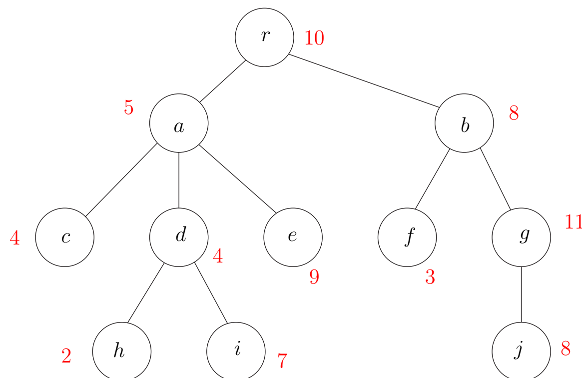
1. Given a graph $G = (V, E)$, a *vertex cover* of G is a subset $S \subseteq V$ of vertices such that for each edge $e = (u, v)$ in G, u or v is in S. That is, the vertices in S *cover* all the edges. Given G and non-negative weights $w(v)$ on the vertices the Min-Weight Vertex Cover problem is to find 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.

Solution: We will solve it by dynamic programming. We assume that the tree T is rooted and let T_u denote the subtree of T rooted at a given node u. For each node u we let $C(u)$ denote the children of u.

For $u \in V$ and $b \in \{0, 1\}$ and $j \in \{0, 1, \ldots, k\}$ we define MinWtVC(u, b, j) as follows: if $b = 1$ it is the minimum weight of a vertex cover of T_u with at most j nodes in which u is included and if $b = 0$ it is the minimum weight of a vertex cover of T_u with at most j nodes. For the base cases if u is a leaf then MinWtVC$(u, 0, j) = 0$ for all $j \geq 0$ and MinWtVC$(u, 1, j) = w(u)$ if $j \geq 1$ and ∞ if $j = 0$. We use ∞ symbolically but we can replace it with a number W that is bigger than the sum of all the vertex weights.

$$\text{MinWtVC}(u, b, j) = \begin{cases} 0 & \text{if } u \text{ is a leaf and } b = 0 \\ w(u) & \text{if } u \text{ is a leaf and } j \geq 1 \text{ and } b = 1 \\ \infty & u \text{ is a leaf, } j = 0,\ b = 1 \end{cases}$$

For the recursive case when u is not a leaf we consider $b = 1$ and $b = 0$ separately. We let $n_u = |C(u)|$ be the number of children of u and we let $C(u, h)$ denote the h'th child of u. If u is included then all the edges incident to u are covered and the number of vertices available to the children of u is $j - 1$ and we need to split the budget among the children. We will try all possible ways to split the

budget over the children and take the minimum.

$$\text{MinWtVC}(u,1,j) = \begin{cases} \infty & \text{if } j = 0 \\ w(u) + \min_{j_1,\ldots,j_{n_u} \geq 0, \sum_{h=1}^{n_u} j_h \leq j-1} \sum_{h=1}^{n_u} \text{MinWtVC}(C(u,h),0,j_h) & \text{otherwise} \end{cases}$$

Now we consider the case $b = 0$. If u is not included then all the children of u have to be included.

$$\text{MinWtVC}(u,0,j) = \begin{cases} \infty & \text{if } j < n_u \\ \min\{\text{MinWtVC}(u,1,j), \min_{j_1,\ldots,j_{n_u} \geq 1, \sum_{h=1}^{n_u} j_h \leq j} \sum_{h=1}^{n_u} \text{MinWtVC}(C(u,h),1,j_h)\} & \\ & \text{otherwise} \end{cases}$$

The return value is MinWtVC$(r,0,k)$ where r is the root of the tree.

We can use a 2-d array of size 2n (two values for each vertex u) to memoize the recursion and use the standard post-order traversal to process the nodes. How long does it take to evaluate MinWtVC$(u,0,j)$ and MinWtVC$(u,1,j)$ based on the definition? The naive way is to do a brute force approach and try all possible ways to split j into n_u non-negative numbers such that their sum is at most j. There are at most j^{n_u} such combinations. This will be exponential if n_u is unbounded. However if each node has at most 2 children then this is at most j^2 and since $j \leq k$ we can evaluate MinWtVC$(u,0,j)$ and MinWtVC$(u,1,j)$ in $O(k^2)$ time. Thus the total run time is $O(nk^2)$ assuming that the tree is binary.

Avoiding the assumption: We will now describe an algorithm that avoids the binary tree assumption. This amounts to doing a Knapsack style DP at each node u to compute MinWtVC(u,b,j) from the values at the children. This requires some additional information. We will also assume that the children of each node are ordered from left to right. For a given node u we use leftsib(u) to denote the sibling of u that is to its left. If there is no left sibling it will be 0. We let T'_u denote the forest consisting of all the trees rooted at u and its siblings to its left. We define an auxiliary function MinWtVCSib$(u,0,j)$ for the min weight of a vertex cover for T'_u with a total of j vertices allowed. Similarly we define an auxiliary function MinWtVCSib$(u,1,j)$ which is the min weight of a vertex cover for T'_u in which u and its siblings to the left have to be included. We write recursive functions to compute MinWtVCSib values once the MinWtVC values for the children of a node are already known.

$$\text{MinWtVCSib}(u,b,j) = \begin{cases} \text{MinWtVC}(u,b,j) & \text{if leftsib}(j) = 0 \\ \min_{0 \leq j' \leq j}(\text{MinWtVCSib}(\text{leftsib}(u),b,j') + \text{MinWtVC}(u,b,j-j')) & \text{otherwise} \end{cases}$$

To compute MinWtVC(u, b, j) for a node which is not a leaf it suffices to know the MinWtVCSib$(C(u, n_u), b, j)$ and MinWtVCSib$(C(u, n_u), b, j-1)$. More formally if u is not a leaf and $C(u, n_u)$ is its rightmost child,

$$\text{MinWtVC}(u, 1, j) = w(u) + \text{MinWtVCSib}(C(u, n_u), 0, j-1)$$

and

$$\text{MinWtVC}(u, 0, j) = \min\{\text{MinWtVC}(u, 1, j), \text{MinWtVCSib}(C(u, n_u), 1, j)\}$$

Thus, it suffices to explain how to compute MinWtVCSib values. To compute MinWtVCSib(u, b, j) requires trying all possible values of j' between 0 and $j-1$ and hence the total computation is $O(j)$ which is $O(k)$ since $j \le k$. There are a total of $O(nk)$ quantities to compute and each takes $O(k)$ time, and thus the total time is $O(nk^2)$. We order the computation as follows. We process nodes in post-order traversal. Leaves are easy. When processing a non-leaf node u we first compute MinWtVC(u, b, j) for all j via MinWtVCSib$(C(u, n_u), b, j)$ values in $O(1)$ time. Then we compute MinWtVCSib(u, b, j) for each j via the values stored in the left sibling of u in $O(k)$ time. ■

Rubric: Standard DP rubric scaled appropriately. Full credit for a solution that works for the binary tree case in $O(nk^2)$ time. Extra credit of 3 pts for a solution that works in $O(nk^2)$ time for the general case.

(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.

Solution: We assume that the tree T is rooted and let T_u denote the subtree of T rooted at a given node u. For $u \in V$ and $b \in \{0, 1\}$ we define NumVC(u, b) as follows. NumVC$(u, 0)$ denotes the number of distinct vertex covers of T_u. NumVC$(u, 1)$ denotes the number of vertex covers of T_u that include u.

We give a recursive definition of NumVC(u, b). The base cases are when u is a leaf, in which case NumVC$(u, 0) = 2$ (corresponding to $\emptyset$ and $\{u\}$) and NumVC$(u, 1) = 1$. For the recursive case, if we have to include u in the solution, then we are not required to include u's children, but if $b = 0$ and we decide to not include u in the solution, then we are required to include all of u's children. The number of vertex covers in T_u is the sum of those which include u and those that do not include u. Let $C(u)$ denote the set of u's children in the tree. We have

$$\text{NumVC}(u, b) = \begin{cases} 2 & \text{if } u \text{ is a leaf and } b = 0 \\ 1 & \text{if } u \text{ is a leaf and } b = 1 \\ \prod_{v \in C(u)} \text{NumVC}(v, 0) & \text{if } b = 1 \\ \prod_{v \in C(u)} \text{NumVC}(v, 0) + \prod_{v \in C(u)} \text{NumVC}(v, 1) & \text{otherwise.} \end{cases}$$

Finally, the solution can be computed by calling NumVC$(r, 0)$.

Now, the memoization data structure will be a $2d$-array of size $2n$, since there are n vertices but b can only take two values. Suppose VC is an array that, at position $VC[u, b]$ stores the value NumVC(u, b). Like before, NumVC(u, b) depends only on values of NumVC for vertices that are in the subtree T_u, and thus a correct evaluation order will be an ordering generated by a post-order traversal of the tree. We can fill in the array VC in this ordering with a simple for-loop. The running time of the algorithm is $O(n)$ assuming multiplication and addition take $O(1)$ time. This is because each entry NumVC(u, b) is filled once and participates only in a constant number of multiplications for its parent. ■

Rubric: Standard DP rubric scaled appropriately.

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.

Solution: Let $P_n = v_1, v_2, \ldots, v_n$ be a path on n nodes. Let $T(n)$ be the number of vertex covers of a P_n. We have $T(1) = 2$ corresponding to the empty set and the single vertex. We can also easily see that $T(2) = 3$ corresponding the three sets $\{v_1\}, \{v_2\}, \{v_1, v_2\}$. We write the following recurrence for $T(n)$ when $n \geq 3$.

$$T(n) = T(n-1) + T(n-2)$$

We justify it as follows. Imagine rooting the path at v_n. We count the vertex covers as the sum of those that contain v_n and those that do not contain v_n. If we include v_n then the edge (v_{n-1}, v_n) is covered by v_n and any vertex cover of P_{n-1} together with v_n is a vertex cover of P_n. Thus the number of such vertex covers is the number of vertex covers of P_{n-1} which, by induction, is $T(n-1)$. If we do not include v_n then we need to include v_{n-1}, and by the same logic the number of vertex covers that include v_{n-1} in P_{n-1} is $T(n-2)$.

We see that the recurrence is the same as the one for Fib except for the base cases. Recall $Fib(0) = Fib(1) = 1$ and hence $Fib(2) = 2$ and $Fib(3) = 3$. Thus $T(n)$ is simply the shifted version of Fib with $T(n) = Fib(n+1)$ for $n \geq 1$.

We say in lecture that $Fib(n) = \frac{\phi^n - (1-\phi)^n}{\sqrt{5}}$ where $\phi = (1+\sqrt{5})/2$. Thus $T(500) = Fib(501) \geq ((1+\sqrt{5})/2)^{501}/\sqrt{5} \geq (1.61)^{501}/\sqrt{5}$. Thus the number of bits required is at least $501 \log_2 1.61 - (\log_2 5)/2 \geq 342$. Thus the answer would not fit in 64 bits. ■

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?

Solution: We can use a big-integer package that explicitly keeps track of numbers which can grow. In our setting the maximum number of digits required is $\leq n$ since the number of vertex covers of a n node graph is at most 2^n. Thus we can explicitly use n digits for each of the quantities we compute in the counting algorithm. Now we have to account for the time to add and multiply n digit numbers. We have $O(n)$ multiplications and additions and hence the total time is $O(nT_{mult}(n,n))$ where $T_{mult}(n,n)$ is the time to multiply two n-digit numbers. A naive bound on $T_{mult}(n,n)$ is $O(n^2)$ but we saw Karatsuba's algorithm that takes $O(n^{\ln_2 3})$. There are even faster algorithms with the currently best known taking $O(n \log n)$ time. ■

Rubric: 0.75pts for expressing the run-time in terms of the number of multiplications in part (i) times the time to multiply two n digit numbers. 0.25 points for explaining how many digits the numbers require.

Standard dynamic programming rubric. 10 points =

- 3 points for a clear and correct English description of the recursive function you are trying to evaluate. (Otherwise, we don't even know what you're *trying* to do.)
  - $-$ 1 for naming the function "OPT" or "DP" or any single letter.
  - No credit if the description is inconsistent with the recurrence.
  - No credit if the description does not explicitly describe how the function value depends on the named input parameters.
  - No credit if the description refers to internal states of the eventual dynamic programming algorithm, like "the current index" or "the best score so far". The function must have a well-defined value that depends *only* on its input parameters (and constant global variables).
  - An English explanation of the *recurrence* or *algorithm* does not qualify. We want a description of *what* your function returns; points for an explanation of *how* that value is computed are assigned in other items.
- 4 points for a correct recurrence, described either using mathematical notation or as pseudocode for a recursive algorithm.
  - $+$ 1 for base case(s). $-$½ for one *minor* bug, like a typo or an off-by-one error.
  - $+$ 3 for recursive case(s). $-$1 for each *minor* bug, like a typo or an off-by-one error.
  - $-$ 2 for greedy optimizations without proof, even if they are correct.
  - **No credit for iterative details if the recursive case(s) are incorrect.**
- 3 points for iterative details
  - $+$ 1 for describing an appropriate memoization data structure. **Hash tables are NOT an appropriate memoization data structure!**
  - $+$ 1 for describing a correct evaluation order; a clear picture is usually sufficient. If you use nested for loops, be sure to specify the nesting order. (In particular, if you draw a rectangle for a 2d array, be sure to label and direct the row and column indices.)
  - $+$ 1 for correct time analysis. (It is not necessary to state a space bound.)

- For problems that ask for an algorithm that computes an optimal *structure*—such as a subset, partition, subsequence, or tree—an algorithm that computes only the *value* or *cost* of the optimal structure is sufficient for full credit, unless the problem specifically says otherwise.
- Iterative pseudocode is ***not*** required for full credit, provided the other details of your solution are clear and correct. We usually give two official solutions, one with pseudocode and one without.

  If your solution does includes iterative pseudocode, you do not need to separately describe the recurrence, memoization structure, or evaluation order. But you ***do*** still need and English description of the underlying recursive function (or equivalently, the contents of the memoization structure). ***Perfectly correct iterative pseudocode, with no explanation or time analysis, is worth at most 6 points out of 10***, or at most 5 points out of 10 if you name the memoization array "DP".
- Partial credit for incomplete solutions depends on the running time of the ***best possible*** completion (up to the target running time). For example, consider a solution that contains *only* a clear English description of a function, with no recurrence or iterative details. If the described function *can* be developed into an algorithm with the target running time, the solution is worth 3 points; however, if the function leads to an algorithm that is slower than the target time by a factor of n, the solution could be worth only 2 points (= 70% of 3, rounded).

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 $\sigma(W)$ 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 $\sigma(W) = \varepsilon$.

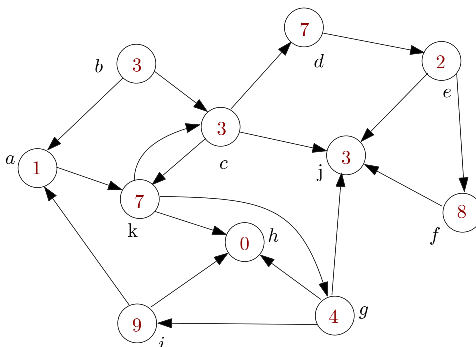


Figure 1. Graph with labels from $\{0, 1, \ldots, 9\}$. The walk a, k, c, k, g, i corresponds to the string 73749.

(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.

Solution: We will use graph layering. We create a new graph $G' = (V', E')$ with 4 layers. We set $V' = V \times \{\epsilon, 3, 7, 4\}$. We define the edges as a series of sets we will add to E':

- Add the edge set $\{((u, \epsilon), (v, \epsilon)) \mid (u, v) \in E\}$ to E'. This makes the ϵ layer a copy of G.
- Add the edge set $\{((u, \epsilon), (v, 3) \mid (u, v) \in E, \ell(v) = 3\}$ to E'. We have the option to enter the 3 layer if there is an edge to a node with label 3. This begins our tracking of the substring 374.
- Add the edge set $\{((u, 3), (v, 7)) \mid (u, v) \in E, \ell(v) = 7\}$ to E'. Once we are in the 3 layer, we should only be able to take edges to nodes with a label of 7.
- Add the edge set $\{((u, 7), (v, 4)) \mid (u, v) \in E, \ell(v) = 4\}$ to E'. Likewise with the 7 to 4 layer.
- Add the edge set $\{((u, 4), (v, 4)) \mid (u, v) \in E\}$ to E'. This makes the 4 layer another copy of G, since we have traversed a walk with the substring 374 we can traverse the rest of the graph freely.

Let $n = |V|$ and $m = |E|$. We can see that our new graph made $4n$ vertices and *at most* $4m$ edges, so our graph size is $O(n + m)$. G' can be constructed in linear time, as each edge set inserted into E' is made by doing a linear scan of E. Thus, the construction of G' takes $O(n + m)$ time.

Claim 1. *There is a path from (s, ϵ) to $(t, 4)$ in G' iff there is an s-t walk in G such that $\sigma(G)$ contains the substring 374.*

If there is a path W in G where $\sigma(W)$ contains the substring 374, then there must be 3 nodes with labels 3,7,4, which are adjacent in said order, where the node with label 3 is reachable from s and the node with label 4 can reach t. Our graph G' is constructed in such a way that some node with a label 3 can be traversed if its reachable from s, and we must travel to nodes with labels 7 and 4 with no other nodes available. Once we reach a node $(u, 4)$, we can traverse the graph to see if t is reachable.

If there is a path W in G' from (s, ϵ) to $(t, 4)$, this implies that there is a route in G that can take 3 nodes consecutively with labels $3, 7, 4$, reachable from s and able to reach t. Thus there is a s-t path W' in G such that $\sigma(W)$ contains the substring 374.

Thus, we construct G' as above, and run WFS starting from (s, ϵ) and check if $(t, 4)$ is reachable. This takes $O(m + n)$ time with respect to the graph size, which G' is of size $O(m + n)$. ■

Rubric: Standard graph modeling rubric scaled appropriately.

(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.

Solution: We use the exact same construction as G' in the previous part, but reverse all the edges. We run WFS from $(t, 4)$ and mark any node of the form (u, ϵ). We return the set of marked nodes.

By reversing the edges we can locate what nodes can reach $(t, 4)$, this still takes $O(n + m)$ time, as G' is of size $(n + m)$ and we use a single WFS call from $(t, 4)$. Correctness follows from the previous part. ■

Rubric: 0.5 for making sure the graph from part (i) is correctly used for this case or adapted appropriately since some solutions may use a graph in part (i) for a specific source.

0.5 for using right algorithm and making sure to use the reverse of the graph.

(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.

Solution: We will use graph modeling again. Our goal is to both keep track of the current configuration of the tokens on the graph, and whether or not we've printed 173 or 374 yet. We create a new graph $G' = (V', E')$. We set $V' = V \times V \times V \times \{0, 1, 2\}$. For simplicity, $\ell(v_1, v_2, v_3)$ gives us the concatenation of all three labels. To simplify edge creation, we will remove some nodes in V' ahead of time:

- Remove any (u_1, u_2, u_3, i) for $i \in \{0, 1, 2\}, u_1, u_2, u_3 \in V$ where u_1, u_2, u_3 are not distinct. This enforces that no two tokens land on the same vertex.
- Remove any $(u_1, u_2, u_3, 0)$ where $\ell(u_1, u_2, u_3) = 374$. This enforces that we cannot print 374 before 173
- Remove any $(u_1, u_2, u_3, 1)$ where $\ell(u_1, u_2, u_3) = 173$. This enforces that we cannot print 173 again.
- Remove any $(u_1, u_2, u_3, 2)$ where $\ell(u_1, u_2, u_3) \in \{173, 374\}$. This enforces that we cannot print 173 or 374 again.

We define the edges that we will add to E':

- For every pair of nodes $(u_1, u_2, u_3, 0), (v_1, v_2, v_3, 0) \in V'$, we add an edge between them, if there is an edge between each pair of u_i, v_i in E and $\ell(u_1, u_2, u_3) \neq 173$.
- For every pair of nodes $(u_1, u_2, u_3, 1), (v_1, v_2, v_3, 1) \in V'$, we add an edge between them, if there is an edge between each pair of u_i, v_i in E and $\ell(u_1, u_2, u_3) \neq 374$
- For every pair of nodes $(u_1, u_2, u_3, 2), (v_1, v_2, v_3, 2) \in V'$, we add an edge between them, if there is an edge between each pair of u_i, v_i in E.

- For every pair of nodes $(u_1, u_2, u_3, 0), (v_1, v_2, v_3, 1) \in V'$, we add an edge between them if there is an edge between each pair of u_i, v_i in E and $\ell(u_1, u_2, u_3) = 173$.
- For every pair of nodes $(u_1, u_2, u_3, 1), (v_1, v_2, v_3, 2) \in V'$, we add an edge between them if there is an edge between each pair of u_i, v_i in E and $\ell(u_1, u_2, u_3) = 374$.

The first three conditions create edges which cover the cases of traversing within each layer without printing 173 or 374. The last two conditions cover traversing between layers if we have entered a 173 or 374 configuration. Specifically, if our starting configuration is 173 or 374, then the only move allowed is to move into the next layer.

The product construction of vertices makes $O(n^3)$ vertices. For each triple of vertices, we consider a singular edge to traverse to the next node, and there's only a constant number of cases which generate edges. There's $O(n^3)$ triples, so at worst we make at most $O(m^3)$ edges between them. Thus, our graph size and construction time is $O(n^3 + m^3)$.

Claim 1. *There is a path from $(s_1, s_2, s_3, 0)$ to $(t_1, t_2, t_3, 2)$ in G' iff there is a walk that prints exactly 173374 in G with the given starting and ending conditions.*

Following our construction of G', we can see that our moves become restricted once we print 173 or 173374, and we can determine if such a walk will print this string starting from $(s_1, s_2, s_3, 0)$ to $(t_1, t_2, t_3, 2)$.

Our algorithm is then:

i. Construct G' as above
ii. Run WFS from $(s_1, s_2, s_3, 0)$ and see if $(t_1, t_2, t_3, 2)$ is reachable.

The construction time is $O(n^3 + m^3)$ vertices, and WFS takes $O(|V| + |E|)$ time with respect to size of the input graph, which gives us a total runtime of $O(n^3 + m^3)$.

■

Rubric: Standard graph modeling rubric scaled appropriately.

Standard graph-reduction rubric. 10 points =

- \+ 3 for constructing the correct graph.
  - +1 for correct vertices
  - +1 for correct edges
  - −½ for forgetting "directed" if the graph is directed
  - +1 for correct weights, costs, labels, or other annotations, if any
  - ○ The vertices, edges, and so on must be described as explicit functions of the input data.
  - ○ For most problems, the graph can be constructed in linear time by brute force; in this common case, no explicit description of the construction algorithm is required. If achieving the target running time requires a more complex algorithm, that algorithm will graded out of 5 points using the appropriate standard rubric, and all other points are cut in half.

+ 3 for explicitly relating the given problem to a specific **problem** involving the constructed graph. For example: "The minimum number of moves is equal to the length of the shortest path in G from the start vertex $(0,0,0)$ any target vertex of the form $(k,\cdot,\cdot)$ or $(\cdot,k,\cdot)$ or $(\cdot,\cdot,k)$." or "There is a French-flag walk from s to t in G if and only if $(s,0)$ can reach $(t,0)$ in H."
  - No points for just writing (for example) "shortest path" or "reachability". Shortest path in which graph, from which vertex to which other vertex? How does that shortest path relate to the original problem?
  - No points for only naming the algorithm, not the problem. "Breadth-first search" is not a problem!
+ 2 for correctly applying the correct **algorithm** to solve the stated problem. (For example, "Perform a single breadth-first search in H from $(0,0,0)$ and then examine every target vertex." or "Whatever-first search in H.")
  - -1 for using a slower algorithm than necessary, for example, Dijkstra's algorithm instead of breadth-first search.
  - -1 for explaining an algorithm from lecture or the textbook instead of just invoking it as a black box.
+ 2 for time analysis in terms of the *input* parameters (not just the number of vertices and edges of the constructed graph).

★ An extremely common mistake for this type of problem is to attempt to modify a standard algorithm and apply that modification to the input data, instead of modifying the input data and invoking a standard algorithm as a black box. This strategy can work in principle, but it is much harder to do it correctly, and it is terrible software engineering practice. ***Clearly correct*** solutions using this strategy will be given full credit, but partial credit will be given only sparingly. Really, just do it the other way.