COS 226	Algorithms and Data Structures	Fall 2025

Solutions Final

1. **Initialization.**

   Don't forget to do this.

2. **Minimum spanning trees.**

   (a) 0 1 2 4 6 8 11

   (b) 0 6 4 1 2 8 11

3. **Depth-first search.**

   (a) 0 2 8 4 3 1 5 7 6 9

   (b) 8 2 5 6 7 1 9 3 4 0

   (c) 6

   The function-call stack contains the vertices 0, 4, 3, 1, 7, 6, in that order.

4. **Shortest paths.**

   (a)

v	0	1	2	3	4	5	6	7
distTo[v]	∞	83	31	0	110	67	37	38

   (b) 0, 1, 4

5. **Hash tables.**

   (a) 6

   (b) 1

   (c) $\frac{4}{10}$

6. **Kd-trees.**

   (a) left child of $(12, 4)$

   (b) $(3, 12), (5, 6), (7, 15), (9, 5)$

7. **Analysis of algorithms.**

   (a) $\Theta(EV)$

   This is the code for Bellman–Ford. The outer loop iterates $\Theta(V)$ times. With the adjacency-lists representation, the inner two loops take $\Theta(E + V)$ time per iteration, which simplifies to $\Theta(E)$ since all vertices are reachable from s. Thus, the total running time is $\Theta(EV)$.

   (b) $\Theta(V^3)$

   The i and v loops each iterate $\Theta(V)$ times. With the adjacency-matrix representation, the innermost loop takes $\Theta(V)$ time per iteration because it must examine each entry in the row corresponding to vertex v. Hence, the total running time is $\Theta(V^3)$.

8. **Maxflows and mincuts.**

   (a) 15

   The capacity of the cut S^* is the sum of the capacities of edges crossing from S^* to T^*, namely 6 + 9 = 15.

   (b) 15

   By the maxflow–mincut theorem, the value of any maxflow equals the capacity of any mincut. Thus, the value of the maxflow f^* is 15.

   (c) 6

   By flow conservation at vertex B, inflow = outflow: $10 + 0 = 4 + x$. Hence $x = 6$.

   (d) 0

   For any maxflow f^* and mincut (S^*, T^*), every forward edge crossing the cut is full and every backward edge is empty. Since $C \to G$ is a backward edge, its flow must be 0.

   (e) 9

   Same reasoning as in part (d): $G \to H$ is a forward edge crossing the mincut, so its flow equals its capacity, $z = 9$.

9. **Dynamic programming.**

   A B H D L B

10. **Graph algorithms.**

    ● ● ○ ● ●

11. **Randomness.**

    If a coin is heads with probability p and tails with probability $1 - p$, then the expected number of coin flips (trials) needed to get the first heads is $1/p$ (the mean of a geometric random variable with success probability p).

    (a) 2

    When no sites are open, the success probability is $p = 1$. Note that two calls to `uniformInt()` *are performed per trial.*

    (b) $2n^2$

    When only one site is blocked, the success probability is $p = \frac{1}{n^2}$. So, the expected number of trials is n^2, and there are two calls to `uniformInt()` *per trial.*

    (c) n^2

    When only two sites are blocked, the success probability is $p = \frac{2}{n^2}$.

    (d) $\Theta(n^2 \log n)$

    The expected number of calls to `uniformInt()` *is*

    $$2 \times \left(\frac{n^2}{1} + \frac{n^2}{2} + \frac{n^2}{3} + \cdots + \frac{n^2}{n^2} \right)$$

    This simplifies to $\Theta(n^2 \log n)$ via the harmonic-sum approximation.

    (e) $\Omega(2^n)$

    This is a Las Vegas randomized algorithm: the number of trials depends on the results of the calls to `uniformInt()`*. There is no predetermined finite upper bound on this number of trials.*

12. **Expert algorithms.**

D For the multiplicative-weights algorithm to predict 1, the *weighted majority* of the experts must predict 1.

F An expert with 6 mistakes will have *double* the weight of an expert who has made 7 mistakes.

T More than half of the total weight will always be assigned to the perfect experts. Thus, the multiplicative-weights algorithm will never make a mistake.

F In the simplified AdaBoost algorithm the weights of the experts always differ by a power of 2. So, the weights of two experts cannot be 4/16 and 7/16.

T Every time the modified elimination algorithm makes a mistake, at least half of the experts are eliminated. Thus, it will make $\leq \log_2 n$ mistakes.

13. **Intractability.**

■ All problems in **P** are also in **NP**.

□ Problem C might not even be a decision problem, so it need not be in **NP**.

■ All problems in **NP** can be solved in exponential time (e.g., by running the verifier on all possible witnesses).

□ If this were true, it would imply **P** ≠ **NP**, but, we do not know whether this is true.

■ If C poly-time reduces to B, this would imply that B cannot be solved in polynomial time either. Since B is in **NP**, this would imply that **P** ≠ **NP**.

14. **Fattest path.**

(a) The main idea is to run BFS or DFS on a modified digraph that keeps only those edges that could appear on a path of bottleneck capacity at least w.

- Build a new digraph G' from G by keeping only edges of capacity $\geq w$.
- Run BFS or DFS in G' to find an $s \rightsquigarrow t$ path.
- If such a path exists, return it; otherwise, return *null*.

This subroutine runs in $O(E+V)$ time.

(b) The main idea is to combine *binary search* with the subroutine BOTTLENECK-PATH(G, s, t, w) from part (a). Sort the edges by capacity so that $w_1 \leq w_2 \leq \cdots \leq w_E$, and binary search over these edge capacities, at each iteration calling the subroutine from part (a) to determine whether there exists a path of bottleneck capacity greater than (or equal) to a specified value. As a loop invariant, we'll maintain an interval $[lo, hi]$ such that

- there exists an $s \rightsquigarrow t$ path with bottleneck capacity $\geq w_{lo}$; and
- there does not exist an $s \rightsquigarrow t$ path with bottleneck capacity $\geq w_{hi}$.

Here's a more detailed description of the binary search algorithm:

- Call BOTTLENECK-PATH(G, s, t, w_1) and return *null* if it returns *null*.
- Call BOTTLENECK-PATH(G, s, t, w_E) and return the path if returns a path.
- Initialize $lo \leftarrow 1$, $hi \leftarrow E$.
- While $lo \neq hi - 1$:
  - Let $mid \leftarrow \lfloor \frac{lo+hi}{2} \rfloor$.
  - If BOTTLENECK-PATH(G, s, t, w_{mid}) returns *null*, set $hi \leftarrow mid$; otherwise, set $lo \leftarrow mid$.
- Return BOTTLENECK-PATH(G, s, t, w_{lo}).

The overall running time is $O((E+V)\log E)$.

- Sorting the edges by capacity using *mergesort* takes $O(E \log E)$ time.
- The binary search has $O(\log E)$ iterations, each taking $O(E+V)$ time.

Partial-credit solutions. The two partial-credit solutions are similar to the full-credit solution, except that they perform binary search (or sequential search) over the interval $[1, U]$ instead of the sorted array of edge capacities.

15. **Tiger cut.**

(a) We want to model an instance of TIGER-CUT as an instance of MIN-ST-CUT. The main idea is to force the s-side of the mincut to contain all black vertices (including white vertices we decide to recolor black) and the t-side to contain all orange vertices (including white vertices we decide to recolor orange). We do this by adding a virtual source s and target t.

- For each vertex v in G, create a vertex v' in G'.
- Create a source vertex s and target vertex t in G'.
- For each black vertex v in G, add an edge $s \to v'$ in G' with capacity ∞ (or $E+1$). *This forces every black vertex to lie on the s-side of any mincut.*
- For each orange vertex v in G, add an edge $v' \to t$ in G' with capacity ∞ (or $E+1$). *This forces every orange vertex to lie on the t-side of any mincut.*
- For each edge $v \to w$ in G, add an edge $v' \to w'$ in G' with capacity 1. *If, after recoloring the white vertices, v is black and w is orange, then $v' \to w'$ contributes 1 to the capacity of any mincut.*

Note 1: in some cases, the edge $v' \to w'$ is unnecessary. For example, if v is orange (or w is black), then $v \to w$ can never point from black to orange, so adding $v' \to w'$ does not affect the objective.

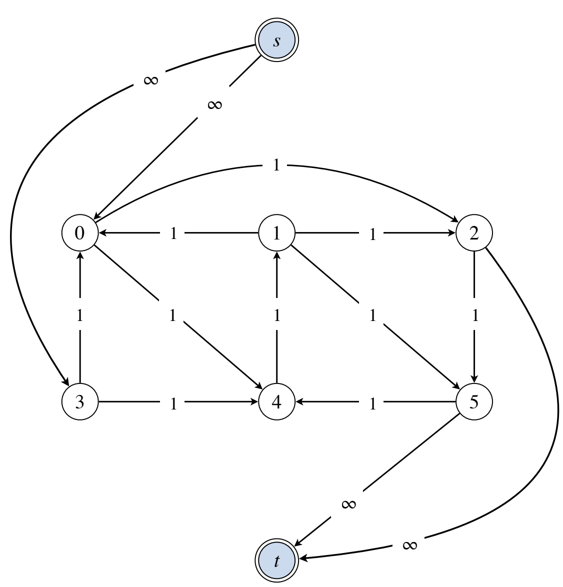
Note 2: if G has no parallel edges, then it suffices to use a capacity of V (instead of $E+1$ or ∞) for edges incident to s or t.

(b) Given a minimum st-cut (S^*, T^*) in G':

- Every vertex in S^* that correspond to a white vertex in G is colored *black*.
- Every vertex in T^* that correspond to a white vertex in G is colored *orange*.

In this coloring, the number of edges that point from black to orange is equal to the capacity of the mincut.

(c)



Here's an alternative solution that removes unnecessary edges. The edge 0→2 could also be removed: it doesn't change the mincut, but it does change the capacity of the mincut.