grids are consistent.

$f(v) = g(v) + h(v)$ is

non-decreasing along any path.

Consistency $\Rightarrow$ Admissibility

Let $v = v_0 \to v_1 \to \cdots \to v_k = t$ be a shortest path from v to t. Iterating consistency:

$$h(v_0) \leqslant w(v_0, v_1) + h(v_1) \leqslant \cdots \leqslant \sum_{i=0}^{k-1} w(v_i, v_{i+1}) + h(v_k).$$

Since $h(v_k) = h(t) = 0$ and $\sum w(v_i, v_{i+1}) = \delta(v, t)$:

$$h(v) \leqslant \delta(v, t), \quad \text{i.e. } h \text{ is admissible.} \square$$

Converse fails: admissible $\not\Rightarrow$ consistent.

A* — The Algorithm

Key Idea

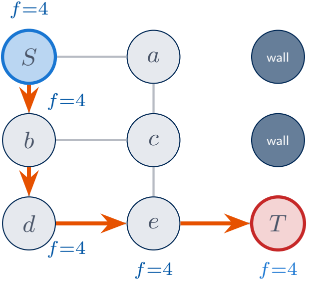
Instead of ordering by $g(v)$ (Dijkstra), A* orders by $f(v) = g(v) + h(v)$ — estimated **total cost** through v.

Input: Graph G, source s, target t, admissible heuristic h

1: $g[s] \leftarrow 0$; all other $g[v] \leftarrow \infty$
2: Insert s into priority queue Q with key $f(s) = h(s)$
3: *closed* $\leftarrow \emptyset$
4: **while** $Q \neq \emptyset$ **do**
5: $u \leftarrow \textsc{Extract-Min}(Q)$ {smallest f-value}
6: **if** $u = t$ **then**
7: **return** $g[t]$ {found shortest path!}
8: **end if**
9: Add u to *closed*
10: **for** each neighbor v of u not in *closed* **do**
11: **if** $g[u] + w(u, v) < g[v]$ **then**
12: $g[v] \leftarrow g[u] + w(u, v)$; update v in Q with key $f(v) = g[v] + h(v)$
13: **end if**
14: **end for**
15: **end while**

A* — Worked Example

Shortest path from S to T on a grid (walls block movement, edges cost 1):



Heuristic: Manhattan distance. A* finds $S \to b \to d \to e \to T$ (cost 4) visiting only **5 vertices**. Dijkstra would also visit a and c.

A* vs. Dijkstra

	Dijkstra	A*
Priority key	$g(v)$	$f(v) = g(v) + h(v)$
Heuristic	None ($h \equiv 0$)	Problem-specific
Explores	All reachable vertices	Goal-directed
Optimality	Always (non-neg. weights)	If h admissible
Neg. weights	No	No
Best for	All-destinations SSSP	Point-to-point SSSP

Key Idea

A* is **optimally efficient**: among algorithms that use the same heuristic, no other optimal algorithm expands fewer vertices than A*.

In practice: A* is the gold standard for point-to-point pathfinding in games, robotics, and navigation systems.

Part 3

All-Pairs Shortest Paths

The Problem

All-Pairs Shortest Paths (APSP)

Given a weighted directed graph $G = (V, E, w)$ with $n = |V|$, compute $\delta(u, v)$ for **every** pair $(u, v) \in V \times V$.

Output: an $n \times n$ matrix D where $D[i][j] = \delta(i, j)$.

Naive approach: Run Dijkstra (or Bellman-Ford) from every vertex.

- Dijkstra $\times V$: $\mathcal{O}(V \cdot E \log V)$ — good for sparse, non-negative graphs
- Bellman-Ford $\times V$: $\mathcal{O}(V^2 E)$ — handles negative weights but slow

Can we do better? **Yes!**

- Floyd-Warshall: $\Theta(V^3)$, simple, handles negative weights
- Johnson's algorithm: $\mathcal{O}(V^2 \log V + VE)$, best for sparse + negative weights

Floyd-Warshall — The Idea

Key Idea

Use **dynamic programming** on *intermediate vertices*.

$d_{ij}^{(k)}$ = weight of the shortest path from i to j using only vertices $\{1, 2, \ldots, k\}$ as intermediates.

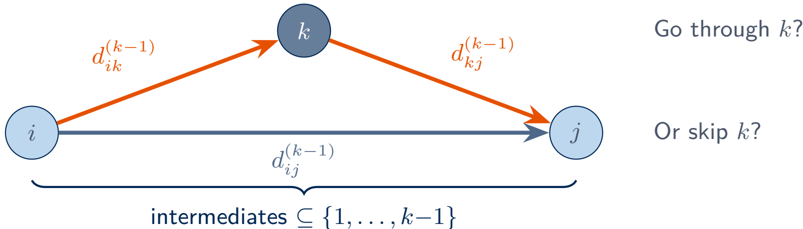
Recurrence:

$$d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)},\ d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \right)$$

Base case: $d_{ij}^{(0)} = w(i,j)$ (direct edge weight, 0 if $i = j$, ∞ otherwise).

Answer: $\delta(i,j) = d_{ij}^{(n)}$ after considering *all* vertices as intermediates.

Floyd-Warshall — Visualizing the Recurrence



Go through k?

Or skip k?

At each step k, we decide: go through vertex k, or not?

Floyd-Warshall — Pseudocode

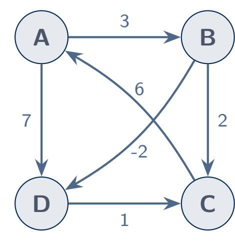
Input: Weighted adjacency matrix $W[1..n][1..n]$
Output: Distance matrix $D[1..n][1..n]$

```
1:  D ← W {D[i][j] = w(i, j); D[i][i] = 0}
2:  for k = 1 to n do
3:      for i = 1 to n do
4:          for j = 1 to n do
5:              if D[i][k] + D[k][j] < D[i][j] then
6:                  D[i][j] ← D[i][k] + D[k][j]
7:              end if
8:          end for
9:      end for
10: end for
11: return D
```

Complexity: $\Theta(V^3)$ time, $\Theta(V^2)$ space

Works with **negative weights**, but *not* negative cycles (detectable: $D[i][i] < 0$).

Floyd-Warshall — Worked Example (Setup)



Vertex order: $1{=}A,\ 2{=}B,\ 3{=}C,\ 4{=}D$.

$D^{(0)}$ — direct edges only (no intermediates):

$$\begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{pmatrix} 0 & 3 & \infty & 7 \\ \infty & 0 & 2 & -2 \\ 6 & \infty & 0 & \infty \\ \infty & \infty & 1 & 0 \end{pmatrix} \end{array}$$

$D^{(1)}$ — allow A as intermediate:
$C \to A \to B = 6{+}3 = 9, \quad C \to A \to D = 6{+}7 = 13.$

$$\begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{pmatrix} 0 & 3 & \infty & 7 \\ \infty & 0 & 2 & -2 \\ 6 & \mathbf{9} & 0 & \mathbf{13} \\ \infty & \infty & 1 & 0 \end{pmatrix} \end{array}$$

Floyd-Warshall — Worked Example (Iterations)

$D^{(2)}$ — allow $\{A, B\}$:
$A{\to}B{\to}C = 3{+}2 = 5$; $A{\to}B{\to}D = 3{+}(-2) = 1$;
$C{\to}B{\to}D = 9{+}(-2) = 7$.

$$\begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{pmatrix} 0 & 3 & \mathbf{5} & \mathbf{1} \\ \infty & 0 & 2 & -2 \\ 6 & 9 & 0 & \mathbf{7} \\ \infty & \infty & 1 & 0 \end{pmatrix} \end{array}$$

$D^{(3)}$ — allow $\{A, B, C\}$:
$B{\to}C{\to}A = 2{+}6 = 8$; $D{\to}C{\to}A = 1{+}6 = 7$;
$D{\to}C{\to}B = 1{+}9 = 10$.

$$\begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{pmatrix} 0 & 3 & 5 & 1 \\ \mathbf{8} & 0 & 2 & -2 \\ 6 & 9 & 0 & 7 \\ \mathbf{7} & \mathbf{10} & 1 & 0 \end{pmatrix} \end{array}$$

$D^{(4)}$ — allow $\{A, B, C, D\}$ (final):
$A{\to}D{\to}C = 1{+}1 = 2$ (improves 5);
$B{\to}D{\to}A = -2{+}7 = 5$ (improves 8);
$B{\to}D{\to}C = -2{+}1 = -1$ (improves 2).

$$\begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{pmatrix} 0 & 3 & \mathbf{2} & 1 \\ \mathbf{5} & 0 & \mathbf{-1} & -2 \\ 6 & 9 & 0 & 7 \\ 7 & 10 & 1 & 0 \end{pmatrix} \end{array}$$

Example

Check: $\delta(B, C) = -1$ via $B \to D \to C$ $(-2 + 1)$; $\delta(A, D) = 1$ via $A \to B \to D$ $(3 - 2)$; diagonal stays 0 ✔ (no negative cycle).

Transitive Closure

Transitive Closure

Given a directed graph G, compute $G^* = (V, E^*)$ where $(i,j) \in E^*$ iff there exists a path from i to j in G.

Adaptation of Floyd-Warshall using *boolean* operations:

$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \lor \left(t_{ik}^{(k-1)} \land t_{kj}^{(k-1)}\right)$$

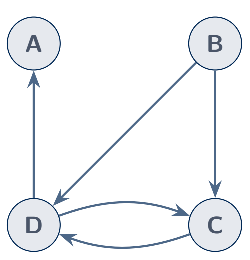
Same $\Theta(V^3)$ time, but uses `OR`/`AND` instead of $\min/+$.

Applications

- **Reachability queries** in databases
- **Dependency analysis** in build systems (can module A reach module B?)
- **Type systems**: subtyping / class hierarchy reachability

Transitive Closure — Worked Example

Edges: $D \to A$, $B \to D$, $D \to C$, $B \to C$, $C \to D$.



A has no outgoing edges, so it reaches no one.

$C \leftrightarrow D$ form a 2-cycle, so each reaches itself.

Initial $T^{(0)}$ (adjacency):

$$\begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix} \end{array}$$

Final $T^{(4)}$ (transitive closure):

$$\begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix} \end{array}$$

B reaches A via $B \to D \to A$; C reaches A via $C \to D \to A$ and itself via $C \to D \to C$; D reaches itself via $D \to C \to D$.

Johnson's Algorithm — Best of Both Worlds

Key Idea

Idea: **reweight** edges to make all weights non-negative, then run Dijkstra from every vertex.

Steps:

1. Add a new vertex s with zero-weight edges to all other vertices
2. Run **Bellman-Ford** from s to get $h(v) = \delta(s, v)$
3. Define new weights: $\hat{w}(u, v) = w(u, v) + h(u) - h(v) \geqslant 0$
4. Run **Dijkstra** from each vertex using $\hat{w}$
5. Recover original distances: $\delta(u, v) = \hat{\delta}(u, v) - h(u) + h(v)$

Complexity:
Bellman-Ford: $\mathcal{O}(VE)$
$V\times$ Dijkstra: $\mathcal{O}(V \cdot E \log V)$
Total: $\mathcal{O}(V^2 \log V + VE)$

Better than Floyd-Warshall for **sparse** graphs ($E \ll V^2$) with negative weights.

Which Algorithm to Choose?

Algorithm	Time	Neg. weights?	Best for
Floyd-Warshall	$\Theta(V^3)$	✔	Dense graphs, simplicity
Johnson's	$\mathcal{O}(V^2 \log V + VE)$	✔	Sparse + negative
$V\times$ Dijkstra	$\mathcal{O}(VE \log V)$	✘	Sparse, non-negative

Key Idea

Dense ($E \approx V^2$): Floyd-Warshall wins with simplicity.
Sparse ($E \approx V$): Johnson's is asymptotically superior.
No negative weights: just run Dijkstra V times.

Part 4

Maximum Flow

Flow Networks — Definitions

Flow Network

A **flow network** $G = (V, E)$ is a directed graph where:

- Each edge $(u, v) \in E$ has a **capacity** $c(u, v) \geqslant 0$
- There is a distinguished **source** s and **sink** t

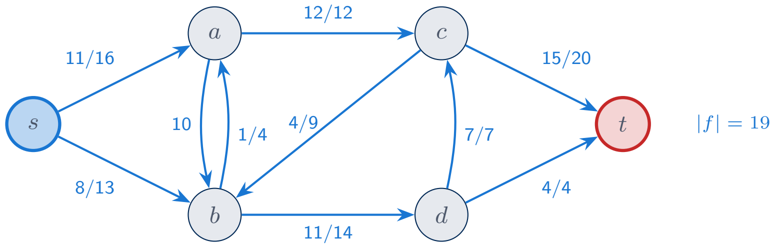
Flow

A **flow** is a function $f : V \times V \to \mathbb{R}$ satisfying:

1. **Capacity constraint:** $0 \leqslant f(u, v) \leqslant c(u, v)$ for all $(u, v) \in E$
2. **Symmetry:** $f(u, v) = -f(v, u)$ for all $u, v \in V$ (opposite direction flows cancel out)
3. **Flow conservation:** $\sum_{v} f(u, v) = 0$ for all $u \in V \setminus \{s, t\}$

The **value** $|f| = \sum_{v} f(s, v)$.

Flow Networks — Example



Labels show $f(u,v)/c(u,v)$. Flow conservation holds at every internal vertex.

Question: Is this the maximum flow? **No!** We can push more.

Residual Networks & Augmenting Paths

Residual Network

Given flow f, the **residual capacity**:

$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & \text{if } (u,v) \in E \\ 0 & \text{otherwise} \end{cases}$$

$G_f = (V, E_f)$ where $E_f = \{(u,v) : c_f(u,v) > 0\}$.

Augmenting Path

Consider a simple path p from s to t in the residual network G_f.

Its **residual capacity**:

$$c_f(p) = \min_{(u,v)\in p} c_f(u,v)$$

Key Idea

The residual network tells us *where we can still push flow* (forward edges) and *where we can cancel flow* (backward edges).

Ford-Fulkerson Method

Input: Flow network $G = (V, E)$ with source s, sink t
Output: Maximum flow f

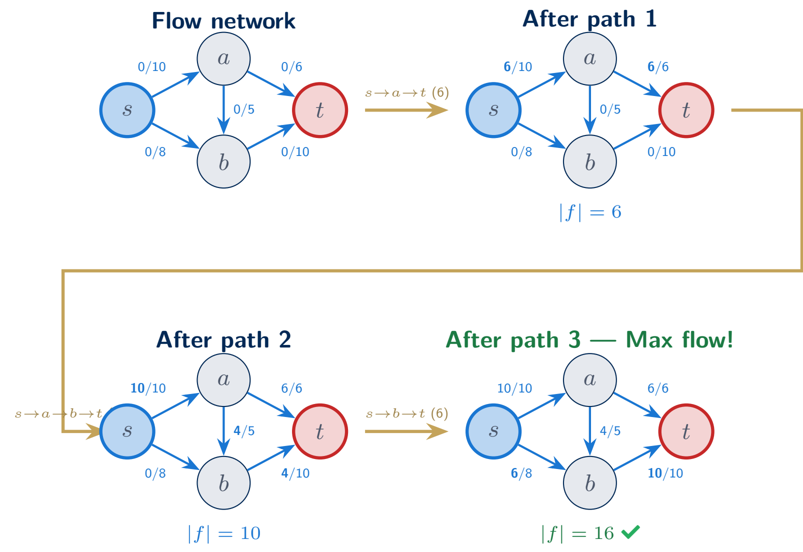
1: **for** each edge $(u, v) \in E$ **do**
2: $\quad f(u, v) \leftarrow 0$
3: **end for**
4: **while** $\exists$ path p from s to t in G_f (so that it can improve the flow) **do**
5: $\quad c_f(p) \leftarrow \min\{c_f(u, v) : (u, v) \in p\}$
6: $\quad$ **for** each edge (u, v) in p **do**
7: $\quad\quad f(u, v) \leftarrow f(u, v) + c_f(p)$
8: $\quad\quad f(v, u) \leftarrow -f(u, v)$ {preserve symmetry}
9: $\quad$ **end for**
10: **end while**
11: **return** f

Running time: $\mathcal{O}(E \cdot |f^*|)$ where $|f^*|$ is the max-flow value.

Warning

Without careful path selection, can be very slow!

Ford-Fulkerson — Step-by-Step Example



The Edmonds-Karp Algorithm

Key Idea

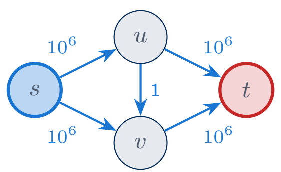
Use **BFS** (on the residual graph without capacities) to always find the *shortest* augmenting path (fewest edges). This guarantees polynomial time!

Key result: each edge can become "critical" at most $|V|/2$ times, so the total number of augmentations is $\mathcal{O}(VE)$.

Variant	Time	Note
Ford-Fulkerson (DFS)	$\mathcal{O}(E \cdot |f^*|)$	Depends on flow value
Edmonds-Karp (BFS)	$\mathcal{O}(VE^2)$	**Always polynomial**
Push-Relabel	$\mathcal{O}(V^3)$	Best for dense graphs

For this course, we focus on Ford-Fulkerson & Edmonds-Karp.

Why BFS Matters — A Pathological Case



Max flow is clearly $2{\cdot}10^6$ (cut around s).

Adversarial DFS choice — alternates through the tiny middle edge:

1. Augment $s \to u \to v \to t$: bottleneck $= \min(10^6, 1, 10^6) = \mathbf{1}$.
   Residual: $u{\to}v$ saturated; reverse edge $v{\to}u$ of capacity 1 appears.
2. Augment $s \to v \to u \to t$ *(via the residual $v{\to}u$)*: bottleneck $= \mathbf{1}$.
   Residual: $v{\to}u$ used up; $u{\to}v$ reappears with capacity 1.
3. Repeat 1–2: each pair of augmentations adds only 2 to $|f|$.

$\Rightarrow 2 \cdot 10^6$ **iterations**, each scanning $\Theta(V{+}E)$ edges.

BFS (Edmonds–Karp) picks *shortest* augmenting paths first: $s{\to}u{\to}t$ (length 2, bottleneck 10^6), then $s{\to}v{\to}t$. $\Rightarrow$ **2 iterations only.**

Plain Ford–Fulkerson runtime $\mathcal{O}(E \cdot |f^*|)$ depends on the *flow value* — exponential in the input size if capacities are huge. BFS achieves $\mathcal{O}(VE^2)$, independent of capacities.

Cuts & the Max-Flow Min-Cut Theorem

Cut of a Flow Network

A **cut** (S, T) partitions V into S and $T = V \setminus S$ with $s \in S$ and $t \in T$.

Capacity: $c(S,T) = \sum_{u \in S} \sum_{v \in T} c(u,v)$ Net flow: $f(S,T) = \sum_{u \in S} \sum_{v \in T} f(u,v)$

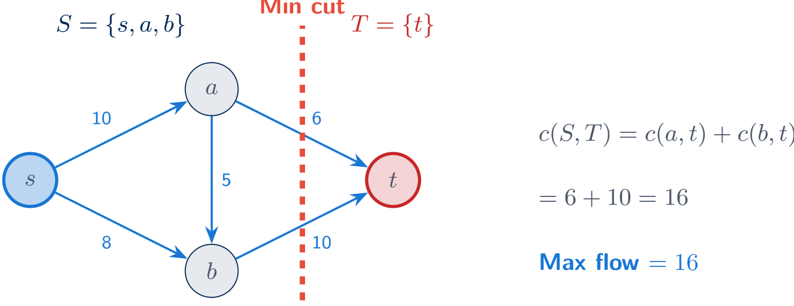
Max-Flow Min-Cut Theorem (Ford & Fulkerson, 1956)

If f is a flow in a flow network $G = (V, E)$ with source s and sink t, then the following conditions are equivalent.

1. f is a **maximum flow** in G
2. The residual network G_f has **no augmenting path**
3. $|f| = c(S, T)$ for some cut (S, T)

In particular: $\max |f| = \min_{(S,T)} c(S, T)$.

Visualizing a Minimum Cut



$c(S,T) = c(a,t) + c(b,t)$

$= 6 + 10 = 16$

Max flow $= 16$

The min cut identifies the **bottleneck** of the network.

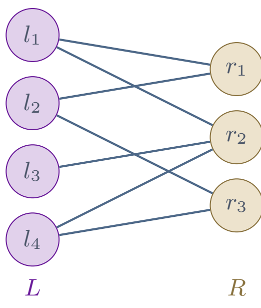
Part 5

Bipartite Graphs & Maximum Matching

Bipartite Graphs — Definition

Bipartite Graph

An undirected graph $G = (V, E)$ is **bipartite** if V can be partitioned into two disjoint sets L and R such that every edge connects a vertex in L to a vertex in R.



Real-world bipartite graphs:

- Candidates $\leftrightarrow$ time slots
- Interns $\leftrightarrow$ teams
- Students $\leftrightarrow$ projects
- Servers $\leftrightarrow$ tasks
- Donors $\leftrightarrow$ recipients

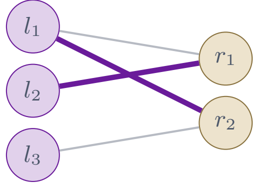
Maximum Matching

Matching

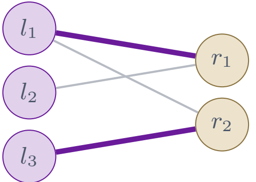
A **matching** $M \subseteq E$ is a set of edges with no shared endpoints: each vertex is incident to *at most one* edge of M.

A **maximum matching** maximizes $|M|$.

A matching of size 2



Maximum matching of size 2



Interview Scheduling

Candidates $= L$, **Time slots** $= R$, edge $=$ "candidate is available at that slot". Maximum matching $=$ schedule that interviews as many candidates as possible.

Reducing Matching to Maximum Flow

Key Idea

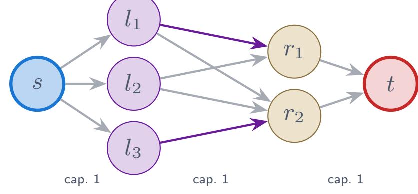
Transform the bipartite graph into a flow network, run max flow, and read off the matching!

Construction:

1. Add source s with edges to all $l \in L$ (capacity 1)
2. Direct all edges from L to R (capacity 1)
3. Add edges from all $r \in R$ to sink t (capacity 1)

Result: Maximum flow value = maximum matching size.

Edges $L \to R$ with $f = 1$ form the matching.



Why Does This Work?

Unit capacities $\Rightarrow$ integer flow

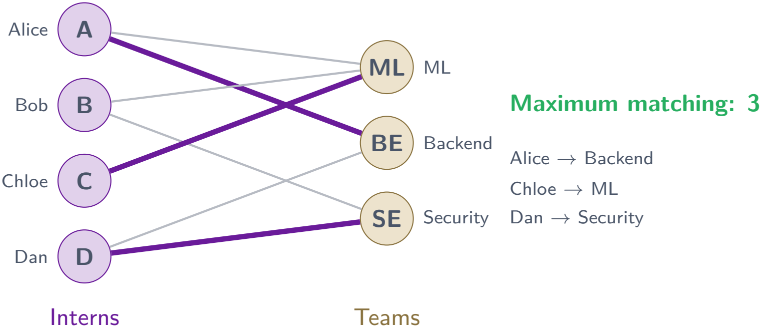
- All capacities are 1
- Ford-Fulkerson preserves integrality (Integrality Theorem)
- So every $f(u, v) \in \{0, 1\}$
- Each vertex in L sends at most 1 unit $\Rightarrow$ matched to $\leqslant 1$ vertex in R
- Each vertex in R receives at most 1 unit $\Rightarrow$ matched to $\leqslant 1$ vertex in L

Integrality Theorem

If all capacities are integers, then the Ford-Fulkerson method produces an **integer-valued** maximum flow.

Complexity:
$|f^*| \leqslant \min(|L|, |R|) = \mathcal{O}(V)$
$|E'| = \Theta(E)$
$\Rightarrow \mathcal{O}(VE)$ time

Matching — Worked Example (Intern Assignment)



Bob is **unmatched** — both his preferred teams are already taken.
3 out of 4 interns are assigned: this is optimal.

Part 6

Summary & What's Next

Algorithm Comparison

Problem	Algorithm	Time	Key idea
SSSP (point-to-point)	A*	$\mathcal{O}((V+E)\log V)$	Heuristic-guided
APSP	Floyd-Warshall	$\Theta(V^3)$	DP on intermediates
APSP	Johnson's	$\mathcal{O}(V^2 \log V + VE)$	Reweighting
Max Flow	Ford-Fulkerson	$\mathcal{O}(E\lvert f^*\rvert)$	Augmenting paths
Max Flow	Edmonds-Karp	$\mathcal{O}(VE^2)$	BFS shortest paths
Bipartite Matching	via Max Flow	$\mathcal{O}(VE)$	Reduction

Key Takeaways

1. **A*** extends Dijkstra with a heuristic for efficient *point-to-point* shortest paths; optimal with admissible heuristics
2. **Floyd-Warshall** is the swiss-army knife for APSP: simple $\Theta(V^3)$ DP, handles negative weights
3. **Johnson's algorithm** combines Bellman-Ford + Dijkstra via clever *reweighting* for sparse graphs
4. **Max flow** is solved by repeatedly finding and saturating *augmenting paths* in the residual graph
5. The **max-flow min-cut theorem** connects two seemingly different quantities: maximum flow = minimum cut capacity
6. **BFS** path selection (Edmonds-Karp) is critical for polynomial-time guarantees
7. Many combinatorial problems **reduce to max flow**: bipartite matching, edge connectivity, scheduling, ...

Thank you! Questions?

felix.chavelli@inria.fr