CS/ECE 374 Sec A ✧ Spring 2025

☙ Homework 8 ❧

Due Wednesday, April 2nd, 2025 at 9am

- You can work in a group of up to **three** students. Read the instructions on the course website for additional details.
- **Submit your solutions electronically on the course Gradescope site as PDF files.** Submit a separate PDF file for each numbered problem. If you plan to typeset your solutions, please use the LaTeX solution template on the course web site. If you must submit scanned handwritten solutions, please use a black pen on blank white paper and a high-quality scanner app (or an actual scanner, not just a phone camera).
- Late submissions will be accepted (for 75% credit) until midnight the day of the deadline.

☞ Some important course policies ☜

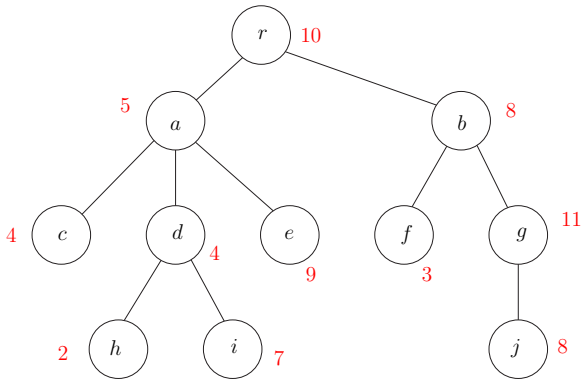
- **You may use any source at your disposal**—paper, electronic, or human—but you ***must*** cite ***every*** source that you use, and you ***must*** write everything yourself in your own words. See the academic integrity policies on the course web site for more details including the policy on using AI tools.
- **Avoid the Two Deadly Sins!** Any homework or exam solution that breaks any of the following rules will be given an ***automatic zero***, unless the solution is otherwise perfect. Yes, we really mean it. We're not trying to be scary or petty (Honest!), but we do want to break a few common bad habits that seriously impede mastery of the course material.
  - Always give complete solutions, not just examples.
  - Always declare all your variables, in English. In particular, always describe the specific problem your algorithm is supposed to solve.

See the course web site for more information.

If you have any questions about these policies,
please don't hesitate to ask in class, in office hours, or on Ed.

---

1. Given a graph $G = (V, E)$, a *vertex cover* of G is a subset $S \subseteq V$ of vertices such that for each edge $e = (u, v)$ in G, u or v is in S. That is, the vertices in S *cover* all the edges. Given G and non-negative weights $w(v)$ on the vertices the Min-Weight Vertex Cover problem is to find a vertex cover S of G with the smallest total weight. The weight of S is the sum of the weights of vertices in S. The Min-Weight Vertex Cover problem is an NP-Hard problem. However it is polynomial-time solvable in trees and we saw this on the lab. For illustration, in the tree below, $\{a, d, b, j\}$ is a vertex cover while $\{r, a, f, g\}$ is not a vertex cover.



   (a) (3 pts) Consider the following generalization of the Min-Weight Vertex Cover problem on trees. Given T, weights $w(v), v \in V$, and an integer k describe an efficient algorithm that finds the weight of a minimum weight vertex cover that has at most k vertices in it. For example if the tree is a path with three vertices with weights $1, 4, 1$ respectively then the min-weight vertex cover has weight 2 (we take the first and third vertices) while it is 4 if we require $k = 1$ since we are forced to take the middle vertex.

   (b) This second part is about *counting* vertex covers in a tree.

      i. (4 pts) Given a tree $T = (V, E)$ describe an efficient algorithm to *count* the number of distinct vertex covers of T. Two vertex covers S_1 and S_2 are distinct if they are not identical as sets of vertices.

      ii. (2 pts) Write a recurrence for the exact number of vertex covers in a path on n nodes. For the base case of a single node tree the answer is 2 since an empty set is also a valid vertex cover for a single node. Use the recurrence to find an exact solution to the recurrence by relating it to the Fibonacci sequence. Would the answer for a path with $n = 500$ fit in a 64-bit integer word? Briefly justify your answer.

      iii. (1 pt) How would you implement your counting algorithm more carefully to run on a 64 bit machine? Accounting for this more careful implementation, what is the running time of your algorithm?

2. We consider a token moving game on a graph. Let $G = (V, E)$ be a directed graph. Each vertex v in G is assigned a label from alphabet $\Sigma = \{0, 1, 2, \ldots, 9\}$; we use $\ell(v)$ to denote the label of v. We think of moving a token starting at a vertex s and moving it along the edges one step at a time. This defines a walk in the graph. Each time a token is moved into a vertex v along an incoming edge it emits the label $\ell(v)$. This naturally defines a string

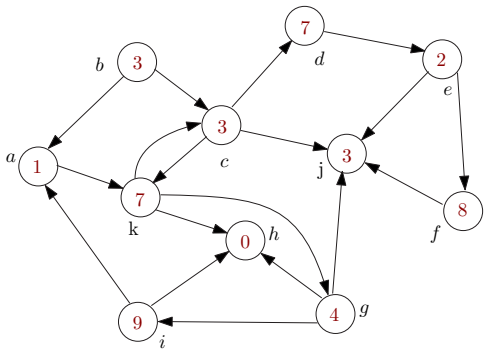


Figure 1. Graph with labels from $\{0, 1, \ldots, 9\}$. The walk a, k, c, k, g, i corresponds to the string 73749.

$\sigma(W)$ with each walk $W = v_1, v_2, \ldots, v_k$ where $\sigma(W) = \ell(v_2)\ell(v_3)\ldots\ell(v_k)$. If $k = 1$ we have $\sigma(W) = \varepsilon$.

(a) (5 pts) Describe an efficient algorithm that given $G = (V, E)$, the labeling function $\ell : V \to \{0, 1, \ldots, 9\}$ and two vertices s, t decides whether there is an s-t walk W such that $\sigma(W)$ contains the substring 374.

(b) (1 pt) Given G, ℓ and a vertex t, describe an efficient algorithm that outputs the set of all vertices $v \in V$ such that there is a v-t walk W_v with the property that $\sigma(W_v)$ contains the substring 374.

(c) (4 pts) Now you have a slightly different game with three distinct tokens A, B, C starting at distinct vertices s_1, s_2, s_3. At each time step all three tokens have to move along an edge and no two tokens can be at the same vertex. If the three tokens (A, B, C) move into vertices whose respective labels are $(1, 7, 3)$ or $(3, 7, 4)$, the corresponding string (173 or 374) is printed. Our goal is to move the tokens from the starting position (s_1, s_2, s_3) to some destination position (t_1, t_2, t_3) in such a way that the exact string 173374 is printed. Describe an efficient algorithm that given G, the labels, the starting positions and the destination positions, checks whether this is possible.

(d) **Not to submit:** What if the graph is undirected? How will your constructions change if the labels are on the edges?

3. **Not to submit:** You've been hired by UIUC to help ensure all the university's buildings are networked together! The university has already built network connections between some of the buildings, and needs your help to finish connecting everything.

(a) (5 pts) Design an algorithm to determine the *number* of new connections the university will have to build to ensure every building is on a single network. Your algorithm should take as input a list of n buildings and m pairs of buildings that already have a direct connection between them. What is the run time of your algorithm in terms of n and m?

(b) (5 pts) Turns out the university wasn't super enthused by your suggestion to run a wire all the way from Siebel to the ARC, so they've now provided you with a list of k "acceptable" connections they're willing to build. Design an algorithm to determine the *number* of acceptable connections the university will have to build to ensure every building is on the same network. (If it is not possible to do this, your algorithm should output "not possible".) What is the run time of your algorithm in terms of n, m, and k?

4. **Not to submit:** This question is about cycles in graphs.

   - Describe a linear time algorithm that given a *directed* graph $G = (V, E)$ and a node $s \in V$ outputs a directed cycle containing s if there is at least one, or correctly states that there is no directed cycle containing s.
   - Describe a linear time algorithm that given an *undirected* graph $G = (V, E)$ and a node $s \in V$ outputs a cycle containing s if there is at least one, or correctly states that there is no cycle containing s.
   - Describe a linear-time algorithm that given a *directed* graph outputs all the nodes in G that are contained in some cycle. More formally you want to output

$$S = \{v \in V \mid \text{there is some cycle in } G \text{ that contains v}\}.$$

5. **Not to submit:** Several problems can be solved via graph modeling. GPS 8 gives you some examples. Solve the following from Jeff Erickson's chapter on basic graph algorithms.

   - Problem 18.
   - Problem 20.
   - Problem 26, part (a).

Solved Problems

4. Professor McClane takes you out to a lake and hands you three empty jars. Each jar holds a positive integer number of gallons; the capacities of the three jars may or may not be different. The professor then demands that you put exactly k gallons of water into one of the jars (which one doesn't matter), for some integer k, using only the following operations:

   (a) Fill a jar with water from the lake until the jar is full.

   (b) Empty a jar of water by pouring water into the lake.

   (c) Pour water from one jar to another, until either the first jar is empty or the second jar is full, whichever happens first.

   For example, suppose your jars hold 6, 10, and 15 gallons. Then you can put 13 gallons of water into the third jar in six steps:

   - Fill the third jar from the lake.
   - Fill the first jar from the third jar. (Now the third jar holds 9 gallons.)
   - Empty the first jar into the lake.
   - Fill the second jar from the lake.
   - Fill the first jar from the second jar. (Now the second jar holds 4 gallons.)
   - Empty the second jar into the third jar.

   Describe and analyze an efficient algorithm that either finds the smallest number of operations that leave exactly k gallons in any jar, or reports correctly that obtaining exactly k gallons of water is impossible. Your input consists of the capacities of the three jars and the positive integer k. For example, given the four numbers $6, 10, 15$ and 13 as input, your algorithm should return the number 6 (for the sequence of operations listed above).

   Solution: Let A, B, C denote the capacities of the three jars. We reduce the problem to breadth-first search in the following directed graph:

   - $V = \big\{(a, b, c) \;\big|\; 0 \le a \le p \text{ and } 0 \le b \le B \text{ and } 0 \le c \le C\big\}$. Each vertex corresponds to a possible ***configuration*** of water in the three jars. There are $(A{+}1)(B{+}1)(C{+}1) = O(ABC)$ vertices altogether.
   - The graph has a directed edge $(a, b, c){\to}(a', b'c')$ whenever it is possible to move from the first configuration to the second in one step. Specifically, there is an edge from (a, b, c) to each of the following vertices (except those already equal to (a, b, c)):
     - $(0, b, c)$ and $(a, 0, c)$ and $(a, b, 0)$ — dumping a jar into the lake
     - (A, b, c) and (a, B, c) and (a, b, C) — filling a jar from the lake
     - $\left\{\begin{array}{ll}(0, a+b, c) & \text{if } a+b \le B \\ (a+b-B, B, c) & \text{if } a+b \ge B\end{array}\right\}$ — pouring from the first jar into the second
     - $\left\{\begin{array}{ll}(0, b, a+c) & \text{if } a+c \le C \\ (a+c-C, b, C) & \text{if } a+c \ge C\end{array}\right\}$ — pouring from the first jar into the third
     - $\left\{\begin{array}{ll}(a+b, 0, c) & \text{if } a+b \le A \\ (A, a+b-A, c) & \text{if } a+b \ge A\end{array}\right\}$ — pouring from the second jar into the first

- $\left\{\begin{array}{ll}(a, 0, b+c) & \text{if } b+c \le C \\ (a, b+c-C, C) & \text{if } b+c \ge C\end{array}\right\}$ — pouring from the second jar into the third
- $\left\{\begin{array}{ll}(a+c, b, 0) & \text{if } a+c \le A \\ (A, b, a+c-A) & \text{if } a+c \ge A\end{array}\right\}$ — pouring from the third jar into the first
- $\left\{\begin{array}{ll}(a, b+c, 0) & \text{if } b+c \le B \\ (a, B, b+c-B) & \text{if } b+c \ge B\end{array}\right\}$ — pouring from the third jar into the second

Since each vertex has at most 12 outgoing edges, there are at most $12(A+1) \times (B+1)(C+1) = O(ABC)$ edges altogether.

To solve the jars problem, we need to find the ***shortest path*** in G from the start vertex $(0, 0, 0)$ to any target vertex of the form $(k, \cdot, \cdot)$ or $(\cdot, k, \cdot)$ or $(\cdot, \cdot, k)$. We can compute this shortest path by calling ***breadth-first search*** starting at $(0, 0, 0)$, and then examining every target vertex by brute force. If BFS does not visit any target vertex, we report that no legal sequence of moves exists. Otherwise, we find the target vertex closest to $(0, 0, 0)$ and trace its parent pointers back to $(0, 0, 0)$ to determine the shortest sequence of moves. The resulting algorithm runs in $O(V + E) = \boldsymbol{O(ABC)}$ ***time***.

We can make this algorithm faster by observing that every move either leaves at least one jar empty or leaves at least one jar full. Thus, we only need vertices (a, b, c) where either $a = 0$ or $b = 0$ or $c = 0$ or $a = A$ or $b = B$ or $c = C$; no other vertices are reachable from $(0, 0, 0)$. The number of non-redundant vertices and edges is $O(AB + BC + AC)$. Thus, if we only construct and search the relevant portion of G, the algorithm runs in $\boldsymbol{O(AB + BC + AC)}$ ***time***. ■

Rubric (for graph reduction problems): 10 points:

- 2 for correct vertices
- 2 for correct edges
  - –½ for forgetting "directed"
- 2 for stating the correct problem (shortest paths)
  - – "Breadth-first search" is not a problem; it's an algorithm.
- 2 points for correctly applying the correct algorithm (breadth-first search)
  - –1 for using Dijkstra instead of BFS
- 2 points for time analysis in terms of the input parameters.
- Max 8 points for $O(ABC)$ time; scale partial credit