CS4234 Lecture Notes

[2020-08-13 Thu]

Contents

1 Introduction

1.1 Assessment

Problem Sets	20%
Mini Project	15%
Tutorial Attendance + Participation	5%
Midterms	25%
Final Assessment	35%

1.2 Optimization Algorithms

Find the minimum/maximum

- Discrete
- Combinatorial
- E.g. integers, graphs/trees

2 Minimum Vertex Cover

Vertex cover: a set of vertices, such that all edges are bounded by some vertex in that set

- Problem: can we find a vertex cover of size k?

Minimum vertex cover: minimum-sized vertex cover

Vertex cover is NP-complete

1. Vertex cover is in NP
   - We can verify a given solution (set of vertices of size k), in $O(V + E)$
2. Vertex cover is NP-hard (Clique, which is NP-hard, can be reduced to vertex-cover, i.e. Clique $\leq_p$ Vertex-cover — polynomial reduction)
   - Construct the complement graph $\bar{G}$
   - G has size k clique $\leftrightarrow$ then $\bar{G}$ has size $(n - k)$ vertex cover

Minimum vertex cover is NP-hard, but not NP-complete

For a complete graph of size k, you need $k - 1$ for a minimum vertex cover

2.1 Special Cases

MVC on *Trees* (not universal)

- For each vertex, either take it or don't take it
  - [IN] If you take it, then children don't need to (but may)
  - [OUT] If you don't take it, then children need to
  - Base case of a leaf: [IN] is 1, [OUT] is 0
- Dynamic programming solution in $O(n)$

MVC on *small* k (not fast)

- Pure brute force is $O(2^n \cdot m)$
- But small k can lead to not too bad outcomes: e.g. $k \leq 2$ — only try all possible pairs
- If k is much smaller than m, we can have a $O(n^k \cdot m)$ algorithm (???)
  - Pick a specific edge (u, v), then there are 2 cases — either pick u or v
  - We can solve the 2 remaining subproblems recursively: on the rest of the graph, with $k - 1$ remaining vertices to pick

2.2 Approximate Algorithm

Algo 1: Probabilistic 2-opt. For every remaining edge, add ONE arbitrary vertex on that edge. Repeat until you get vertex cover

- 2-opt only in expectation
- But size can be very huge: consider a huge star graph

Algo 2: Deterministic 2-opt. For every remaining edge, add BOTH vertices bordering that edge. Repeat until you get vertex cover

- 2-opt deterministically: size will not exceed 2 x OPT (optimal answer size) — (proof: see PDF???)

Algo 3: Pick the next vertex greedily, using the heuristic with the *highest degree* (use priority queue). Repeat until you get vertex cover

- But this algorithm can have a $\log n$ factor worse than the optimal answer

2.3 NP-hard problems

Desirable solutions

1. Fast
2. Optimal
3. Universal

For NP-hard problems, we can only have *2 out of 3*.

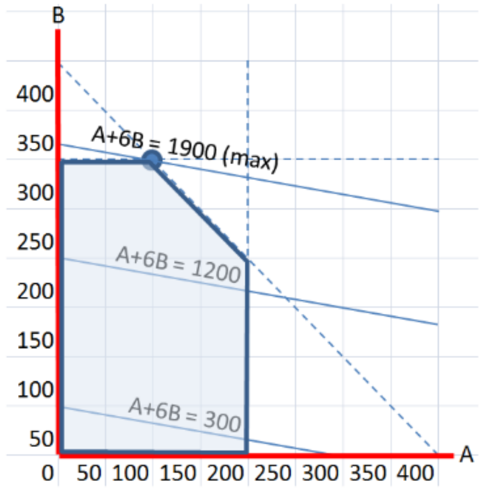
3 Linear Programming

3 components to LP problem:

1. Real-valued variables $x_1, \ldots, x_n$
2. Objective function $f(x_1, \ldots, x_n)$ — linear combination of variables
3. Set of m constraints that limits the solution space

3.1 Example

$$\begin{aligned} max(A + 6B) \text{ where:} \\ A \leq 200 \\ B \leq 300 \\ A + B \leq 400 \\ A \geq 0 \end{aligned}$$



The polygon is called a *simplex*

Line is that of $A + 6B = 1900$ — such a line always intersects the maximum at a *vertex*, i.e. an intersection of some constraints

3.2 Simplex Method

1. Find any feasible vertex v
2. Examine all neighbouring vertices of v, $u_1, \ldots, u_k$
3. Calculate $f(u_1), \ldots, f(u_k)$
   - If $f(v)$ is the maximum among its neighbours, stop and return v
   - Otherwise, choose *any one* of the neighbouring vertices u_i where $f(u_i) > f(v)$
   - Repeat using vertex u_i

(To find a vertex, we mathematically find the intersection of constraints) (To find generate a vertex's neighbours, we can drop one constraint and put in another)

Why is it the maximum if the algorithm stops in step 3?

- The geometry of the simplex is convex, and the objective function is linear $\rightarrow$ if $f(v)$ is maximum amongst neighbours, then it is maximum in entire feasible region

Why will it always terminate?

- Will have explored all vertices at some point
- Worst case running time: $O(m^n)$ (number of vertices with n variables and m constraints is $\binom{m}{n}$)

3.3 LP Solvers

Solving LP in Windows: Turn on Excel's 'Solver Add-in'; Run 'Data -> Solver' to see the answers

Solving LP in Ubuntu: Use `lp_solve`

3.4 General LP Form

General form: maximise $c^T x$ where $Ax \leq b$ and $x \geq 0$

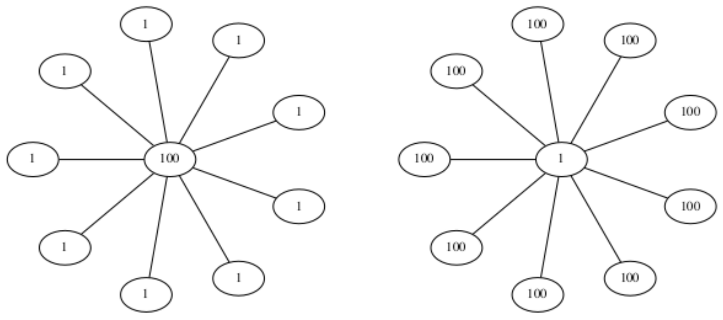
1. Set of variables $x_1, \ldots, x_n$
2. Linear objective to maximise, $c^T x$ — c and x are vectors, so $c^T x$ is dot product
3. Set of linear constraints written as matrix equation, $Ax \leq b$

Some manipulations to put it into standard LP form:

- Convert min to max problem by using negatives and flipping inequality sign direction

4 Minimum-Weight Vertex Cover

In MWVC, each vertex comes with a given weight.



4.1 MWVC as Integer Linear Program (ILP)

We'll show that ILP is NP-hard by reducing MWVC to it.

Let x_j be a 0/1 variable, where 0 means not in VC and 1 means in VC.

$$min(\sum_{j=1}^{n} w(v_j) \cdot x_j) \text{ where:}$$

$$\begin{aligned} x_i + x_j &\geq 1 \text{ for all } (i,j) \in E \\ x_j &\geq 0 \text{ for all } j \in V \\ x_j &\leq 1 \text{ for all } j \in V \\ x_j &\in \mathbb{Z} \text{ for all } j \in V \end{aligned}$$

4.2 MWVC as (Relaxed) Linear Program

Relax the integer constraint: now x_j isn't strictly 0/1, but any real number in between

- Now MWVC solution using LP can be more optimal than ILP (smaller cost), i.e. $OPT(G) = OPT(ILP) \geq OPT(LP)$

$$min(\sum_{j=1}^{n} w(v_j) \cdot x_j) \text{ where:}$$

$$\begin{aligned} x_i + x_j &\geq 1 \text{ for all } (i,j) \in E \\ x_j &\geq 0 \text{ for all } j \in V \\ x_j &\leq 1 \text{ for all } j \in V \end{aligned}$$

Approximation: after solving LP, round up x_i if value ≥ 0.5

- Rounded answer $\leq 2 \times OPT(LP)$
- Proof (???)

4.3 MWVC Deterministic 2-opt Algorithm

Bar-Yehuda and Even's algorithm

- Idea: for each edge, decrease the weight of each of its nodes by the minimum of both weights; final vertex cover is all nodes with final weight = 0

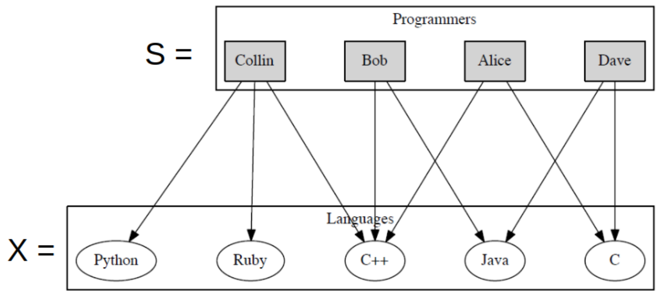
5 Minimum Set Cover

Set cover: A set cover I of subsets $S_1, S_2, \ldots, S_m$ of $X = \{x_1, x_2, \ldots, x_n\}$ such that $\bigcup_{j \in m} S_j = X$

- Minimum set cover: a set cover I of minimum size (fewest number of subsets)

Minimum vertex cover and minimum dominating set can be reduced to minimum set cover, in polynomial time

Example: how to find the smallest set of programmers in S, who will cover all possible languages in X?



5.1 Greedy Minimum Set Cover

Algorithm

- Add sets greedily, one at a time, until everything is covered
- At each step, choose the next set that covers the most as-yet uncovered elements
- $O(\log n)$ approximation algorithm

Analysis

Let $x_1 \ldots x_n$ be the greedy order of selection of elements, let $c(j)$ be the number of elements covered at the same time, let $cost(x_j) = \frac{1}{c(j)}$ (such that cost of covering all elements in the same set is 1).

Suppose we have covered $x_1 \ldots x_{j-1}$, and want to cover the remaining $x_j \ldots x_n$, which has $(n - j + 1)$ elements. Let OPT be the cost of covering these remaining elements.

Then $OPT \geq \frac{n-j+1}{c(j)} = (n - j + 1)cost(x_j)$, so $cost(x_j) \leq \frac{OPT}{n-j+1}$.

Let $|I|$ be the size of the set cover of the greedy algorithm. Then:

$$\begin{aligned}
|I| &= \sum_{j=1}^{n} cost(x_j) \\
&\leq \sum_{j=1}^{n} \frac{OPT}{(n - j + 1)} \\
&\leq OPT \sum_{i=1}^{n} \frac{1}{i} \\
&\leq OPT(\ln n + O(1))
\end{aligned}$$

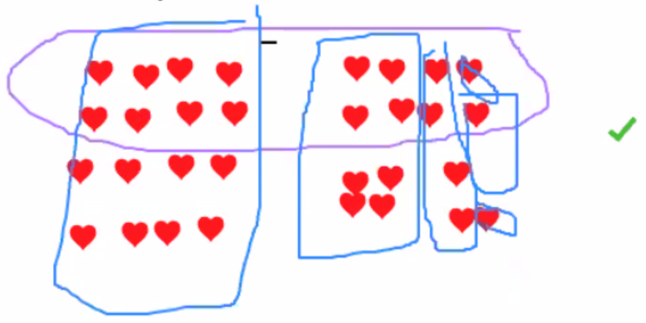
So the greedy set cover is at most $O(\log n)$ times optimal.

Example Question

Q1. `Min-Set-Cover` Instance (6 marks)

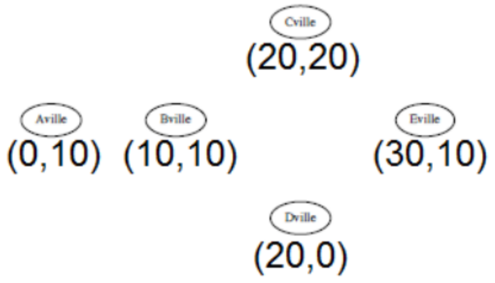
Let's assume that we have a set X with $n = 32$. To simplify the question, assume that $X = \{1, 2, 3, \ldots, 30, 31, 32\}$. Create a `Min-Set-Cover` instance with that X and your chosen set $S = \{S_1, S_2, \ldots, S_m\}$ (The number of subsets m is up to you) so that your test case makes the $O(\ln n)$-approximation `Greedy-Set-Cover` as discussed in class produces a solution that requires $\lfloor \ln 32 \rfloor = \lfloor \log_e 32 \rfloor = \lfloor 3.46 \rfloor = 3$ times more subsets than the optimal answer.

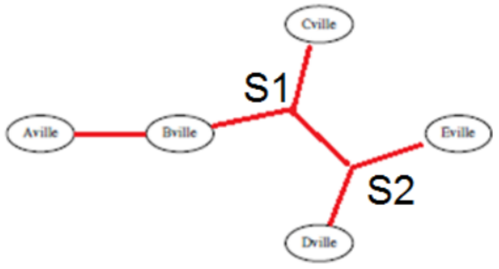
The "easiest" question in Midterm Test S1 AY 2018/19



6 Steiner Tree

Motivation: MST using only existing vertices can be expensive; you can add additional vertices (*Steiner points*) to reduce the total cost!





All 3 versions of the Steiner Tree problem (Euclidean, Metric, General) are *NP-hard*.

Still, Euclidean is the hardest, General is easier, and Metric is the easiest.

6.1 Euclidean Steiner Tree

Let R be a set of points in the Euclidean plane. Find a set of Steiner points S and a minimum spanning tree $T = (R \cup S, E)$ (that minimizes weight), whereby edge weight is given by $d(u, v) = |u - v|$, i.e. Euclidean distance.

($\star$) In an optimal solution, there are at most $(n-2)$ Steiner points. Each Steiner point has degree 3, forming 120 degree angles

Problem

- Given: set of required vertices R
- Find: set S of Steiner vertices, minimum spanning tree $T = (R \cup S, E)$, whereby $d(u, v) = |u - v|$

6.2 Metric Steiner Tree

Generalise the Euclidean Steiner Tree, but the distance function $d(u, v)$ can be *any* metric function!

($\star$) The set of possible Steiner vertices S is *given*.

Metric function properties

- Non-negativity: $d(u, v) \geq 0$
- Identity: $d(u, u) = 0$
- Symmetric: $d(u, v) = d(v, u)$
- Triangle inequality: $d(u, v) + d(v, w) \geq d(u, w)$

Problem

- Given: set of required vertices R, set of Steiner vertices S, and distance metric $d : (R \cup S) \times (R \cup S) \to \mathbb{R}$

- Find: subset $S' \subset S$ of Steiner vertices, minimum spanning tree $T = (R \cup S', E)$

6.3 General Steiner Tree

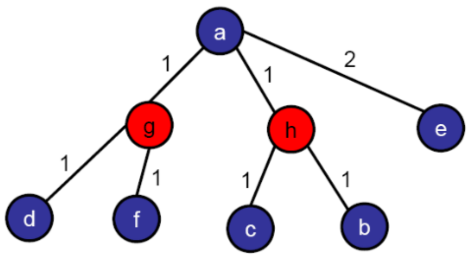
Generalise the Metric Steiner Tree, but d does not need to be a distance metric.

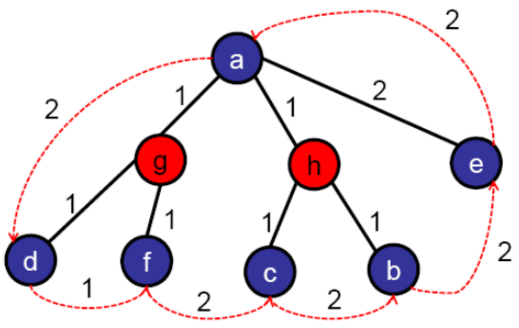
Problem

- Given: set of required vertices R, set of Steiner vertices S, and graph $G = (V, E)$ whereby each edge $(u, v) \in E$ has a weight $d(u, v)$
- Find: subset $S' \subset S$ of Steiner vertices, minimum spanning tree $T = (R \cup S', E)$

6.4 Metric-ST Approximation

MST is a 2-approximation algorithm for Metric-ST.





1. Begin with the optimal Steiner tree T, which can have some Steiner vertices.
2. Using DFS traversal, generate a cycle C of T, whose length is $cost(C) = 2 \times cost(T)$.
3. Now we can bypass the Steiner vertices to create a cycle C'. Note that the cost won't increase (because of triangle inequality), so $cost(C') \leq 2 \times cost(T)$.
4. Now we can remove duplicate vertices to create C'', then break any one edge in the cycle to create a spanning tree T'. So $cost(T') \leq cost(C'') \leq cost(C') \leq 2 \times cost(T)$.

Therefore, MST has cost $\leq 2 \times cost(T)$, the optimal Steiner tree.

6.5 General-ST Approximation

Idea: reduce General-ST to Metric-ST, solve Metric-ST using approximation, convert Metric-ST solution to General-ST solution

Metric completion

- Use APSP (e.g. $O(V^3)$ Floyd-Warshall's) to make non-metric edge weights into metric ones (make the edge weights the shortest path lengths)

Reconstruction

- Assume $d(i,j) > d(i,k) + d(k,j)$
    - So APSP shortens $d(i,j)$ into $d(i,k) + d(k,j)$
- If the 'virtual' edge (i,j) is taken in the Metric-ST version, then take edge (i,k) and (k,j) in the General-ST version
- Some edges may overlap and create cycles, so just take the MST of that

7 Travelling Salesman Problem (TSP)

TSP: find a minimum cost cycle that visits all points.

Formally, given vertices V and a non-negative distance function $d : V \times V \to \mathbb{R}$ (complete graph!), find a cycle $C = (e_1, \ldots, e_n)$ of minimum cost that contains all points in V, where cost of cycle C is $\sum_{e \in C} d(e)$

Brute force: runs in $O(N!)$ time (there are $N!$ permutations)

Dynamic programming: runs in $O(N^2 \cdot 2^{N-1})$ time — if memoize parts of the subtours (remember the best way to complete the remaining part of the tour)

7.1 TSP Variants

- M (Metric) vs G (General): Triangle inequality
- R (Repeats-OK) vs NR (No-Repeats)

	Repeats	**No Repeats**
Metric	M-R-TSP	M-NR-TSP
General	G-R-TSP	G-NR-TSP

All of are *NP-hard*

- NR version of TSP is NP-hard: Hamiltonian cycle reduces to it
- 3 of 4 variants are equivalent and have approximations, except for G-NR-TSP (even approximations are NP-hard)

M-R-TSP $\leftrightarrow$ M-NR-TSP

- NR to R: trivial, no change (all NR solutions are also legal R solutions)
  - $OPT(R) \leq OPT(NR)$
- R to NR:
  - For all repeated vertices, take the 'shortcut' by cutting out the repeated vertex, which cannot increase cost (by triangle inequality)
  - So that will give a better/equal solution
  - $OPT(NR) \leq OPT(R)$
- Hence $OPT(R) = OPT(NR)$
- For c-approximation algorithms — cycles produced in one approximation algorithm can be converted to the other, cost of cycle $d(C) \leq c \cdot OPT(R) = c \cdot OPT(NR)$

M-R-TSP $\leftrightarrow$ G-R-TSP

- G to M: trivial, no change (because if we have a general approx algorithm, it will work for metric as well, which is more restrictive)
- M to G: perform metric completion on general graph by running APSP, transforming it to a metric problem, so it works on a metric algorithm
  - General instance (V, d_g) transformed to metric instance (V, d_m)
  - Run c-approx algorithm A on the metric instance (V, d_m) to produce cycle C
  - Then undo process by replacing each edge (u, v) in C with shortest path from u to v in the original graph, producing cycle C'
  - If C' contains repeats, it's fine because it allows for repeats

  - $d(C') = d(C) \leq c \cdot OPT(V, d_m)$, while $OPT(V, d_m) \leq OPT(V, d_g)$, so $d(C') \leq c \cdot OPT(V, d_g)$

(???)

[Stopped at 12:58PM]

8 Max-Flow and Min-Cut (NOT NP-hard)

8.1 Max-Flow

Flow

Input:

- Directed graph $G = (V, E)$
- Edge capacities $c(e)$
- Source s and target t (or sink)

Output:

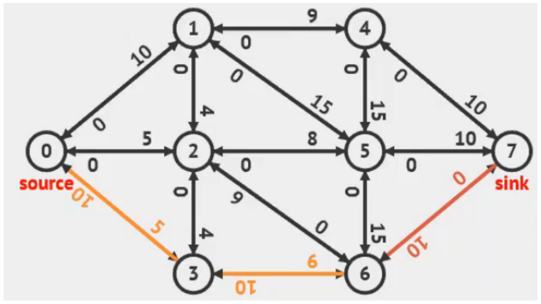
- Assignment of flow $f(e)$ to each edge, with these constraints:
  - $0 \leq f(e) \leq c(e)$
  - For each vertex apart from s and t, flow-in $=$ flow-out (equilibrium constraint)
  - Unidirectional, no cycle with net flow
- Value of st-flow, $value(f) = \sum_{v:(s,v)\in E} f(s,v) = \sum_{v:(v,t)\in E} f(v,t)$

Max flow: Find an st-flow with maximum value

8.2 Ford-Fulkerson Algorithm

(Requires integer capacities)

1. Construct a bidirectional *residual graph*, i.e. edges indicate how much flow can still be increased.
2. Initially, there is no flow, so set all forward capacities to their max, and backward capacities to 0.
3. While there exists one, find any *augmenting path*, a path from s to t with residual capacity.
   - Decrease the forward capacity by the bottleneck.
   - Increase the backward capacity by the bottleneck.



How to find an augmenting path: use any $O(V + E)$ graph traversal algorithm to find any path from s to t

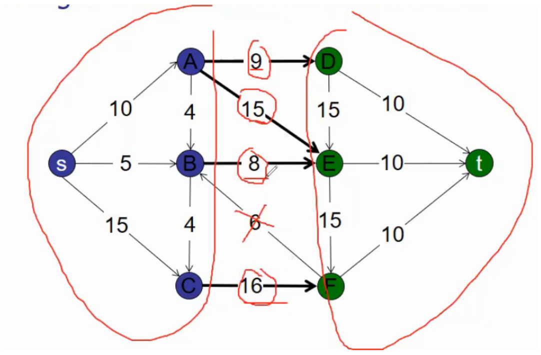
Always terminates: duh, if capacities are integers

Always finds the max flow: no augmenting path exists, so the flow is maximum (by augmenting path theorem below; a simple proof by contradiction)

8.3 Cuts and Flows

Cut: An st-cut partitions vertices into two disjoint sets S and T, where $s \in S$ and $t \in T$

Capacity of cut: The *capacity* of an st-cut is the sum of capacities of the edges that *cross* the cut from $S \to T$. (NOT $T \to S$!)



Net flow across cut: The *net flow* across an st-cut is the sum of flows from $S \to T$, minus the sum of flows from $T \to S$. (i.e. forward-flows - back-flows)

Equality of cuts and flows

- Let f be a flow
- Then for all possible st-cuts, $net\ flow\ across\ cut(S, T) = value(f)$, no matter how you partition the vertices into S and T!

Proof by induction:

- Base case: $S = \{source\}$
- Inductive step: add any node X reachable from S to S.
  - Since in-flow to X = out-flow from X

Weak duality of cuts and flows

Weak duality: Let f be a flow, and (S, T) be an st-cut. Then $value(f) \leq capacity(S, T)$

- Proof: $value(f) = flow\ across\ cut(S, T) \leq capacity(S, T)$

($\star$) Max flow–min cut theorem

For a max flow f, and (S, T) an st-cut with minimum capacity, $value(f) = capacity(S, T)$

Augmenting path theorem

Any maximum flow f has no augmenting paths in the residual graph (proof below)

Equivalent statements

The following statements are equivalent:

1. There exists a (minimum) st-cut whose capacity is the value of f
2. f is a maximum flow
3. There is no augmenting path with respect to f

$1 \to 2$: There exists an f capacity cut $\to$ f is maximum

- Assume (S, T) is an st-cut with minimum capacity equal to f
- For all flows g, $value(g) \leq capacity(S, T)$ (weak duality)
- For all flows g, $value(g) \leq value(f)$
- f is a maximum flow

$2 \to 3$: f is a maximum flow $\to$ there are no augmenting paths (augmenting path theorem)

- Assume otherwise, that there is some augmenting path

- Then flow can be increased by sending along augmenting path
- So f is not a maximum flow, contradiction

$3 \to 1$: no augmenting paths $\to$ there exists an f capacity cut

- Suppose there is no augmenting path
- Let S be vertices reachable from source in residual graph; let T be remaining vertices
- This st-cut has capacity f:
  - All $S \to T$ are saturated
  - All $T \to S$ are empty
  - So $value(f) = net\ flow(S, T) = capacity(S, T)$

8.4 Min-Cut

Cut: value of (S, T) cut is sum of edges from $S \to T$, such that removing them disconnects S and T

Min-Cut: cut with minimum value

- By maxflow-mincut theorem, value of min-cut = value of max-flow

8.5 Ford-Fulkerson Analysis: $O(m^2U)$

Ford-Fulkerson is *slow*: $O(m^2U)$

- Each iteration: $O(m)$ to find augmenting path, where $m = |E|$ (we assume $m \geq n$, so ignore $O(n)$ for updating capacities in residual graph)
- Number of iterations: $O(mU)$. Max flow is at most mU, where U is the max capacity among outgoing edges connected to s, m is number of edges in graph
- Total cost: $O(m^2U)$ (this is a pretty gross upper bound)

8.6 Edmonds-Karp Algorithm: $O(m^2n)$

Idea: find the *shortest* augmenting path (e.g. using BFS)!

Runtime: $O(m^2n)$

- Shortest augmenting path length never decreases
- Each iteration: $O(m)$ to find shortest augmenting path
- Number of iterations: at most $m \cdot n$

Proving that shortest augmenting path length never decreases

When we augment, two things will happen:

- We "delete" at least 1 bottleneck edge in residual graph
  - Does not shorten the augmenting path
- We could add backward/reverse edges in residual graph
  - Does not shorten the augmenting path either (why???)

Proving that there are at most $m \cdot n$ iterations

When we augment, every bottleneck edge along augmenting path will be "deleted" from residual graph R.

$(\star)$ Show that each of the m edges in R can become the bottleneck edge at most $\frac{n}{2}$ times

Let A and B be 2 vertices connected by an edge in R.

- If (A, B) becomes bottleneck edge for the first time, $dist(s, B) = dist(s, A) + 1$
- After augmenting, edge (A, B) is deleted from R
- But (A, B) can reappear if the flow from A to B is decreased, when reverse edge (B, A) appears on some other shortest augmenting path afterwards
- Then $dist(s, A) = dist(s, B) + 1$
- Hence $dist(s, A) = dist(s, B) + 1 \geq \textit{old-dist}(s, B) + 1 = \textit{old-dist}(s, A) + 2$
- So if the bottleneck edge reappears, the distance to the same vertex A must have increased by at least 2
- Since shortest path is at most length n, (A, B) can only be bottleneck $\frac{n}{2}$ times
- Since there are $O(m)$ edges, there are at most $O(mn)$ times you'll encounter bottleneck edges

8.7 Dinic's Algorithm: $O(mn^2)$

- Find level graph with BFS
- For each blocking flow f' in the level graph:
  - Augment the flow by f'

Analysis

- There can only be $O(n)$ level graphs
- For each level graph, we only run BFS once and then DFS max n times (why?)

9 Push-Relabel Algorithm

Previously, we choose a *legal* flow iteratively until we get the max flow

Now, we greedily push as much flow as possible from s, then cut back. We start from possibly *illegal* flows and iteratively make them *legal*

Idea of Push-Relabel:

- Push as much flow as possible from s
- While there exists a vertex with unbalanced flow (flow-in > flow-out)
  - Invariant: No $s \to t$ path in residual graph R (source and sink disconnected)
  - Calculate excess flow in that vertex (flow-in - flow-out)
  - Push some excess flow on an edge in residual graph R

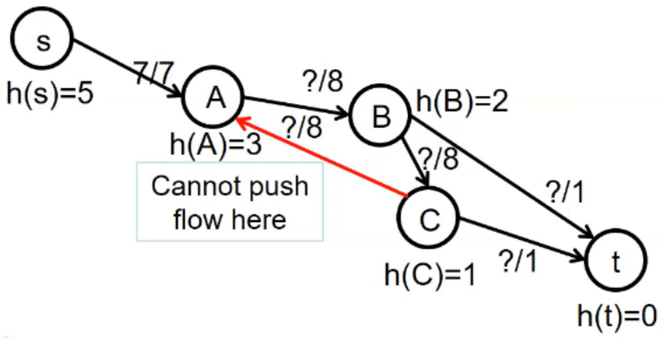
Pre-flow: assignment of flow $f(u, v) \geq 0$ to every edge $(u, v) \in E$ such that:

- We always satisfy capacity constraints: $f(u, v) \leq c(u, v)$ for all $(u, v) \in E$
- Flow-in $\geq$ flow-out: $\sum_z f(z, u) \geq \sum_w f(u, w)$ for all $u \in V - \{t\}$
- Pre-flow is *feasible* if all $x(u) = 0$ for all $u \in V$

Excess: let $x(u) = excess(u) = \sum_z f(z, u) - \sum_w f(u, w)$

Problem: how do we avoid pushing flow in cycles?

- Solution: make an *elevation map.* Assign each vertex a *height*, where you can only push from higher to lower heights $\to$ acyclic DAG



Problem: we can get stuck with excess flow with nowhere to push it if it's acyclic!

- Solution: relabel operation

Relabel: to "raise the height"

Basic Push-Relabel Algorithm

- Heights start at 0, except source which is at height n
- s pushes out as much flow as possible
- While f is not feasible (i.e. exists one vertex u such that $x(u) > 0$):
  - $r(u, v) = c(u, v) - f(u, v) + f(v, u)$
  - If there exists some vertex u (not source or sink) and some vertex v (including source and sink) where u has excess, (u, v) has capacity left and u is higher than v (i.e. $x(u) > 0$, $r(u, v) > 0$, $h(u) > h(v)$):
  - Then push b units of flow from u to v, where $b = min(x(u), r(u, v))$ — the bottleneck

    - Else choose any vertex v with excess, and raise its height by 1 — relabel

PUSH:

- If $b = r(u, v)$, then the push is *saturating*; u becomes balanced only if $r(u, v)$ is also $x(u)$
- If $b \neq r(u, v)$ but $b = x(u)$, then the push is *non-saturating*; u becomes balanced

RELABEL:

- Take a vertex and raise its height by 1

(??? TODO: CARRY ON)

10 Max-Cardinality Bipartite Matching

(Graph matching is NOT NP-hard, it has polynomial time algorithms!)

Matching: a subset M of edges in G such that no two of them meet at a common vertex

Maximum Matching: a matching that contains the largest possible number of edges

Max-Cardinality Bipartite Matching (MCBM): maximum matching with bipartite graph

- Much easier with bipartite graphs

Weighted Matching: mimimize or maximise the weights

10.1 Overview of Matching Problems

Bipartite?

- Greedy? => (Un)weighted MCBM, use greedy bipartite matching
- Non-greedy?
  - Unweighted? => Unweighted MCBM, use augmenting path
  - Weighted? => Weighted MCBM, use min-cost-max-flow or Kuhn-Munkres

Non-Bipartite?

- Small? => (Un)weighted MCM, use DP with bitmask (maybe ~2^{20})
- Big?
  - Unweighted? => Unweighted MCM, use Edmonds' matching (not NP-hard!)
  - Weighted? => Weighted MCM, can be done in polynomial time but it's hard

10.2 Unweighted MCBM

- Augmenting Path: $O(VE)$
- Dinic's Max-Flow: $O(\sqrt{V}E)$
- Hopcroft-Karp: $O(\sqrt{V}E)$
- Augmenting Path++: $O(kE)$

Augmenting Path: $O(nm)$

- Lemma: A matching M is maximum in $G \leftrightarrow$ There is no more augmenting path in G
- Algorithm: For each vertex in the left side:
  - If there exists an augmenting path, flip edges along augmenting path
- Only need one pass through all left vertices!

Ford-Fulkerson: $O(m^2)$ (since $U = 1$)

- Convert to max flow with all directed edges, weight = 1

Dinic's Max-Flow: $O(\sqrt{n}m)$

- Convert to max flow with all directed edges, weight = 1

Bipartite Matching with Capacity, i.e. Assignment Problem

- E.g. each person can take up to 5 modules (not only 1-1 matching, but up to 1-5!); find the maximum number of matchings between people and modules

Hopcroft-Karp: $O(\sqrt{n}m)$

- It's basically doing the same as Dinic's, so just use Dinic (even though this has a slightly lower constant factor)

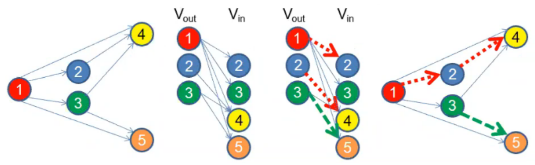
Augmenting Path++: $O(km)$

- Idea: Pre-process the graph by randomly matching each left vertex with another right vertex if possible
- There will only be k unmatched vertices leftover
- Then run the standard augmenting path algorithm above

10.3 Technique: Splitting to V_{in} and V_{out}

Min-Path-Cover: split to v_{in} and v_{out}, then use MCBM!

- Idea: for every matching in MCBM, we "chain" together two paths in Min-Path-Cover => one fewer path



10.4 Weighted MCBM

Min-Cut Max Flow: $O(mn * mCU)$ where C is max cost, U is max capacity

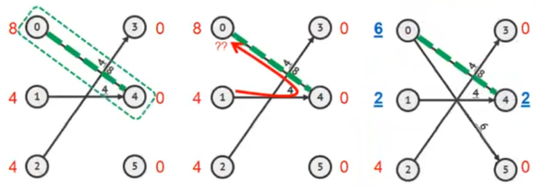
- Not a suggested solution.
- Idea: introduce *cost*. Convert to max flow as usual, all edges with capacity 1.
- Find augmenting paths with the *lowest costs*. Use Bellman-Ford to handle negative costs (e.g. backflows) in the augmenting path

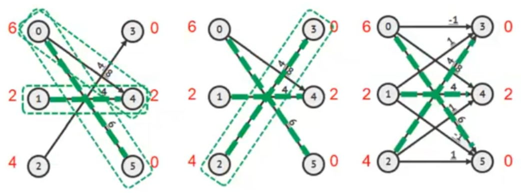
Hungarian Algorithm (Kuhn-Munkres): $O(n^3)$

- By default, it's for a max-weighted perfect bipartite matching;
  - Modify to support *min*-weighted: negate edge weights
  - Modify to support *non-perfect* matching: add dummy vertices and edges with irrelevant (-∞) weights

Equality subgraph, where $label(u) + label(v) == w(u, v)$:

- In the beginning, perform an initial greedy step. From each LHS vertex, pick the outgoing edge with the max weight; this max weight will be the vertex's label.
- If there are 'stuck' augmenting paths: we need to relabel to open up new edges!
- Relabel the equality subgraph: relabel the vertices, which preserves existing edges and adds new edges in the equality subgraph.
  - We do this by decreasing labels of selected LHS vertices, and increasing labels of selected RHS vertices by the same constant





How do we decide where and how much to relabel?

- Try all possible $\ell(u) + \ell(v) - w(u, v)$, for all $u \in S$ and $v \notin T$

Adding dummy edges/vertices

- Result will be $ans + 2 \times -INF =>$ we can set INF to -10^9

11 Maximum-Cardinality Matching (No longer Bipartite)

Graph is NOT bipartite if there exists an odd-length cycle

11.1 Unweighted MCM

Edmonds' Matching++

- First, pair vertices randomly (as many as you can)
- Perform the following as long as there is an unprocessed vertex:
- If there exists an augmenting path: *expand* the blossom, flipping the status of edges along that augmenting path
- Else if there exists an odd-length cycle: *contract* the odd cycle into 1 vertex, and update the graph

Code is hard to implement, just copy lol

11.2 Weighted MCM, Small

DP with Bitmask (max N≈20)

- Bitmask: contains N bits, each bit represents taking each vertex or not

11.3 Weighted MCM, Large

Too bad.

12 Stochastic Local Search

Local search: start at a random position in the search space. Then iteratively move from a position to a local neighbouring position, making small changes along the way. *Typically incomplete*, i.e. not guaranteed to find optimal solutions

Perturbative search: search step — *modify* one or more solution components

- E.g. 2-exchange in a TSP tour

Constructive search: search step — *extend* with one or more solution components

- E.g. greedy nearest neighbour heuristic

Local search sacrifices completeness and optimality, but it has an *any-time property*; and the longer you run it, the better the solution quality or probability

Stochastic search: stochastic means *random*

Some well-known SLS methods:

- Evolutionary/genetic algorithms
- Simulated annealing
- Tabu search

12.1 Definitions

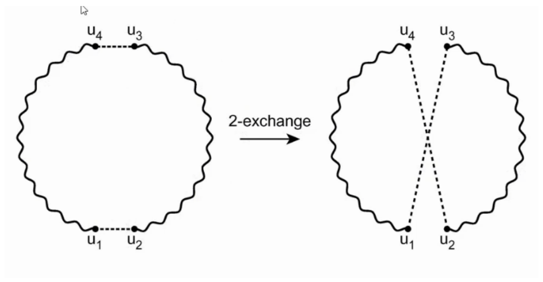
Let π be a problem instance of COP.

- Search space $S(\pi)$: set of all possible instances
- Solution set $S'(\pi) \subseteq S(\pi)$: optimal solutions
- Neighbourhood relation $N(\pi) \subseteq S(\pi) \times S(\pi)$
- Set of memory states $M(\pi)$
- Initialisation function $\to D(S(\pi) \times M(\pi))$
  - Specifies probability distribution over initial search positions and memory states
- Step function $S(\pi) \times M(\pi) \to D(S(\pi) \times M(\pi))$
- Termination function $S(\pi) \times M(\pi) \to D(\{T, F\})$
- Neighbourhood set of candidate solution s, $N(s)$
- Neighbourhood graph of problem instance π, $G_N(\pi)$
- k-exchange neighbourhood
- Search step: pair of search positions s, s' whereby s' can be reached from s in one step
- Search trajectory: finite sequence of search positions
- Search strategy: specified by init and step function
- Evaluation function $g(\pi) : S(\pi) \to R$
- Local minimum: search position whose neighbours are non-improving, i.e. position $s \in S$ such that $g(s) \leq g(s')$ for all $s' \in N(s)$

12.2 Generic SLS algorithm

- *Initialise*
- While it's not time to terminate yet:
  - *Step* function
  - If the new solution is better, take it
- Output your solution

12.3 TSP: 2-exchange



12.4 Hill-Climbing for TSP

Hill-climbing, i.e. iterative improvement/descent

- Init: greedy nearest neighbour heuristic
- Step: randomly choose equally from improving neighbours
- Neighbourhood: 2-exchange neighbourhood
- Terminate: when there's no improving neighbour available (local optimum)

12.5 Escaping Local Optima

Restart: whenever you encounter a local optimum, re-initialize search

- Often costly due to cost of initialization

Non-improving steps: whenever you encounter local optimum, allow selection of candidate solutions with *equal* or *worse* evaluation function value

Intensification vs Diversification

- Intensification: greedily increase solution quality by exploiting evaluation function
- Diversification: avoiding search stagnation by preventing search from getting stuck in confined regions

NOTE:

- Local minimum depends on g and neighbourhood relation N
- Larger neighbourhoods $N(s)$ induces:
  - Neighbourhood graphs with smaller diameter;
  - Fewer local minima
  - But also more costly

12.6 Neighbourhood Pruning

Idea: reduce size of neighbourhoods by excluding neighbours that are unlikely to improve g.

($\star$) Candidate lists for TSP

- Idea: good solutions likely include *short* edges
- Candidate list of vertex v: a limited length k list v's neighbours, sorted by shortest edge first
  - Pre-process to sort each vertex v's k shortest edges
  - Then it's k^2 instead of N^2
- Search steps always involve edges to elements of candidate lists

How to choose improving neighbour in each step?

- Best improvement: this requires evaluating all neighbours

13 Meta-Heuristics and Performance

13.1 Randomised Iterative Improvement (RII)

In each search step, with a fixed probability, perform an *uninformed random walk* step instead of an *iterative improvement* step

While termination condition is not satisfied:

- with probability *wp*:
  - choose a neighbour s' of s uniformly at random
- otherwise:
  - choose a neighbour s' of s such that $g(s') < g(s)$
- $s := s'$

When run for sufficiently long, it is guaranteed to find the optimal solution with high probability

RII is often outperformed by more complex SLS methods

13.2 Probabilistic Iterative Improvement (PII)

Accept worsening steps with probability depending on deterioration in evaluation function value, i.e. the bigger the deterioration, the smaller the probability

Metropolis condition: start with high temperature T, which controls likelihood of accepting worsening steps

- $p(s, s', T) := 1$ if $f(s') \leq f(s)$
- $p(s, s', T) := e^{\frac{f(s)-f(s')}{T}}$ if $f(s') \leq f(s)$

Simulated Annealing (SA)

T decreases according to a *annealing schedule*

SA:

- Determine initial candidate solution s
- Set initial temperature T according to *annealing schedule*
- While termination condition is not satisfied:
  - Probabilistically choose neighbour s' of s using *proposal mechanism*
  - If s satisfies probabilistic *acceptance criterion* (depending on T):
    - $s := s'$
  - Update T according to *annealing schedule*
- Proposal mechanism: often uniform random choice from $N(s)$
- Acceptance criterion: often Metropolis condition

Annealing schedule

- Temperature updating scheme: typically geometric cooling ($T := \alpha * T$), better than linear cooling; usually $\alpha \approx 1$, e.g. 0.99

Implementation details

- Precompute acceptance probabilities!! Use table for (Δ,T): computing exponential function can be expensive

(Simulated Annealing doesn't really work well for TSP)

Tabu Search (TS)

Idea: use aspects of search history (memory M) to escape from local minima

Tabu Search

- Associate tabu attributes with candidate solutions, or usually the solution components
- Forbid steps to search positions recently visited/used by iterative improvement, based on tabu attributes

Tabu Search:

- Determine initial candidate solution s
- While termination condition is not satisfied:
  - Determine set N' of non-tabu neighbours of s
  - Choose a best-improving candidate solution s' in N'
  - Update tabu attributes based on s'
  - $s := s'$
- Idea: remember and *tabu* the last k edges
- Tabu tenure TT: things are declared *tabu* for a fixed number of subsequent search steps
  - TT too low: search stagnates (unable to escape from local minima)
  - TT too high: search becomes ineffective (due to overly restricted search path, admissible neighbourhood is too small)
- Robust Tabu Search:
  - Repeatedly but *randomly* choose TT from a given interval $[lo..hi]$
- Reactive Tabu Search:
  - Dynamically adjust TT during search
  - Start TT to be low at first (explores neighbours); then increase TT to explore neighbours more

Crucial factors for Tabu Search success

- Choice of neighbourhood relation
- Efficient evaluation of candidate solutions (caching and incremental update)
- Setup of Tabu Tenure, what to set as tabu

13.3 Iterated Local Search (ILS)

Idea: 2 types of SLS steps:

- Subsidiary local search (SLS) — intensification
- Perturbation steps — diversification

Perturbation step: e.g. 4-exchange

Acceptance criteria

- Always accept the better of 2 candidate solutions
- Always accept the more recent of 2 candidate solutions

13.4 Population-based SLS methods

Manipulate not just one, but a set of candidate solutions at each time step

Evolutionary Algorithm

Mutation, recombination, selection

Memetic algorithm: apply SLS along with evolutionary algorithm (population of candidate solutions)

TSP recombination:

- How to do this? How to combine fragments from parents?

Selection:

- Probabilistic chances, where probability of selection is proportional to fitness value $g(s)$