ADS2 — Week “Network 2” (Slides-first) — Full Notes & Worked Solutions

Field	Value
Title	ADS2 Network Flow II — Weekplan “Network 2”
Date	2025-10-18
Author	ADS2 Notes Copilot
Sources used	**weekplan.pdf** (pp.1–2); **slides-01…15.png** (Network Flow II deck); **ex_7_8.pdf** (KT 7.8, pp.418–419); **ex_7_14.pdf** (KT 7.14, p.421); exercise figures **Pasted image.png**, **Pasted image (2).png**, **Pasted image (3).png**, **Pasted image (4).png**
Week plan filename	weekplan.pdf

General Methodology and Theory

- **Max-flow / Min-cut.** Residual network, augmenting paths; **Edmonds–Karp (EK)** uses BFS in the residual graph → $O(V \cdot E^2)$. **Capacity scaling** augments only on residual edges with capacity $\geq \Delta$; start Δ = highest power of 2 $\leq$ max out of *s*; halve Δ when no Δ-eligible path exists → $O(E^2 \log C)$. (Slides.)
- **Matching and b-matching by flow.** s→Left (row/agents) with row quotas, unit edges to Right (columns/tasks), Right→t with column quotas. Hopcroft–Karp for unit matching $O(E\sqrt{V})$. (Slides.)
- **Disjoint paths and connectivity.** Unit-capacity edges: max number of edge-disjoint s–t paths equals min size of an s–t cut (Menger). (Slides.)
- **Node capacities via splitting.** Replace v by v_in→v_out of capacity c(v); redirect incident edges accordingly. (Slides; used in KT 7.14b.)
- **Certificates.** When done, show a cut (S,T) whose crossing capacity equals $|f|$; for infeasibility explain the bottleneck in plain language.

Notes (slide highlights you’ll reuse)

- **EK path rule:** forward edges with leftover cap, backward edges with positive flow; bottleneck δ is min residual along the path. Tie-break lexicographically when listing neighbors.
- **Scaling phases:** Δ sequence …, 8, 4, 2, 1. Within a phase, each augmentation adds $\geq\Delta$; $\leq 2m$ augmentations per phase; overall $O(m^2 \log C)$. (Slides 37–39.)
- **Bipartite matching via flow:** unit edges, value of flow = size of matching. (Slides 41–47.)
- **Edge-disjoint paths / connectivity:** set all caps to 1; run max-flow / min-cut; dotted edges in slides mark a min cut. (Slides 52–56.)

Coverage Table (enumerated strictly from weekplan.pdf)

Weekplan ID	Canonical ID	Title/Label (verbatim)	Assignment Source	Text Source	Status
1	—	The Edmonds-Karp algorithm and the scaling algorithm	weekplan.pdf p.1	figure images: **Pasted image.png** (two graphs) + slides (EK/scaling)	Solved
2	KT 7.8	Blood Donations	weekplan.pdf p.1	ex_7_8.pdf pp. 418–419 + **Pasted image (4).png** (table)	Solved
3	—	Christmas Trees (from the Exam E15)	weekplan.pdf p.1	weekplan text + **Pasted image (2).png** (example grid)	Solved
4	KT 7.14	Escape	weekplan.pdf p.1	ex_7_14.pdf p.421	Solved
5	CSES 1696	School dance	weekplan.pdf p.2	external statement (not uploaded); solved by standard matching mapping	Solved (method)
6	—	[*] Euler tours in mixed graphs	weekplan.pdf p.2	weekplan text + **Pasted image (3).png** (example pair)	Solved

Solutions

Exercise 1 — EK & Capacity Scaling (two graphs)

Concept mapping. Compute max flow and a min cut on each graph; show the **augmenting paths with δ**. Use EK (BFS) and then repeat with **scaling** (Δ phases 4→2→1 etc.).

Left graph (nodes s, L, F, C, B, M, G, A, t; capacities as in **Pasted image.png**).

Mini augmentation trace (EK):

Step	s→t path (BFS in residual)	δ (delta)	Saturated edges	Residual note
1	s→C→G→t	2	s→C, G→t	add C→s, t→G of 2

Step	s→t path (BFS in residual)	δ (delta)	Saturated edges	Residual note
2	s→L→F→A→t	3	F→A	add A→F of 3
3	s→L→F→G→A→t	3	F→G	add G→F of 3
4	s→L→F→G→C→B→M→t	1	M→t	add t→M of 1

Result |f| = **9**. **Cut certificate:** S = {s, L, F}; crossing edges: s→C(2), F→A(3), F→G(4) → sum **9**, hence max flow = 9.

Scaling (Δ = 4,2,1) gives the same value with fewer within-phase paths (slides 13–21 illustrate the mechanics).

Right graph (A,B,C,D,E,F,G,H with s,t; **Pasted image.png**). EK trace (one valid BFS order):

Step	s→t path	δ	Saturated edges
1	s→t	2	s→t
2	s→D→E→F→t	3	D→E, E→F
3	s→D→E→G→H→t	1	D→E fully
4	s→D→A→B→C→F→t	2	F→t fully
5	s→D→A→B→C→F⇢E→G→H→t	3	A→B fully

Result |f| = **11**. **Cut certificate:** S = {s, D, A, B}; crossing edges: s→t(2) + D→E(4) + B→C(5) = **11**.

Pitfalls. Forgetting backward edges; in scaling: treating it as “fattest path” instead of Δ-filtered BFS.

Variant drill. Reduce A→t on the left graph from 6→4: recompute to get |f| = **8**; same S but smaller capacity.

Transfer Pattern. *Archetype:* max s–t flow + min-cut certificate. *Cues:* “augmenting paths,” “EK,” “scaling.” *Mapping:* build residual, BFS, augment; then report (S,T) by reachability in residual. *Certificate:* list crossing edges and sum.

Exercise 2 — KT 7.8 Blood Donations

Model (slides-first). Build a flow network: - Source to donor-type nodes with capacities = supplies (O:50, A:36, B:11, AB:8). - Compatibility edges: O→{O,A,B,AB}; A→{A,AB}; B→{B,AB}; AB→{AB} (unit-capacity per unit of blood; practically cap = ∞ between compatible types). - Patient-type nodes to sink with capacities = demands (O:45, A:42, B:8, AB:3).

Computation (integer max-flow). One optimal allocation: - O→O: 44, O→A: 6; A→A: 36; B→B: 8; AB→AB: 3.
Total treated = **97** patients.

Why not 100? (cut & plain-English). Type A needs 42 but has only 36 → must borrow **6** from O. Then O has ≥50−6 = **44** left for its own 45 patients ⇒ at least **one O-patient** cannot be treated. This is a min-cut-style bottleneck and proves optimality (|f| = 97).

Pitfalls. Sending type A to AB while starving O; forgetting that O is the only universal donor.

Variant drill. If A-supply were 38, O would loan 4 and everyone could be served (**100**).

Transfer Pattern. *Archetype:* multi-commodity supply→demand via compatibility DAG; solved as max-flow. *Cues:* supplies/demands by class, compatibility table. *Mapping:* donors = left, patients = right; caps = supply/demand; run max-flow. *Certificate:* shortfall forces O-loans; count remaining O.

✅**Answer:** Maximum patients served **97**; exactly **1 O-patient** remains untreated.

Exercise 3 — Christmas Trees (Exam E15)

Goal. Place as many tables as possible on an n×m grid with **≤2 per row** and **≤1 per column**, forbidden at tree cells.

Flow model (b-matching). For each row R_i add s→R_i with cap 2; for each column C_j add C_j→t with cap 1; for each empty cell (i,j) add R_i→C_j (cap 1). Run max-flow.

Correctness. Feasible flows correspond to placements; |f| is the number of tables. Min-cut upper-bounds any placement.

Runtime. With Hopcroft–Karp on the induced bipartite graph: **$O(E\sqrt{V})$**; with EK: $O(V \cdot E^2)$.

Example check. The provided 4×8 instance (**Pasted image (2).png**) has optimum **7**.

Pitfalls. Greedy per-row or per-column can block future placements; forgetting to omit tree cells.

Variant drill. If a column j allows up to 2 tables, set cap(C_j→t)=2.

Transfer Pattern. *Archetype:* bipartite **b-matching** via flow. *Cues:* "≤2 per row, ≤1 per column," grid placement with forbidden cells. *Mapping:* rows⇄columns; unit cell edges; capacities encode row/column limits. *Certificate:* min cut selecting tight rows/columns.

✅**Answer:** Build the network above and return |f| (7 in the example grid).

Exercise 4 — KT 7.14 Escape (edge- and node-disjoint)

(a) Edge-disjoint routes. Add super-source s→*x (cap 1) for each x∈X; keep original edges with cap 1; connect each safe u∈S to t* with large cap (or 1 if "at most one per safe"). Run max-flow; feasible iff value = |X|.

(b) Node-disjoint routes. Split every vertex v into v_in→v_out with cap 1 (or c(v)); replace each (u,v) by u_out→v_in (cap 1). Keep s→*x_in and u_out→t* for u∈S. Run max-flow; feasible iff value = |X|.

Why it works. Integrality gives a set of disjoint paths from any unit max-flow; conversely, a family of k disjoint paths yields a flow of value k.

Complexity. Linear-time build; EK is fine here; Dinic/HK faster on unit graphs.

Pitfalls. Forgetting to set caps to 1; not splitting nodes for part (b).

Variant drill. If safe node u can accept c(u) evacuees, use cap v_in→v_out = c(u) for v=u.

Transfer Pattern. *Archetype:* disjoint paths via flow; *Cues:* "routes do not share edges/nodes." *Mapping:* unit edges + super-source/sink; node-split for node capacities. *Certificate:* value = |X| with path decomposition.

✅**Answer:** Build the respective networks; **YES** iff max-flow equals |X|.

Exercise 5 — CSES 1696 School Dance (method card)

- **I/O gist.** n boys, m girls, E allowed pairs; print maximum number of pairs and the pairs.
- **Solve:** Bipartite matching via Hopcroft–Karp (or max-flow). s→boys (1), allowed edges (1), girls→t (1). Extract matched edges.
- **Time:** $O(E\sqrt{(n+m)})$.
- **Pitfalls:** 1-based indices in output; don't print unmatched nodes.

Transfer Pattern. *Archetype:* unit bipartite matching. *Certificate:* matching size = flow value; edges in the matching.

✅**Answer:** Reduce to bipartite matching and run HK; output the matching.

Exercise 6 — [*] Euler tours in mixed graphs

Goal. Orient undirected edges so final directed graph has an **Euler tour**.

Balances. Let r(v)=in_dir(v)−out_dir(v) from pre-oriented edges; let udeg(v) be number of incident undirected edges. If we orient e=u—v as u→v, then r(u)−=1 and r(v)+=1. Each v must receive exactly **h(v)=(udeg(v)−r(v))/2** incoming orientations from its incident undirected edges. Feasible only if all h(v) are integers in [0, udeg(v)]. (Parity check.)

Reduction (slides hint ⇒ b-matching). Build bipartite graph **(E_u ↔ V)**: left nodes are undirected edges e, right nodes are vertices. Connect e to its two endpoints. Find a b-matching that matches each e exactly once and matches each vertex v exactly **h(v)** times. Flow build: - s→e (cap 1) for each undirected edge e; e→u and e→v (cap 1); v→t (cap h(v)). - Feasible flow of value |E_u| ⇒ choose the matched endpoint as the **head** of e (tail is the other endpoint). Combined with directed edges, every v now has in=out; connectivity (assumed when ignoring directions) gives an Euler tour.

Complexity. $O(E\sqrt{V})$ via b-matching flow.

Pitfalls. Skipping parity check; demanding strong connectivity (not needed); producing negative h(v).

Variant drill. Fix orientations of some undirected edges first, update r(·), re-run the test.

Transfer Pattern. *Archetype:* circulation with vertex demands via edge→vertex **b-matching**. *Certificate:* feasible b-matching of size |E_u|.

✅ **Answer: YES** iff all h(v) are integers in range and the b-matching flow returns value |E_u|; orientation is read off from the matching.

Puzzle — 99 Cops (<299 questions)

Plan. (i) Use the Boyer–Moore majority trick to find one honest cop in ≤98 questions by pair-cancelling suspects; (ii) ask the honest cop about all others (≤98 more). Total ≤196.
Why it works. Honest cops are a strict majority, so cancellation leaves an honest survivor; an honest witness then labels everyone correctly.

Summary (one-page refresher)

- **Algorithms used:** EK and scaling; Hopcroft–Karp / flow for matching and b-matching; node-splitting for node capacities.
- **Certificates to give:** min (S,T) cut with crossing sum = |f|; for Blood, the O-loan bottleneck ⇒ 97; for Trees, cut picks tight rows/columns; for Mixed-Euler, b-matching of size |E_u|.
- **Transfer cues:** "≤ per row/column" ⇒ b-matching; "routes don't share" ⇒ disjoint paths + node-split; "compatibility table" ⇒ supply→demand flow; "augmenting path trace" ⇒ EK/scaling.
- **Notation:** |f| flow value; δ bottleneck; r(v)=in−out on pre-directed part; h(v) incoming heads needed from undirected edges.

Network Flow II

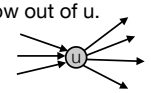
Inge Li Gørtz

KT 7.3, 7.5, 7.6

Network Flow

- Network flow:
  - graph G=(V,E).
  - Special vertices s (source) and t (sink).
  - Every edge e has a capacity c(e) ≥ 0.
  - Flow:
    - **capacity constraint:** every edge e has a flow 0 ≤ f(e) ≤ c(e).
    - **flow conservation:** for all u ≠ s, t: flow into u equals flow out of u.

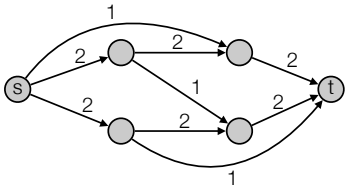
$$\sum_{v:(v,u)\in E} f(v,u) = \sum_{v:(u,v)\in E} f(u,v)$$



    - Value of flow f is the sum of flows out of s minus sum of flows into s:

$$v(f) = \sum_{v:(s,v)\in E} f(e) - \sum_{v:(v,s)\in E} f(e) = f^{out}(s) - f^{in}(s)$$

  - **Maximum flow problem:** find s-t flow of maximum value

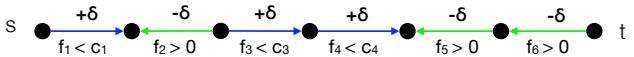


Today

- Applications
- Finding good augmenting paths. Edmonds-Karp and scaling algorithm.

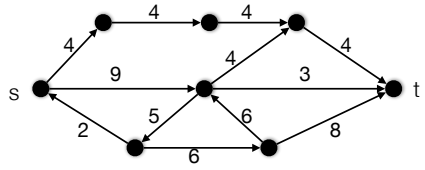
Ford-Fulkerson

- Find (any) augmenting path and use it.
- Augmenting path (definition different than in CLRS): s-t path where
  - forward edges have leftover capacity
  - backwards edges have positive flow



  - Can add extra flow: min($c_1 - f_1$, f_2, $c_3 - f_3$, $c_4 - f_4$, f_5, f_6) = δ

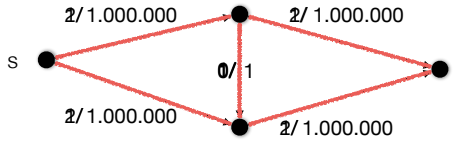
- To find augmenting path use DFS or BFS:



Ford-Fulkerson

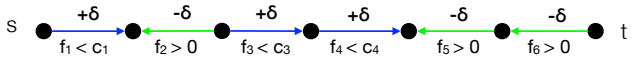
- Integral capacities:
  - Each augmenting path increases flow with at least 1.
  - At most $v(f)$ iterations
  - Find augmenting path via DFS/BFS: $O(m)$
  - Total running time: $O(m \cdot v(f))$
- Lemma. If all the capacities are integers, then there is a maximum flow where the flow on every edge is an integer.

- Bad example for Ford-Fulkerson:



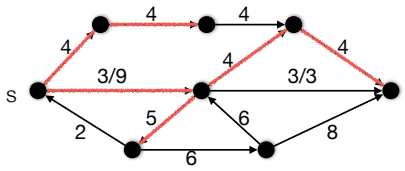
Edmonds-Karp

- Find *shortest* augmenting path and use it.
- Augmenting path (definition different than in CLRS): s-t path where
  - forward edges have leftover capacity
  - backwards edges have positive flow



  - Can add extra flow: min($c_1 - f_1$, f_2, $c_3 - f_3$, $c_4 - f_4$, f_5, f_6) = δ

- To find augmenting path use ***BFS***:



Edmonds-Karp

- Find *shortest* augmenting path and use it.
- Augmenting path (definition different than in CLRS): s-t path where
  - forward edges have leftover capacity
  - backwards edges have positive flow

  - Can add extra flow: min($c_1 - f_1$, f_2, $c_3 - f_3$, $c_4 - f_4$, f_5, f_6) = δ

- To find augmenting path use *BFS*:

Edmonds-Karp

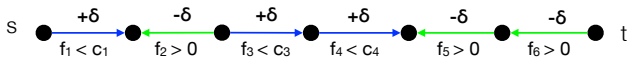
- Find *shortest* augmenting path and use it.
- Augmenting path (definition different than in CLRS): s-t path where
  - forward edges have leftover capacity
  - backwards edges have positive flow

  - Can add extra flow: min($c_1 - f_1$, f_2, $c_3 - f_3$, $c_4 - f_4$, f_5, f_6) = δ

- To find augmenting path use *BFS*:

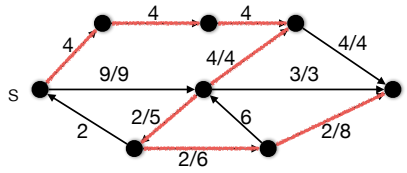
Edmonds-Karp

- Find *shortest* augmenting path and use it.
- Augmenting path (definition different than in CLRS): s-t path where
  - forward edges have leftover capacity
  - backwards edges have positive flow



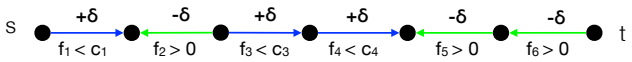
- Can add extra flow: $\min(c_1 - f_1, f_2, c_3 - f_3, c_4 - f_4, f_5, f_6) = \delta$

- To find augmenting path use *BFS*:



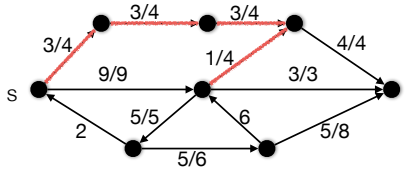
Edmonds-Karp

- Find *shortest* augmenting path and use it.
- Augmenting path (definition different than in CLRS): s-t path where
  - forward edges have leftover capacity
  - backwards edges have positive flow



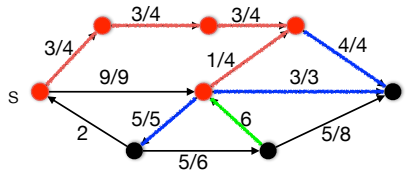
- Can add extra flow: $\min(c_1 - f_1, f_2, c_3 - f_3, c_4 - f_4, f_5, f_6) = \delta$

- To find augmenting path use *BFS*:



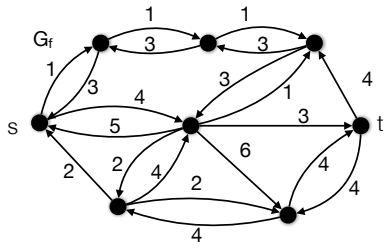
Find a minimum cut

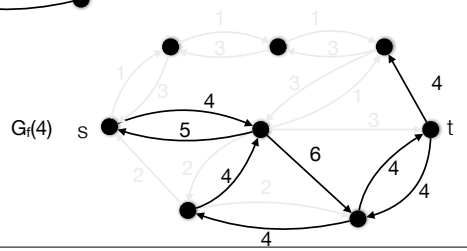
- When there are no more augmenting s-t paths:
- Find all augmenting paths from s.
- The nodes S that can be reached by these augmenting paths form the left side of a minimum cut.
  - edges out of S have $c_e = f_e$.
  - edges into S have $f_e = 0$.
  - Capacity of the cut equals the flow.



Scaling algorithm

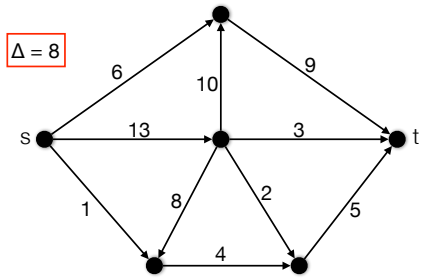
- Scaling parameter Δ
- Only consider edges with capacity at least Δ in residual graph $G_f(\Delta)$.
- Example: Δ = 4





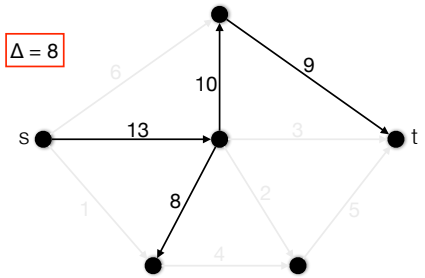
Scaling algorithm

- Scaling parameter Δ
- Only consider edges with capacity at least Δ in residual graph $G_f(\Delta)$.
- Start with Δ = “highest power of 2 ≤ largest capacity out of s”



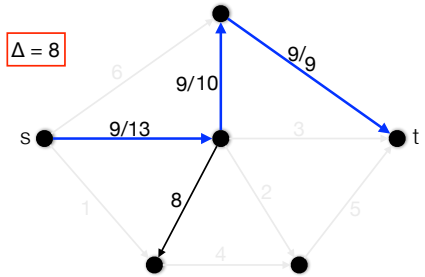
Scaling algorithm

- Scaling parameter Δ
- Only consider edges with capacity at least Δ in residual graph $G_f(\Delta)$.
- Start with Δ = “highest power of 2 ≤ largest capacity out of s”



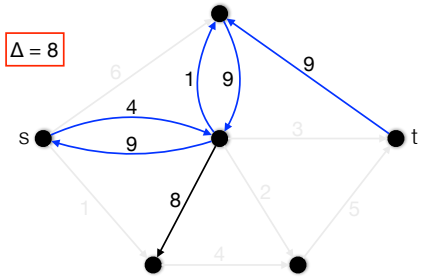
Scaling algorithm

- Scaling parameter Δ
- Only consider edges with capacity at least Δ in residual graph $G_f(\Delta)$.
- Start with Δ = “highest power of 2 ≤ largest capacity out of s”



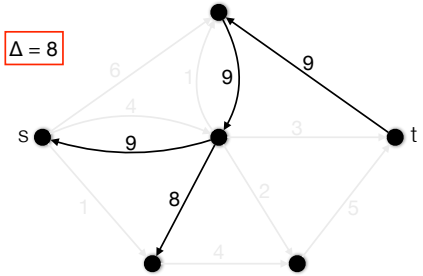
Scaling algorithm

- Scaling parameter Δ
- Only consider edges with capacity at least Δ in residual graph $G_f(\Delta)$.
- Start with Δ = “highest power of 2 ≤ largest capacity out of s”



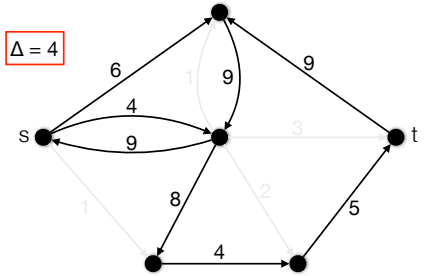
Scaling algorithm

- Scaling parameter Δ
- Only consider edges with capacity at least Δ in residual graph $G_f(\Delta)$.
- Start with Δ = “highest power of 2 ≤ largest capacity out of s”
- When no more augmenting paths in $G_f(\Delta)$: Δ = Δ/2 (new phase).



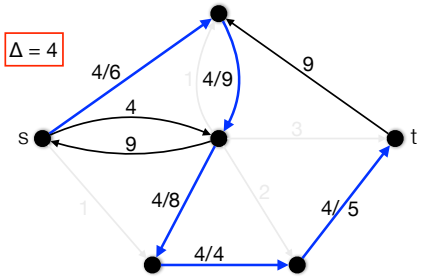
Scaling algorithm

- Scaling parameter Δ
- Only consider edges with capacity at least Δ in residual graph $G_f(\Delta)$.
- Start with Δ = “highest power of 2 ≤ largest capacity out of s”
- When no more augmenting paths in $G_f(\Delta)$: Δ = Δ/2 (new phase).



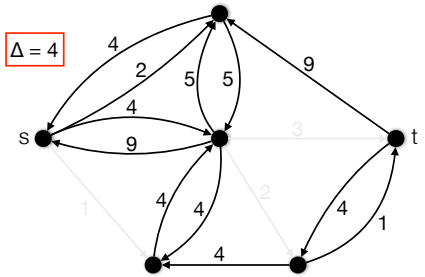
Scaling algorithm

- Scaling parameter Δ
- Only consider edges with capacity at least Δ in residual graph $G_f(\Delta)$.
- Start with Δ = “highest power of 2 ≤ largest capacity out of s”
- When no more augmenting paths in $G_f(\Delta)$: Δ = Δ/2 (new phase).



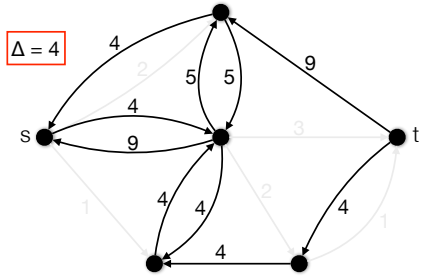
Scaling algorithm

- Scaling parameter Δ
- Only consider edges with capacity at least Δ in residual graph $G_f(\Delta)$.
- Start with Δ = “highest power of 2 ≤ largest capacity out of s”
- When no more augmenting paths in $G_f(\Delta)$: Δ = Δ/2 (new phase).



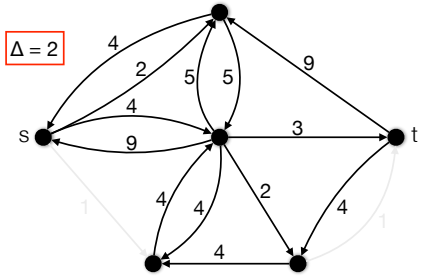
Scaling algorithm

- Scaling parameter Δ
- Only consider edges with capacity at least Δ in residual graph $G_f(\Delta)$.
- Start with Δ = “highest power of 2 ≤ largest capacity out of s”
- When no more augmenting paths in $G_f(\Delta)$: Δ = Δ/2 (new phase).



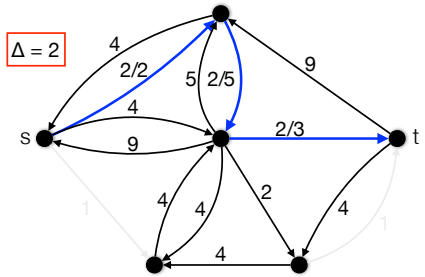
Scaling algorithm

- Scaling parameter Δ
- Only consider edges with capacity at least Δ in residual graph $G_f(\Delta)$.
- Start with Δ = “highest power of 2 ≤ largest capacity out of s”
- When no more augmenting paths in $G_f(\Delta)$: Δ = Δ/2 (new phase).



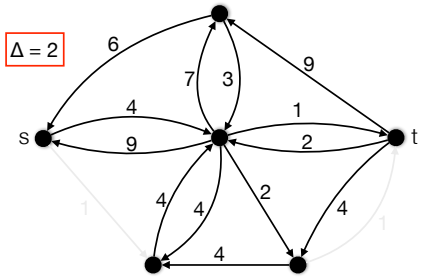
Scaling algorithm

- Scaling parameter Δ
- Only consider edges with capacity at least Δ in residual graph $G_f(\Delta)$.
- Start with Δ = “highest power of 2 ≤ largest capacity out of s”
- When no more augmenting paths in $G_f(\Delta)$: Δ = Δ/2 (new phase).



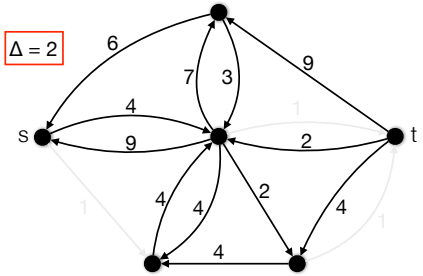
Scaling algorithm

- Scaling parameter Δ
- Only consider edges with capacity at least Δ in residual graph $G_f(\Delta)$.
- Start with Δ = “highest power of 2 ≤ largest capacity out of s”
- When no more augmenting paths in $G_f(\Delta)$: Δ = Δ/2 (new phase).



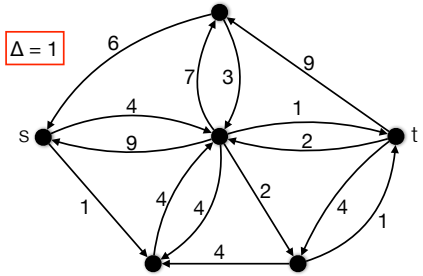
Scaling algorithm

- Scaling parameter Δ
- Only consider edges with capacity at least Δ in residual graph $G_f(\Delta)$.
- Start with Δ = “highest power of 2 ≤ largest capacity out of s”
- When no more augmenting paths in $G_f(\Delta)$: Δ = Δ/2 (new phase).



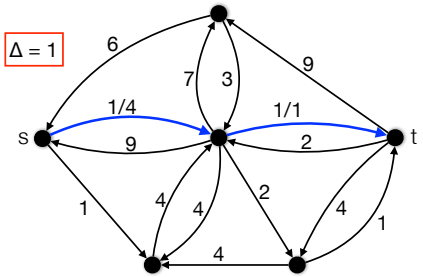
Scaling algorithm

- Scaling parameter Δ
- Only consider edges with capacity at least Δ in residual graph $G_f(\Delta)$.
- Start with Δ = “highest power of 2 ≤ largest capacity out of s”
- When no more augmenting paths in $G_f(\Delta)$: Δ = Δ/2 (new phase).



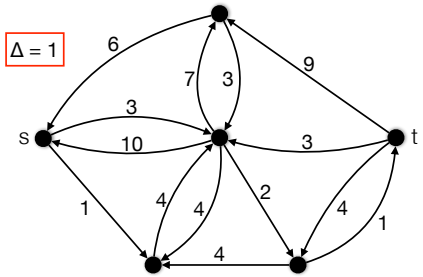
Scaling algorithm

- Scaling parameter Δ
- Only consider edges with capacity at least Δ in residual graph $G_f(\Delta)$.
- Start with Δ = “highest power of 2 ≤ largest capacity out of s”
- When no more augmenting paths in $G_f(\Delta)$: Δ = Δ/2 (new phase).



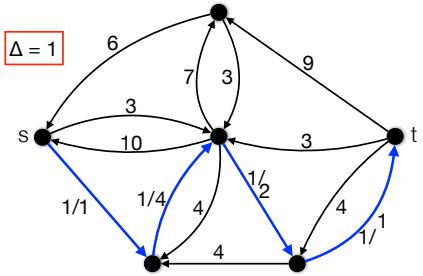
Scaling algorithm

- Scaling parameter Δ
- Only consider edges with capacity at least Δ in residual graph $G_f(\Delta)$.
- Start with Δ = “highest power of 2 ≤ largest capacity out of s”
- When no more augmenting paths in $G_f(\Delta)$: Δ = Δ/2 (new phase).



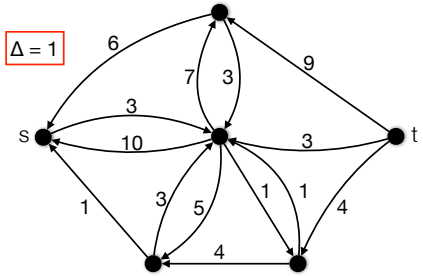
Scaling algorithm

- Scaling parameter Δ
- Only consider edges with capacity at least Δ in residual graph $G_f(\Delta)$.
- Start with Δ = "highest power of 2 ≤ largest capacity out of s"
- When no more augmenting paths in $G_f(\Delta)$: Δ = Δ/2 (new phase).



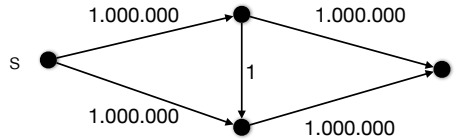
Scaling algorithm

- Scaling parameter Δ
- Only consider edges with capacity at least Δ in residual graph $G_f(\Delta)$.
- Start with Δ = "highest power of 2 ≤ largest capacity out of s"
- When no more augmenting paths in $G_f(\Delta)$: Δ = Δ/2 (new phase).

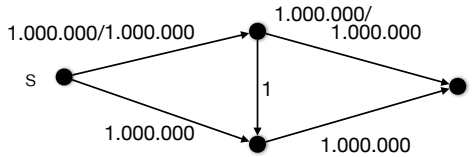


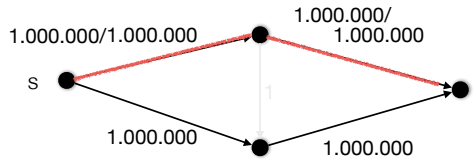
- Stop when no more augmenting paths in $G_f(1)$.

Scaling algorithm

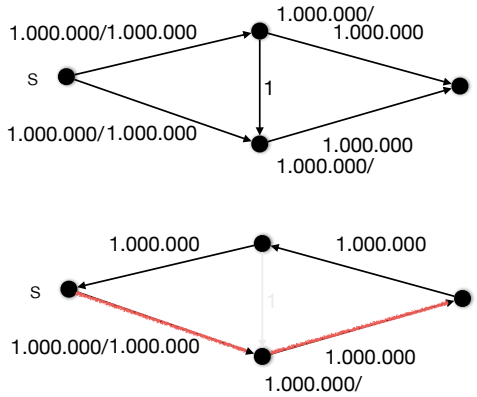


Scaling algorithm

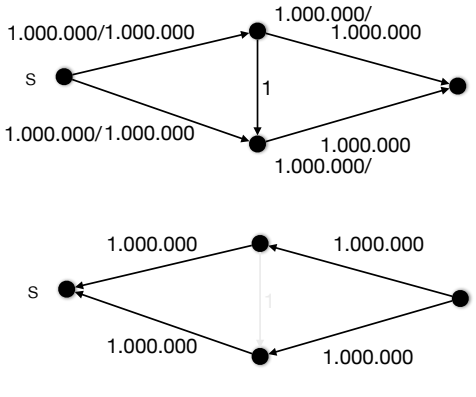




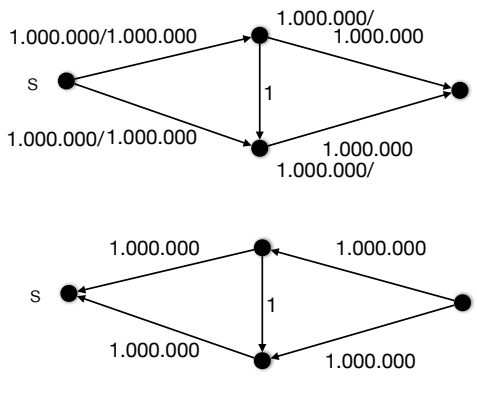
Scaling algorithm



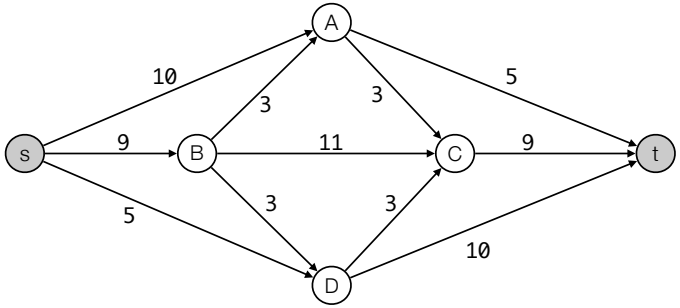
Scaling algorithm



Scaling algorithm



Exercise

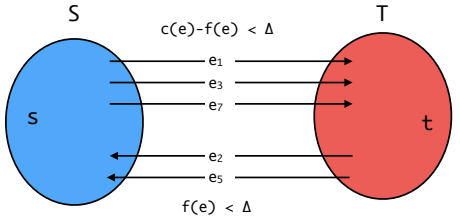


Scaling algorithm

- Running time: O(m^2 log C), where C is the largest capacity out of s.
- Lemma 1. *Number of scaling phases: 1 + ⌈lg C⌉*
- Prove: number of augmentation paths found in a scaling phase is at most 2m.
- Lemma 2. *Let f the flow when Δ-scaling phase ends, and let f* be the maximum flow. Then v(f*) ≤ v(f) + mΔ.*

Scaling algorithm

- Lemma 2. *Let f the flow when Δ-scaling phase ends, and let f*be the maximum flow. Then v(f*) ≤ v(f) + mΔ.*
- By the end of the phase there is a cut c(S,T) ≤ v(f) + mΔ.



$$c(S,T) = c(e_1) + c(e_3) + c(e_7)$$

$$v(f) = f(e_1) + f(e_3) + f(e_7) - f(e_2) - f(e_5)$$

$$\begin{aligned} c(S,T) - v(f) &= c(e_1) + c(e_3) + c(e_7) - f(e_1) - f(e_3) - f(e_7) + f(e_2) + f(e_5) \\ &= c(e_1) - f(e_1) + c(e_3) - f(e_3) + c(e_7) - f(e_7) + f(e_2) + f(e_5) \\ &< \Delta + \Delta + \Delta + \Delta + \Delta = 5\Delta \end{aligned}$$

Scaling algorithm

- Running time: O(m^2 log C), where C is the largest capacity out of s.
- Lemma 1. *Number of scaling phases: 1 + ⌈lg C⌉*
- Lemma 2. *Let f the flow when Δ-scaling phase ends, and let f* be the maximum flow. Then v(f*) ≤ v(f) + mΔ.*
- Lemma 3. *The number of augmentations in a scaling phase is at most 2m.*
  - First phase: can use each edge out of s in at most one augmenting path.
  - f flow at the end of previous phase.
  - Used Δ' = 2Δ in last round.
  - Lemma 2: v(f*) ≤ v(f) + mΔ' = v(f) + 2mΔ.
  - "Leftover flow" to be found ≤ 2mΔ.
  - Each agumentation in a Δ-scaling phase augments flow with at least Δ.

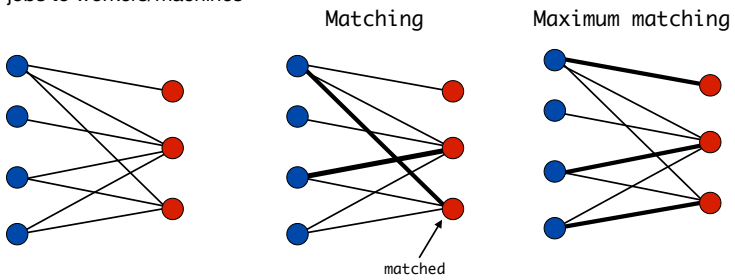
Maximum flow algorithms

- Edmonds-Karp: O(m^2n)
- Scaling: O(m^2 log C)
- Ford-Fulkerson: O(m v(f)).
- Preflow-push O(n^3)
- Other algorithms: O(mn log n) or O(min($n^{2/3}$, $m^{1/2}$)m log n log U).

Maximum Bipartite Matching

- Bipartite graph: Can color vertices red and blue such that all edges have a red and a blue endpoint.
- Matching: Subset of edges M ⊆ E such that no edges in M share an endpoint.
- Maximum matching: matching of maximum cardinality.

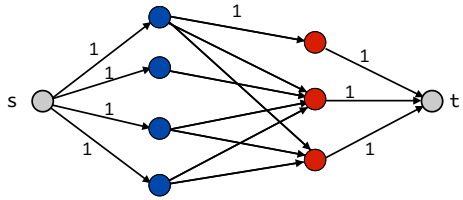
- Applications:
  - planes to routes
  - jobs to workers/machines



Maximum Bipartite Matching

- Bipartite graph: Can color vertices red and blue such that all edges have a red and a blue endpoint.
- Matching: Subset of edges M ⊆ E such that no edges in M share an endpoint.
- Maximum matching: matching of maximum cardinality.

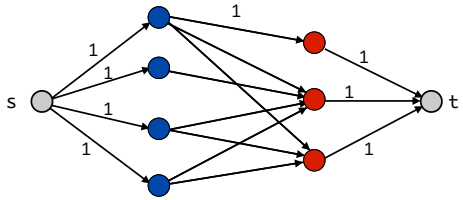
- Solve via flow:



Maximum Bipartite Matching

- Bipartite graph: Can color vertices red and blue such that all edges have a red and a blue endpoint.
- Matching: Subset of edges M ⊆ E such that no edges in M share an endpoint.
- Maximum matching: matching of maximum cardinality.

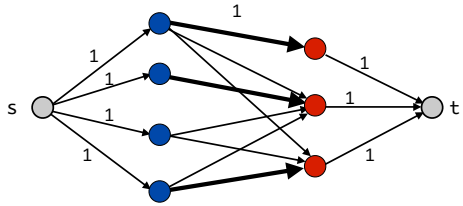
- Solve via flow:
  - Matching M => flow of value |M|



Maximum Bipartite Matching

- Bipartite graph: Can color vertices red and blue such that all edges have a red and a blue endpoint.
- Matching: Subset of edges M ⊆ E such that no edges in M share an endpoint.
- Maximum matching: matching of maximum cardinality.

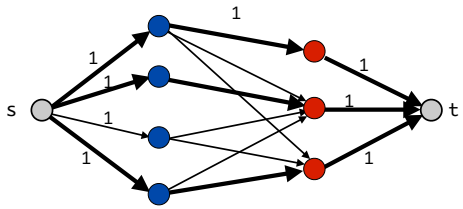
- Solve via flow:
  - Matching M => flow of value |M|



Maximum Bipartite Matching

- Bipartite graph: Can color vertices red and blue such that all edges have a red and a blue endpoint.
- Matching: Subset of edges M ⊆ E such that no edges in M share an endpoint.
- Maximum matching: matching of maximum cardinality.

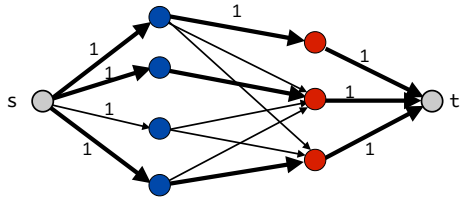
- Solve via flow:
  - Matching M => flow of value |M|



Maximum Bipartite Matching

- Bipartite graph: Can color vertices red and blue such that all edges have a red and a blue endpoint.
- Matching: Subset of edges M ⊆ E such that no edges in M share an endpoint.
- Maximum matching: matching of maximum cardinality.

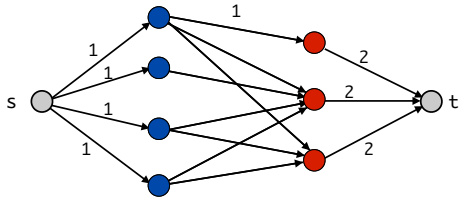
- Solve via flow:
  - Matching M => flow of value |M|
  - Flow of value v(f) => matching of size v(f)



Maximum Bipartite Matching

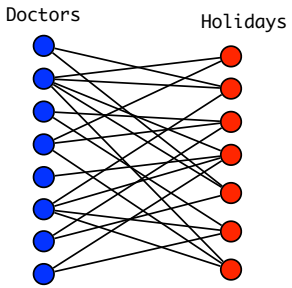
- Bipartite graph: Can color vertices red and blue such that all edges have a red and a blue endpoint.
- Matching: Subset of edges M ⊆ E such that no edges in M share an endpoint.
- Maximum matching: matching of maximum cardinality.

- Solve via flow:
- Can generalize to general matchings



Scheduling of doctors

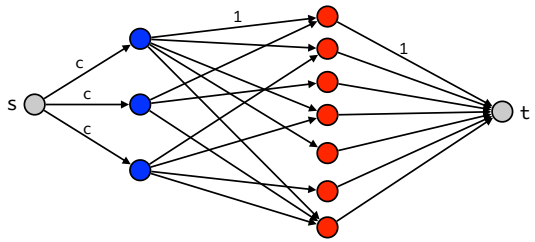
- X doctors, Y holidays, each doctor should work at at most 1 holiday, each doctor is available at some of the holidays.



- Same problem, but each doctor should work at most c holidays?

Scheduling of doctors

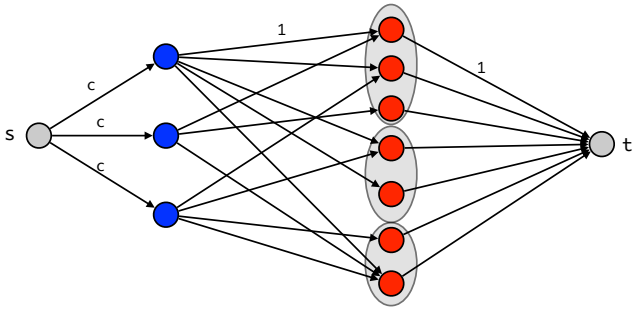
- X doctors, Y holidays, each doctor should work at at most c holidays, each doctor is available at some of the holidays.



- Same problem, but each doctor should work at most one day in each vacation period?

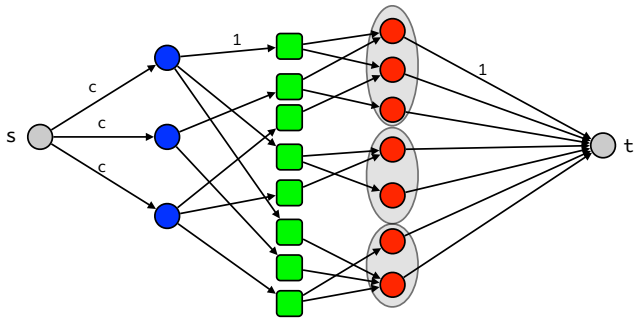
Scheduling of doctors

- X doctors, Y holidays, each doctor should work at at most c holidays, each doctor is available at some of the holidays.
- Same problem, but each doctor should work at most one day in each vacation period?



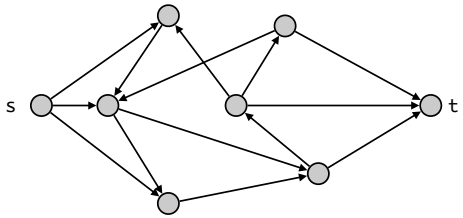
Scheduling of doctors

- X doctors, Y holidays, each doctor should work at at most c holidays, each doctor is available at some of the holidays.
- Same problem, but each doctor should work at most one day in each vacation period?



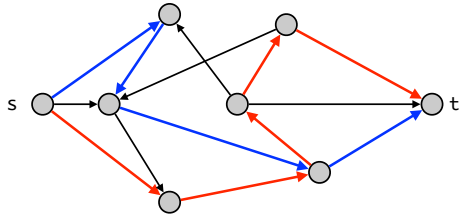
Edge Disjoint paths

- Problem: Find maximum number of edge-disjoint paths from s to t.
- Two paths are edge-disjoint if they have no edge in common.



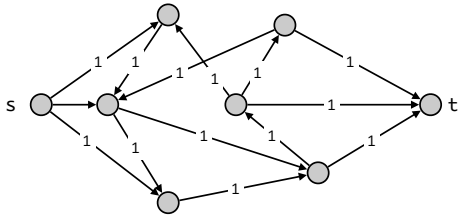
Edge Disjoint paths

- Edge-disjoint path problem. Find the maximum number of edge-disjoint paths from s to t.
- Two paths are edge-disjoint if they have no edge in common.



Edge Disjoint Paths

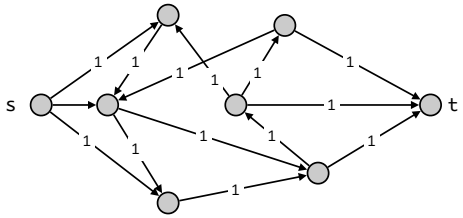
- Reduction to max flow: assign capacity 1 to each edge.



- Thm. Max number of edge-disjoint s-t paths is equal to the value of a maximum flow.
  - Suppose there are k edge-disjoint paths: then there is a flow of k (let all edges on the paths have flow 1).
  - Other way (graph theory course).
- Ford-Fulkerson: v(f) ≤ n (no multiple edges and therefore at most n edges out of s) => running time O(nm).

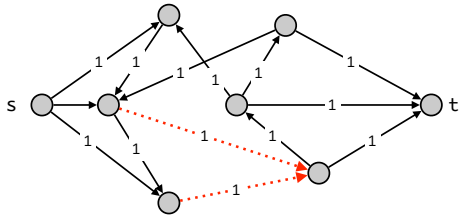
Network Connectivity

- Network connectivity. Find minimum number of edges whose removal disconnects t from s (destroys all s-t paths).



Network Connectivity

- Network connectivity. Find minimum number of edges whose removal disconnects t from s (destroys all s-t paths).



- Set all capacities to 1 and find minimum cut.
- Thm. (Menger) The maximum number of edge-disjoint s-t paths is equal to the minimum number of edges whose removal disconnects t from s.

Baseball elimination

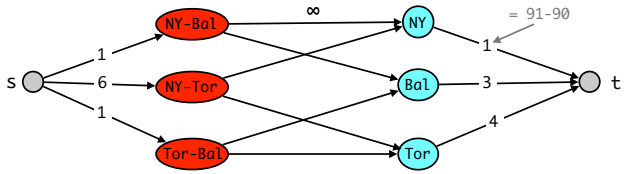
Team	Wins	Games left	Against			
			NY	Bal	Tor	Bos
New York	92	2	-	1	1	0
Baltimore	91	3	1	-	1	1
Toronto	91	3	1	1	-	1
Boston	90	2	0	1	1	-

- Question: Can Boston finish in first place (or in tie of first place)?
- No: Boston must win both its remaining 2 and NY must loose. But then Baltimore and Toronto both beat NY so winner of Baltimore-Toronto will get 93 points.
- Other argument: Boston can finish with at most 92. Cumulatively the other three teams have 274 wins currently and their 3 games against each other will give another 3 points => 277. 277/3 = 92,33333 => one of them must win > 92.

Baseball elimination

Team	Wins	Games left	Against			
			NY	Bal	Tor	Bos
New York	90	11	-	1	6	4
Baltimore	88	6	1	-	1	4
Toronto	87	11	6	1	-	4
Boston	79	12	4	4	4	-

- Question: Can Boston finish in first place (or in tie of first place)?

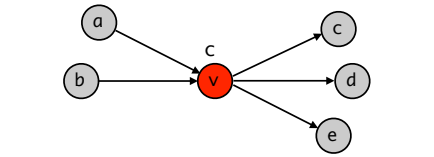


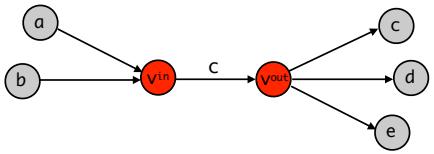
Boston can get at most 79 + 12 = 91 points

- Boston cannot win (or tie) ⇔ max s-t flow < 8.

Node capacities

- Capacities on nodes.



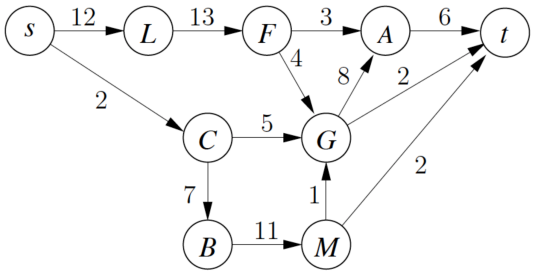


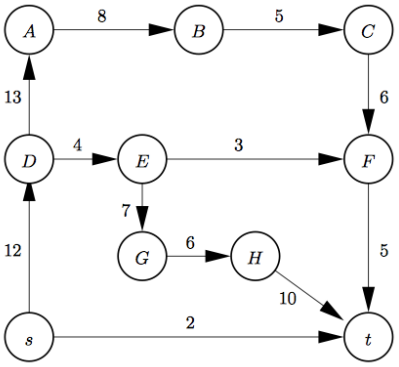
Reading Material

At the lecture we will talk more about network flow. We will consider two applications: bipartite matching and finding disjoint paths. We will also discuss how to find good augmenting paths. For reading material see the webpage.

Exercises

1 [*w*] **The Edmonds-Karp algorithm and the scaling algorithm** Use both the Edmonds-Karp's algorithm and the scaling algorithm to compute a maximum flow and minimum cut on the two graphs below. For each augmenting path write the nodes on the path and the value you augment the path with.





2 Blood Donations Solve exercise KT 7.8.

3 Christmas Trees (from the Exam E15) The Dean has asked you to arrange the annual Christmas party for the students at DTU. You have to make a plan for how to place the tables in the hall. The local fire department has divided the hall up into an $n \times m$ grid of subsquares and declared that you can place at most two tables in each row and at most one in each column. Unfortunately, the Dean who loves Christmas has put up Christmas trees in many of the subsquares. You cannot place a table in a subsquare with a Christmas tree.

Example Here $n = 4$ and $m = 8$. The $*$ are Christmas trees and T are tables. In the example the maximum number of tables that can be placed is 7.

$*$	T						T
T	$*$	$*$	T	$*$	$*$		$*$
	$*$	$*$		$*$	$*$	T	$*$
	$*$	T	$*$		T	$*$	

3.1 Model the problem as a graph problem. Explain how you model the problem as a graph problem and draw the graph corresponding to the example above.

3.2 Describe an algorithm that given n, m, and the placement of the Christmas trees computes the maximum number of tables you can place in the hall. Analyze the asymptotic running time of your algorithm. Remember to argue that your algorithm is correct.

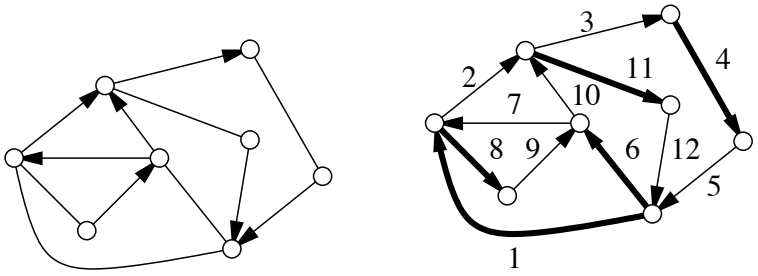
4 Escape Solve exercise KT 7.14.

5 CSES School dance Solve https://cses.fi/problemset/task/1696

6 [∗] **Euler tours in mixed graphs** It is well-known that a strongly connected graph has an Euler tour (a cycle that contains each edge exactly once) if and only if any vertex have the same number of ingoing and outgoing edges.

In the following we consider mixed graphs, which are graphs where some edges are directed and some are not. We assume that the graphs are connected, i.e., if we ignore the orientation on all directed edges, then the graph is connected.

We want an algorithm to decide whether it is possible to assign directions to all undirected edges, such that the graph has an Euler tour. In the example below the mixed graph on the left can be assigned directions (graph on right) such that there is an Euler tour (the numbers on the edges denotes the order the edges can be visited in).



Give an algorithm to decide wether it is possible to assign directions to all undirected edges, such that a graph G has an Euler tour.

Hint: Consider a bipartite graph, where the nodes on one side are edges in G and the nodes on the other side are the vertices in G.

Puzzle of the week: 99 Cops A town has 99 cops. A cop is either honest or corrupt, the majority of the cops is honest. You need to figure out all the corrupt cops, with less than 299 questions. All cops know who is honest and who is corrupt, but only honest cops will answer truthfully. Corrupt cops may lie arbitrarily. For security reasons you can only ask one type of question: You may ask cop X whether cop Y is corrupt. This question will by answered by X with either "Y is corrupt" or "Y is honest".