Shortest path

Heaps

- Heaps are an efficient way of implementing the priority queue for Dijkstra
- Binary min-heap = a binary tree in which layers 1, .., l−1 are full and layer l may be partially full but is filled from the left, and every node's key is no smaller than its parents key
- A binary tree (and so a heap) can be represented as a sequence
- Root has index 1
- Index of left child of node with index i is 2i
- Index of right child of node with index i is 2i + 1
- Index of parent of node with index i is ⌊i/2⌋

Heap Algorithms For Dijkstra

- heapDecreaseKey(H, i, new):

```
p(u_i) = new
// Upheapify
While i > 1 and key of parent(u_i) > key of u_i
    u_j = parent(u_i)
    Swap u_i and u_j in H
    u_i = u_j
```

- heapRemoveMin(H):

```
ret = v_1
Move v_n to root (v_1) in H
// Downheapify
i = 1
v_j = child of v_{i=1} with smallest key
while key v_j < key of v_i
    Swap v_i and v_j in H
    i = j
    v_j = child of v_i with smallest key
return ret
```

Dijkstra

- [n] = {1, …, n}
- A path = a sequence of distinct vertices (v_0, …, v_n) for which there are arcs (v_0, v_1), …, (v_{n-1}, v_n)
- Dijkstra's shortest path algorithm:

```
d(s) = 0
d(u) = +∞ for u ∈ V\{s}
T = (u_1=s, …, u_n) where u_2, …, u_n are V\{s} in an arbitrary order
While T is not empty
    u = heapRemoveMin(H)
    For v ∈ {v: (u, v) ∈ E ∧ v ∈ T}
        alt = d(u) + c(u, v)
        heapDecreaseKey(H, indexOf(v), alt) if d(v) > alt
return d
```

- Proposition: Terminates for any (finite) graph
  Proof: Terminates iff T = ∅. T = V at start which is finite and exactly one element is removed from T every iteration after doing at most O(|E|) work which is also finite

Dijkstra: Correctness: Lemmas

- Let $d^*(u)$ = the minimum cost of an su-path
- Lemma 1: The following is a loop invariant: $\forall u \in V.\ d(u) \geq d^*(u)$
  Proof: We will use induction on the length l of the minimum of the minimum cost su-path
  Base case l=0: There is exactly one vertex u with an su-path (), u=s. Hence, $d^*(u) = 0$ and indeed $d(u) = 0$
  Inductive hypothesis: The loop invariant holds for the first k iterations
  Inductive step: Consider the $k+1^{th}$ u to be selected. Recall d(v) is non-increasing for all $u \in V$.
  Case d(v) is not altered: We are done.
  Case some d(v) is decreased: Then, $d(v) = d(u) + c(u, v)$. By IH, $d(u) \geq d^*(u)$ and so $d(v) \geq d^*(u) + c(u, v) \geq d^*(v)$ as $d^*(u) + c(u, v)$ describes the cheapest cost of a particular type of sv-path and $d^*(v)$ describes the cheapest cost of all sv-paths. We can apply this logic for every value that is altered in every iteration.
- Lemma 2: If $P = (v_0=s, \ldots, v_m=u)$ is a minimal su-path, then $\forall i \in \{0, \ldots, m\}$ [$(v_0, \ldots, v_i)$ is a minimal sv_i-path].
  Proof: Assume for the sake of contradiction, there exists a path P and values of i for which the statement does not hold, take the smallest such i. Then, there exists a sv_i-path $(v'_0=s, \ldots, v'_k=v_i)$ that is cheaper than the sv_i-path $(v_0, \ldots, v_i)$ and so $c(v'_0, v'_1) + \ldots + c(v_{k-1}, v_k) < c(v_0, v_1) + \ldots + c(v_{i-1}, v_i)$.
  We can extend each of these to obtain su-paths $(v'_0=s, \ldots, v'_k=v_i, v_{i+1}, \ldots, v_m=u)$ and $(v_0, \ldots, v_i, v_{i+1}, \ldots, v_m=u)$, this adds the same cost $c(v_i, v_{i+1}) + \ldots c(v_{m-1}, v_m)$ to each path and so by our inequality the former is cheaper than the later, so P was not a minimal su-path?!

Dijkstra: Correctness

Theorem: **If all costs are non-negative**, then $\forall u \in V[d(u) = d^*(u)]$ at the end of the algorithm

Proof: Assume for the sake of contradiction $\exists u \in V$: $d(u) \neq d^*(u)$, by lemma 1 $d(u) > d^*(u)$, if there is more than one such u pick the one that was removed from T first.

We will consider the state at the moment the algorithm picks u from T. Pick an arbitrary minimum su-path $P = (v_0=s, \ldots, v_m=u)$ and let v_i be the first member of P to be in T. Then, $v_{i-1} \notin T$ [and as $u \neq s$, $1 \leq i \leq m$] and so $d(v_{i-1}) = d^*(v_{i-1})$ by our assumption as it has been removed from T and is not u. By the algorithm, $d(v_i) \leq d(v_{i-1}) + c(v_{i-1}, v_i)$ [as v_{i-1} has already been chosen].

$d(v_{i-1}) + c(v_{i-1}, v_i) = d^*(v_{i-1}) + c(v_{i-1}, v_i) = c(v_0, v_1) + \ldots + c(v_{i-2}, v_{i-1}) + c(v_{i-1}, v_i)$ by lemma 2 $= d^*(v_i)$ by lemma 2. Hence, $d(v_i) \leq d^*(v_i)$ [in fact $d(v_i) = d^*(v_i)$ by lemma 1] and so $i \neq m$.

Finally, $d^*(v_i) \leq d^*(u)$ as all costs are non-negative but $d^*(u) < d(u)$ by our assumption and so the algorithm would not have picked u but instead v_i as $d(v_i) \leq d^*(v_i) < d(u)$?!

Bellman-Ford

- The underlying dynamic programming of Dijkstra is widely applicable, but being able to only consider each vertex as the source only once is a luxury only present with non-negative costs
- A path does not repeat vertices whereas a walk can even repeat edges
- A negative circuit is a circuit (walk with $v_0=v_m$) with negative total cost
- A cycle is a circuit in which the only repetition of edges is $v_0=v_m$
- Lemma: If there are no negative circuits, then for every st-walk W there exists an st-path P that is at least as cheap as W. Proof: If a walk is not a path, it contains a circuit. As there are no negative circuits, removing this circuit will not increase the cost. Repeat until a path is obtained.
- If a graph is undirected it either contains a negative circuit reachable from s or no negative edges are reachable from s and so Dijskta can be used
- **Bellman-Ford shortest path algorithm:**

```
n = |V|
d(s) = 0
d(u) = +∞ for u ∈ V\{s}
for i = 1, …, n
    for v ∈ V
        d_i(v) = min({d_{i-1}(v)} ∪ {d_{i-1}(u) + c(u, v): (u, v) ∈ E})
return "Negative circuit detected" if d_n ≠ d_{n-1}
return d_n
```

Bellman-Ford: Correctness

- Theorem: Bellman-Ford detects a negative circuit iff there is one
  Proof: Non-examinable
- Theorem: When it does not detect a negative-circuit, Bellman-Ford returns d^*
  Proof sketch due to time: It can be shown by induction on i that **at the end of each iteration $d_i(u)$ = the shortest path su-path out of the paths with length at most i** (denote this $d_i^*(u)$). As all paths have at most $|V|=n$ vertices, this would suffice.

Minimum Spanning Tree

Graphs

- We only deal with undirected graphs for MST
- Our MST algorithms can only handle non-negative costs, but unlike Dijkstra adding a sufficiently large constant is a legal reduction
- **Cutset of $A \subseteq V = \delta(A) = \{(u, v) \in E: u \in A \text{ and } v \in V \backslash A\}$**
- Lemma: G is connected $\Leftrightarrow ((\forall A \in 2^V. \delta(A) = \varnothing \Rightarrow (A \text{ is non-proper } (A = V \text{ or } A = \varnothing))))$
  Proof: $\Rightarrow$: Obvious
  $\Leftarrow$: The contrapositive is G is not connected $\Rightarrow \exists A \in 2^V$: $\delta(A) = \varnothing$ and A is proper.
  Pick an arbitrary not connected G. Then, G contains connected components $S_1, \ldots, S_k$ ($k \geq 2$). Hence, $\delta(S_1) = \varnothing$ and S_1 is proper as required.

Prim's

- **Prim's algorithm:**

```
T = ({r}, ∅) for an arbitrary r ∈ T
while V(T) ≠ V(G)
    Pick {x, y} ∈ argmin_{σ(V(T))} c
    V(T) ∪= {x, y} // Exactly one of these is new, but we don't know which
    E(T) ∪= {{x, y}}
return T
```

- **We call $E' \subseteq E(G)$ extendable iff there exists an MST T such that $E(T) \supseteq E'$**
- Theorem: Prim's is correct
  Proof: We will use induction on |E(T)| to show that E(T) is extendable is a loop invariant and hence as the loop break condition is T being spanning (and we maintain connectivity and acyclicity throughout) conclude that the return value must be an MST as it is extendable but cannot be extended other than by doing nothing.
  Base case |E(T)|=0: E(T) = ∅ which is a subset of every set
  Inductive hypothesis: Holds up to |E(T)|=k
  Inductive step: Consider |E(T)|=k+1. Let T_P denotes the state of T at the end of previous iteration (when the inductive hypothesis still held) and e = {x, y} denote the edge added to T_p to form T. Let $M \subseteq G$ be an arbitrary MST with $E(M) = E(T_P)$. If $e \in E(M)$, then T is obviously extendable. We now consider the case $e \notin E(M)$. Adding e to M forms a cycle and there is exactly one other edge in that is in both the cycle (i.e. in E(M)) and in $\sigma(V(G)\backslash V(T_P))$ ($=\sigma(V(T_P))$ as the graph is undirected) call this f. As e was a cheapest edge in $\sigma(V(T_P))$, adding e and removing f will form a new spanning tree and will not increase the cost and so we have constructed a new MST from E(T) ($f \notin E(T)$ as $f \in \sigma(V(T_P))$ and $f \neq e$).

Kruskal's

- **Kruskal's algorithm:**

```
F = (V, ∅)
While F is disconnected
    Pick e ∈ argmin_{e ∈ E(G)\E(F): Adding e to F does not introduce a cycle} c
    E(F) ∪= {e}
return F
```

  - *Cycles can be (efficiently) detected using breadth-first search*
- Theorem: Kruskal's algorithm is correct
  Proof: We will use induction on |E(F)| to show that E(F) is extendable is a loop invariant and hence as the loop break condition is F is connected (and F is spanning and acyclic throughout) conclude that the return value must be an MST as it is extendable but cannot be extended other than by doing nothing.
  Inductive hypothesis: Holds up to the end of some iteration
  Inductive step: If the loop breaks instead of starting a new iteration, we are done. We now consider the case where a new iteration starts.
  Call the connected components of F S_1, …, S_k ($k \geq 2$ as F is not connected). Let $e = \{x, y\}$ be the edge chosen by the algorithm, x and y must be in different connected components are otherwise adding e would introducing a cycle but e was chosen. We will have to add an edge that joins these connected components at some point and the algorithm chose one with minimal cost (looking over a superset of this cutset and so is also minimal within this cutset), by the same argument as Prim's the resulting tree must be extendable.

Network flow

Flow networks

- A network (V, A, b, s, t, c) is a digraph (V, A) equipped with source (start) vertex s, t sink (target) vertex t and capacity (bound) function b (b: $A \mapsto \mathbb{R}_{\geq 0}$ or equivalently but more conveniently b: $V^2 \mapsto \mathbb{R}_{\geq 0}$ where b(x, y) = 0 if $(x, y) \notin A$)
- **A flow in N is a function f: $A \mapsto \mathbb{R}_{\geq 0}$ such that:**
  - **Flows do not exceed capacities ($\forall uv \in A$. f(uv) ≤ b(uv)) and**
  - **Flow is conserved** (at every intermediate node flow in = flow out) **($\forall v \in V\{s, t\}$. $\sum_{(u, v) \in E}$ (f(u, v) − f(v, u)) = 0)**
- **Value of a flow f = |f| = $\sum_{v \in V}$ (f(s, v) − f(v, s)) = the net flow out of s**
- **X is an s-t cut iff $s \in X$ and $t \notin X$**
- **Capacity of an st-cut X = b(δ(X)) = $\sum_{(u, v) \in \delta(X)}$ b(u,v) = sum of capacities of edges <u>from</u> nodes in X <u>to</u> nodes outside X**
- **Flow value lemma: Value of a flow f = net flow across any st-cut X** := $\sum_{(u,v) \in \delta(X)} f(u,v) - \sum_{(u,v) \in \delta(V\setminus X)} f(u,v)$
  Proof:
  By flow conservation, $\forall v \in V\{s, t\}$. $0 = \sum_{(u,v) \in A} f(u,v) - \sum_{(v, u) \in A} f(v,u)$
  Thus, $0 = \sum_{v \in X\{s\}} (\sum_{(u,v) \in A} f(u,v) - \sum_{(v, u) \in A} f(v,u))$ [note $t \notin X$ by definition] $= \sum_{(u,v) \in A} f(s,v) + \sum_{(u,v) \in \delta(V\setminus X)} f(u,v) - \sum_{(v, u) \in \delta(X)} f(v, u))$ as s binds to u in $(u, v) \in A$ and arcs where both directions are inside the cut cancel out to 0 respectively. Hence, $0 = |f| + \sum_{(u,v) \in \delta(V\setminus X)} f(u,v) - \sum_{(v, u) \in \delta(X)} f(v,u))$ and so $|f| = \sum_{(v, u) \in \delta(X)} f(v,u)) - \sum_{(u,v) \in \delta(V\setminus X)} f(u,v)$.
  Corollary: The capacity of every st-cut is at least the net-flow across it and so minimum capacity over all st-cuts ≥ maximum flow value over all flows

Ford-Fulkerson Theorem

- T**he residual diagraph of a flow f in a flow network N is $D_N(f)$** = (V, A'={(x, y): [(x, y) ∈ A and f(x, y) < b(x, y)] or [(y, x) ∈ A and f(y, x) > 0], b'(x, y) = b(x, y) − f(x, y) if (x, y) ∈ A, f(y, x) if (y, x) ∈ A} **— the same vertices as N and a forward edge with the remaining capacity (if non-zero) and a reverse edge with the used capacity (if non-zero)**
- **Width of a path = smallest b' on it [the maximum amount flows can be changed along it without breaking the constraints]**
    - **A critical edge is an edge with b' of the width**
- **Ford-Fulkerson Theorem: A flow is maximal iff there are no augmenting paths (st-paths in the residual network)**

  Proof:

  If there exists an augmenting path P, then |f| is not maximal: By the construction of the residual network, the width of P is strictly positive. By the definition of width, |f| can be increased by the width and so was not maximal.

  If there does not exist an augmenting path, then |f| is maximal: Let X = {x ∈ V: $D_N(f)$ has an sx-path}. Then, δ(X) is an st-cut in N. Moreover, every edge leaving the cut must be saturated (flow = capacity) else it would be present in $D_N(f)$ and so would be inside the cut. Similarly, every edge entering the cut must be unused (flow = 0) because otherwise it would also have an edge in $D_N(f)$ in the other direction and so would be inside the cut. Hence, by flow value lemma, |f| = capacity of X (− 0) and so is maximal as each cut capacity is an upper bound on the possible flow value.

Algorithms for MAX-FLOW

- **Ford-Fulkerson algorithm:**

```
f(x, y) = 0 for (x, y) ∈ A
while there exists an augmenting path (a path from s to t) p in D_N(f)
    For each forward edge taken in p, increase f at the corresponding edge in N by the width of p
    For each backward edge taken in p, decrease f at the corresponding edge in N by the width of p
return f
```

- *If capacities are integers, Ford-Fulkerson terminates but time taken depends on capacities*
- If capacities are not quantized, Ford-Fulkerson may not terminate
- **Edmonds–Karp algorithm modifies Ford-Fulkerson to provide a termination guarantee by simplifying the arbitrary choice of augmenting path to only those that use the fewest edges (e.g. breadth-first-search)**
  - Breadth-first-search algorithm:

```
d(s) = 0
d(v) = +∞ for v ∈ V\{s}
For i = 1, …, |V| − 1
    For uv ∈ A
        if d(u) = i − 1 and d(v) = +∞ // u was discovered in last step and v is not yet discovered
            d(v) = i
            prev(v) = u
```

Max-flow-min-cut Theorem

- **Max-flow-min-cut theorem: minimum capacity over all st-cuts = maximum flow value over all flows**
  Proof: Omitted due to time
  - *min-cut and max-flow are polytime equivalent* — To obtain a min-cut from a max flow: Use reachability in residual graph from s to create X [this is the construction we used in the proof of Ford-Fulkerson]

Matchings

Menger's Theorem

- **Menger's Theorem: Let S, T be disjoint subsets of the vertices of a digraph G=(V, A). Then the cardinality of the largest set of arc-disjoint ST-paths is equal to that of the smallest cutset over ST-cuts (or equivalently the smallest set of edges which if all removed would disconnected S from T).**
  Proof: It is obvious that there are at most as many disjoint paths as edges in the smallest cut set. It remains to show that we can construct enough paths.
  Construct N = (V'=V ∪ {s, t}, A'=A ∪ {sx: x ∈ S) ∪ {yt: y ∈ T}, b(e) = 1 if e ∈ A and a sufficiently large number [e.g. |A|+1] otherwise], s, t) and find an integral max-flow f in N (e.g. Edmonds-Karp as all capacities are integer). Let k = |f|. AWLOG that if there is a bidirectional arc between nodes at most one of them is used in the f.
  We can then construct a subgraph H = (V', A''={xy ∈ A': f(xy) > 0}). Pick an arbitrary st-path P=(v_0=s, …, v_m=t) in H, the flow of each edges in P that were from A will be 1. We can decrease f along P by 1 to create a new flow f' with |f'|=k−1 **and then construct a new H' and P' by the same procedure to find a f''** with |f''|=k−2 **and so on until we obtain $f^{(k)}$ with $|f^{(k)}|=0$**. This gives us a sequence P, …, $P^{(k)}$ of necessarily edge disjoint paths in G (after stripping away s and t).
  It remains to show that that the size of the min ST-cut in G is k. By the max-flow-min-cut theorem, there exists a min st-cut X in N such that b(δ(X)) = |f| = k. As there are k disjoint ST-paths in G, k ≤ |A|, hence no edge not from G can be in δ(X) as these were assigned capacity at least |A| + 1. Hence, X is also an ST-cut in G and contains k arcs (as all the arcs it contains had capacity 1). X is not necessarily a min ST-cut from this argument but this suffices due to our previous arguments.

Matching number (MAX-MATCHING)

- **A matching in an undirected graph is a set of edges with no shared vertices** or equivalently such that each vertex in the graph is connected to a most one edge in the matching
- **Matching number of G = μ(G) = max {|M|: M is a matching in G}** — mu for matching number
- **Vertex cover number of G = τ(G) = min {|X|: X ⊆ V and every arc in G has an endpoint in X}**
- Lemma: μ(G) ≤ τ(G)
  Proof: By definition each edge in a matching requires a distinct vertex to cover it. Hence, just to cover a given matching we need as many nodes as there are edges in the matching, to cover all edges in the entire graph we may need further nodes.
- **Kőnig's theorem: G is bipartite ⇒ μ(G) = τ(G)**
  Proof: Let G be an arbitrary bipartite graph with partitions P, Q. It will suffice to show that μ(G) ≥ τ(G).
  Construct a flow network N by directing all edges to be from P into Q and adding dummy source and sink nodes connected to all nodes in P and Q respectively. Assign edges that were in G sufficiently large capacity (e.g. |E|+1) and edges incident on the dummy nodes capacity 1 [note this is the other way round to Menger's].
  We can define a bijection between matchings M in G and integral flows f in N, f(xy) = 1 ⇔ xy ∈ M.
  Pick an arbitrary min cut X in N. X can be written in the form X = {s} ∪ P′ ∪ Q′ where P′ ⊆ P and Q′ ⊆ Q. As μ(G) cannot exceed |E|, by max-flow-min-cut no edges that were in G can be present in X as X is a min-cut and μ(G) corresponds to a max flow. Hence, (P\P′) ∪ Q′ is trivially a vertex cover in G. Moreover, b(δ(X)) = number of arcs from S that do not go into P′ [i.e. go into P\P′] + number of arcs that leave Q′ = |P − P′| + |Q′|.
  Thus, τ(G) ≤ |P − P′| + |Q′| = b(δ(X)) = |f| for any max flow f by max-flow-min-cut = μ(G) as required.

Hall's marriage theorem

- **Hall's marriage theorem: Let G = (V, E) be a bipartite graph with partitions P, Q. Then, G has a matching of size $|P| \Leftrightarrow \forall S \subseteq P. |N(S)| \geq |S|$ where N(S) = neighbourhood of $S = \{y \mid \exists x \in S: (x, y) \in E\} \subseteq Q$**
  Proof: ⇒: A matching of size |P| is a witness to P having at least |P| neighbours and so on.
  ⇐: Let $P' \cup Q'$ be an arbitrary vertex cover and consider $S = P \backslash P'$. Then, $|S| \leq |N(S)| \leq |Q'|$. Moreover, $|S| = |P| - |P'|$ and so $|P| \leq |Q'| + |P'| = |P' \cup Q'|$ as P' and Q' are disjoint. So, as G is bipartite and $|P| \leq \tau(G)$, by Kőnig's theorem $|P| \leq \mu(G)$ and so G must have a matching of size |P|.
  - *Marriage formulation: Given sets of men and women and a mutual love relation, we can arrange loving marriages of all women to distinct men if (and only if) for every subset of the women there are at least as many men who love a woman in that subset as there are women in that subset*

Weighted matchings

- **Assignment problem: We have n workers (w_1, …, w_n), n tasks (t_1, …, t_n), and a cost matrix $[C_{ij}]_{nxn}$ of the cost of w_i doing t_j. Find a minimum cost bijective assignment.**
  - Deduce adding a constant to a row/column of C does not affect optimality
- **Hungarian Algorithm: <u>Precondition: All costs are non-negative</u>**

```
while G = (W={w_1, …, w_n} ∪ T={t_1, …, t_n}, {w_i t_j: c_ij = 0}) has no perfect matching
    Find a minimal vertex cover W′ ∪ T′ in G [the smallest set of rows and columns that when removed from C remove all zeroes] // e.g. use Ford-Fulkerson to find a min cut
    Y = {c_ij: w_i ∈ W\W′ and t_j ∈ T\T′} // All costs that are not touched by the vertex cover
    m = min Y
    for (i, j) ∈ Y
        c_ij -= m // reduce costs without making any negative
    for (i, j) ∈ {c_ij: w_i ∈ W′ and t_j ∈ T′} // All costs that are touched twice by the vertex cover
        c_ij += m
    // For costs that are touched exactly once, we do nothing
Return such a perfect matching  [0 is the smallest cost we allow and hence such a matching must be optimal]
```

Correctness proof sketch: We can rewrite the cost updating step as add m/2 for each row and each column that is in vertex cover, subtract m/2 for each row and each column that is not in vertex cover. Hence, we are not affecting optimality.

Stable matchings

- **mw, m'w' is pair of unstable pairs iff ∃ m', w': w' $>_m$ w and m $>_{w'}$ m'**
- **A matching is stable iff it contains no unstable pairs**
- **Gale-Shapley Algorithm: <u>Precondition: Sets of men and women are equinumerous</u>**

```
while there is a man who does not hold a maybe
    Every man proposes to the top woman on his remaining list
    Every woman says maybe to their preferred proposal and no to every other
proposal overriding a previous maybe if necessary
    Each rejected man removes the top woman [i.e. the one who just rejected
him] from his list
return the matching induced by the maybes
```

- Proof of termination of Gale-Shapley: In each run of the loop either at least one mans list shrinks or the loop is ending. We terminate in $O(n^2)$
- Proof of completeness of Gale-Shapley matching: Obvious
- **Proof of stability of Gale-Shapley matching: Assume for the sake of contradiction ∃ m, w, m', w': mw and m'w' are matched but w' $>_m$ w and m $>_{w'}$ m'. As m was matched with w which they prefer less than w', they must have proposed to and been rejected by w' in favour of some other proposal from an m''.** Hence, m'' $>_{w'}$ m. w' may or may not have subsequently rejected m'' but if so they would only have done so for an even better proposal. Hence, m' $\geq_{w'}$ m'' $>_{w'}$ m. But, by our assumption m $>_{w'}$ m' and so by transitivity m $>_{w'}$ m?!

Stable matchings: Optimality properties

- **opt(p) = The highest ranked of p's partners over all stable matchings**
- **pess(p) = The lowest ranked of p's partners over all stable matchings**
- **We call a matching male/female-optimal/pessimal iff it is stable and it is optimal/pessimal for all males/females**
- **Lemma: Any male-optimal matching is female-pessimal**
  Proof:
  Let S be an arbitrary male optimal matching and mw be an arbitrary pair in M (hence, w = opt(m)). We will show that m = pess(w).
  For any stable matching A, w' that m is matched to in A, and m' that w is matched to in A, w' $\leq_m$ w as w = opt(m) but then m' $\geq_w$ m as A is stable.
  Thus, there is no stable matching in which w is matched with someone worse than m and they are matched with m in S so this arbitrary S is indeed the pessimal case for this arbitrary w.
- **Theorem: Any Gale-Shapley matching is male-optimal (and thus female-pessimal)**
  Proof: Assume for the sake of contradiction there exists a man m that is rejected by w = opt(m), if there is more than one instance pick the first to occur in the execution of the algorithm. Then, there must be an m' that proposes to w and m' $>_w$ m. By our first instance condition, either w is opt(m') and they stay together or w is preferred to opt(m') but is always unstable but has not yet been rejected, either way w $\geq_{m'}$ opt(m').
  As w = opt(m), there is a stable matching in which m and w are matched. Let w^* = the match of m' in this stable matching. Then, w $\geq_{m'}$ opt(m') $\geq_{m'}$ w^*, and w $\neq$ w^* as w is matched to m and m $\neq$ m'. Hence mw and m'w^* are a pair of unstable pairs as m' $>_w$ m and w $>_{m'}$ w^* and so there is no stable matching in which w and m match?!

Linear programming

Standard form

- A linear programming problem instance in approximately canonical form is a **cost vector $c = (c_1, \ldots, c_n)$**, a **constraint vector $(b_1, \ldots, b_m)$** and a **constraint matrix $A = [w_{ij}]_{nxm}$** — note if we want sign constraints at this point we must explicitly include them
- **We call a vector $x = (x_1, \ldots, x_n)$ feasible iff $Ax \leq b$** and we call the set of all such x the feasible region
- The linear programming algorithmic problem is to find an $x \in \operatorname{argmin}_{x \in F} c \cdot x$ — we call such x optimal (note **optimal ⇒ feasible**)
- We can rewrite any linear programming problem instance in approximately canonical form to be in **standard form: min $c \cdot x$ subject to $Ax = b$ and $x \geq 0$**
  Construction:
  1. Replace each $a_j \cdot x \leq b_j$ with $a_j + s_j = b_j$
  2. Replace each x_i with $y_i^+ - y_i^-$
  3. Add sign constraint $(y_1^+, , \ldots, y_n^+, y_1^-, \ldots, y_n^-, s_1, \ldots, s_m) \geq 0$
- For notational convenience, **a_i = the row vector of the ith row of A** $= (w_{i1}, \ldots, w_{in})$ and **A_i = the column vector of the ith column of a** $= (w_{1i}, \ldots, w_{mi})^T$

Basic solutions

- Lemma: **Any LP in standard form can be reduced to have linearly independent constraints**. Proof: If there is a linear dependency then a constraint can be written as a linear combination of other constraints and so is satisfied iff those constraints are satisfied and hence is redundant as its removal will not affect the feasible region.
- **$A_{X = \{a, \ldots, k\}}$ = A with all m rows but with the columns with index not in X deleted**
- $x_{X = \{a, \ldots, k\}}$ = x with entries with index not in X deleted
- **I is a basic set iff A_I has full rank (all m rows are still linearly independent)**
- **A_I is a basis iff I is a basic set <u>and I=|m|</u>** iff A_I is invertible
- A basic set I induces a basic solution defined by Ax = b and $\forall i \in [n] \backslash I.\ x_i = 0$ — all x_i not in the basis are 0 and x is feasible except possibly the non-negativity constraint
- Every basis is square and so **the basic solution is defined by the basis A_I is $x_I = A_I^{-1} b$ and $\forall i \in [n] \backslash I.\ x_i = 0$** and so is unique and always exists
  - To find this easily: Augment A_I with b, use elementary row operations to get identity matrix on LHS, read off RHS to get x_I
- **A basic solution is a basic feasible solution iff all its entries are non-negative**

Basic solutions are all you need

- Theorem: Given a linear program in standard form with A with full rank, non-empty feasible set F, and objective function bounded below the following hold: i) There exists a basic feasible solution and ii) For every feasible solution there exists a basic feasible solution with objective function value at most that of the feasible solution. Hence, **there exists a bfs that is an optimal solution to the LP**.
  Proof: Omitted due to time
- Naive LP solving algorithm:

```
M = +∞
for I ⊆ [n]
    continue if |I| != m or rank(A_I) != m
    x = A_I^-1 b
    if x > 0 and c·x < M
        M = c·x
        x* = x
if M = +∞
    return "Infeasible"
if adding the constraint c·x ≤ M − 1 makes the problem infeasible [e.g. run same
algorithm but without this branch]
    return "Value={M}, solution={x}"
else // The problem was still feasible
    return "Unbounded, value=−∞"
```

Multi-dimensional geometry

- **A half-space is $\{x \in \mathbb{R}^n: a \cdot x \leq b \text{ and } a \in \mathbb{R}^n \text{ and } b \in \mathbb{R}\} \subseteq \mathbb{R}^n$**
- **A polyhedron is $\{x \in \mathbb{R}^n: Ax \leq b \text{ and } A \in \mathbb{R}^{n \times m} \text{ and } b \in \mathbb{R}^m\} \subseteq \mathbb{R}^n$** — an intersection of m half-spaces
- **A hyperplane is $\{x \in \mathbb{R}^n: Ax = b \text{ and } A \in \mathbb{R}^{n \times m} \text{ and } b \in \mathbb{R}^m\} \subseteq \mathbb{R}^n$** — an intersection of two half-spaces with same magnitudes of coefficients but opposite signs
- **A polytope is a polyhedron which is bounded** (there exists a neighbourhood of finite radius around the origin that encloses it)
- **A face of a polyhedron P is argmin $\{\lambda \cdot x: x \in P\}$** — does not exist if $\{\lambda \cdot x: x \in P\}$ is unbounded below
  - Every polytope is a face of itself through $\lambda = 0$
- **A vertex is a singleton face** (a point $x \in P$ for which there exists a λ such that $\lambda \cdot x$ is a global minimum over $x \in P$)
- **An edge is a face that is the convex combination of two vertices**
- Lemma: **The feasible region is a polyhedron (trivial) and the set of basic feasible solutions is exactly the set of vertices** (vertex $\Leftrightarrow$ bfs)
  Proof: Omitted due to time

Simplex Algorithm

Simplex algorithm: Precondition: The provided initial basic set I gives a basic feasible solution

1. The initial tableau is A augmented above by c and to the right by b topped with z = 0
2. Use elementary row operations in rows 1-n to obtain a (possibly row permuted) identity matrix in A_I
3. Preserving the property from step 2, add/subtract rows to row 0 to obtain zeroes in the columns that are in the basic set I
4. If all entries in c are non-negative, we stop and return the bfs corresponding to I [as per our previous definition, $x_i = b_k$ where k is the row of A where 1 occurs in column i if $i \in I$, 0 if $i \notin I$] alongside its value which is −z
5. If there exists a column for which c is negative and every entry in that column in A is non-positive, we stop and return "Unbounded"
6. [If we have reached this point, there exists a column for which c is negative and for all such columns there exists an A entry that is positive]
7. Pick a $j \in \{j \in [n]: c_j < 0\}$ e.g. pick an argmin
8. Pick a $k \in \text{argmin}_{i \in [m]} \{b_i/a_{ij}: a_{ij} > 0\}$ — this time argmin is not a heuristic but required as it ensures b remains positive and so it is not only a basic solution but a basic feasible solution
9. Pivot on (k, j). $I = I \setminus \{s\} \cup \{j\}$ where s = the column of A that the 1 in the identity part of row k occurs in
10. Go back to step 2 to get a 1 for column j in row k [this was the point of our min rule for picking k] — this will naturally remove s from the basis unless the current solution is degenerate in which case it doesn't matter

Finding an initial basic feasible solution

Algorithm:

1. Make all $b_i \geq 0$ by negating b_i and a_i for all i for which $b_i < 0$
2. Add extra variables $s_1, \ldots, s_m \geq 0$ and rewrite each constraint $a_i \cdot x = b_i$ as $a_i \cdot x + s_i = b_i$
3. Replace the objective function with min $s_1 + \ldots + s_m$
4. $\{n+1, \ldots, n+m\}$ (i.e. $x=0$) is an obvious basic feasible solution, run simplex with this
5. If value > 0, the original LP was infeasible
6. Otherwise the solution is of the form $(x_1, \ldots, x_n, 0, \ldots, 0)$ and thus $(x_1, \ldots, x_n)$ is a basic feasible solution to the original LP and trimming down our A and b components of the tableau gives us our initial tableau

Correctness of simplex

- **Lemma: The simplex algorithm does not alter the feasible region induced by the tableau at each step**
  Proof: As we only use elementary row operations, our new constraints are linear combinations of the previous ones and so have the exact same sets of solutions
- **Lemma: For each x, $c \cdot x - z$ is constant throughout the steps. Moreover, $c \cdot x - z = c^O \cdot x$ where c^O denotes the original c (the one used in the actual objective function)**
  Proof: At step 0 this is certainly true. Assume for the sake of contradiction it becomes false for the first time at some step. Then, some γR_i has been added to R_0 such that $(c + \gamma a_i) \cdot x - (z + \gamma b_i) - [c \cdot x - z] \neq 0$ where c and z denote the values at the start of the breaking step. Thus, $\gamma a_i \cdot x - \gamma b_i \neq 0$. But then $\gamma a_i \cdot x \neq \gamma b_i$ and so x is not feasible?!
- Theorem: If it halts, simplex is correct (gives a feasible problem and a bfs for it, gives an optimal solution and its value iff the problem is bounded and says unbounded iff the problem is unbounded)
  Proof: Omitted due to time

Simplex Algorithm: Pivoting Rules

- For some inputs, some ways of choosing the pivot (k, j) [recall j is the column to be added and k is the row that decides which column s to be removed] can cause the algorithm to enter a cycle, if the same rule is always used entering a cycle obviously means the algorithm will not halt
- **$x < y$ iff x is lexicographically smaller than y iff x is numerically smaller than y in the first entry in which they differ**
- **Theorem: The lexicographic pivoting rule (still have freedom in picking j (subject to the standard condition $c_j < 0$) but use the k (out of those that obey the standard minimization condition** ($a_{kj} > 0$ and b_k/a_{kj} is minimal}**) for which $a_k/a_{kj}=(a_{k1}/a_{kj}, \ldots, a_{kn}/a_{kj})$ is lexicographically minimal) guarantees no cycling**
  Proof: Out of scope
- **Theorem: Bland's rule (Use the first index j such that $c_j < 0$ (the standard condition) and use the k (out of those that obey the standard minimization condition (out of those that obey the standard minimization condition** ($a_{kj} > 0$ and b_k/a_{kj} is minimal}**) such that the index s (column of 1 in identity matrix part of row k) removed by pivoting on (k, j) is minimal) guarantees no cycling**
  Proof: Non-examinable

Constructing dual

- **Our standard form is min c·x subject to Ax = b and x ≥ 0, the dual of such a program is max y·b subject to yA ≤ c** which can of course be rewritten in standard form if necessary
- **Each decision variable of primal corresponds to a constraint of dual and each constraint of primal corresponds to a decision variable of dual**
- Each constraint in primal becomes a sign constraint in dual and each sign constraint in primal becomes a constraint in dual
- Dual of dual is primal
- While we can do all dual constructions via standard form, it is feasible to convert directly:
  - min c·x becomes max y·b — if the primal is a max, rewrite it as a min first rather than trying to be clever
    - **RHS of constraints become coefficients for objective function and vice versa**
  - $a_i x \leq b_i$ becomes $y_i \leq 0$
  - $a_i x \geq b_i$ becomes $y_i \geq 0$
  - $x_j \geq 0$ becomes $yA_j \leq c_j$
    - **As we go from m constraints over n variables to n constraints over m variables, the coefficient matrix is effectively transposed**
  - $x_j \leq 0$ becomes $yA_j \geq c_j$
  - $x_j \in \mathbb{R}$ becomes $A_j = c_j$

Duality theorems

- **Weak duality theorem: Denote the objective function values of a pair of feasible solutions to some primal min LP and its dual max LP Z_P and Z_D respectively. $Z_P \geq Z_D$**
  Proof: Let x be an arbitrary feasible solution to the primal. As matrix multiplication is associative, (yA)x = y(Ax) = y·b as x is feasible.
  Let y be an arbitrary feasible solution to the dual. **As y is feasible, yA ≤ c and as x was a feasible solution to the primal x ≥ 0. Hence, (yA)x ≤ c·x.**
  Finally, by substitution, y·b ≤ c·x
- **Strong duality theorem: If an LP has finite value z, then its dual also has value z**
  [i.e. weak duality is exact equality if LP is feasible and bounded]
  Proof: Non-examinable as end of term treat
- **Fully duality theorem (corollary of prior duality theorems):**
  - **One of primal and dual is feasible and bounded ⇒ both are and values of objective function are the same at optimal solutions**
  - **One of primal and dual is infeasible ⇒ other is infeasible or unbounded**
  - **One of primal and dual is unbounded ⇒ other is infeasible**

Complementary slackness

- **Complementary slackness theorem: Let x be a feasible solution to a primal standard form LP and y be a feasible solution to its dual max LP. Then x, y are optimal** [by strong duality if one is optimal the other must also be] **⇔ ∀j. $(c_j - yA_j)x_j = 0$ or equivalently the dot product of the slack in either problem with the solution to the other is zero**
  Proof: As x is feasible, b = Ax. Thus, $c \cdot x - y \cdot b = (c - yA) \cdot x = \sum_{j \in [n]}(c_j - yA_j)x_j$
  ⇐: $\forall j.\ (c_j - yA_j)x_j = 0 \Rightarrow \sum_{j \in [n]} y_j b_j - yA_j x_j = 0 \Rightarrow c \cdot x - y \cdot b = 0 \Rightarrow c \cdot x = y \cdot b$
  ⇒: By strong duality, $c \cdot x = y \cdot b$ and so $c \cdot x - y \cdot b = 0$. Hence, $\sum_{j \in [n]}(c_j - yA_j)x_j = 0$ and so as the terms are non-negative every term must be 0
  Corollary: In an optimal solution, if the slack variable for the ith constraint of the dual is non-zero, then the ith primal decision variable is zero — this is useful if dual has fewer decision variables than primal as can solve dual more easily and can from that obtain a solution to primal