25 (100 PTS.) GraphCarts

GraphCarts is a new open-world virtual reality racing game. A race course in GraphCarts is modeled as a directed graph G, where each road segment is associated with a distance $w(u, v) > 0$ that describes the amount time it takes to get from intersection u to intersection v. Since the graph is directed, some roads are one-way. Even among two-way roads, it may be slower to go in one direction than another, i.e. it is possible that $w(u, v) \neq w(v, u)$. You start at a designated intersection s, and the game ends as soon as you reach the destination intersection t. By "open-world," what the developers of GraphCarts mean is that you are welcome to travel through the same intersection or along the same road segment multiple times.

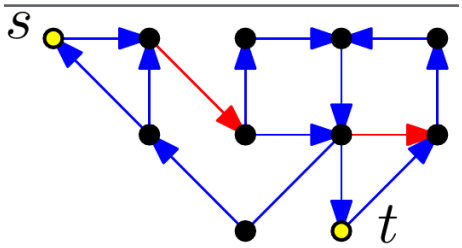
The main challenge in GraphCarts is that some of the road segments contain *checkpoints* ($\varphi(u, v) = 1$ if there is a road segment with a checkpoint on it from intersection u to intersection v, $\varphi(u, v) = 0$ otherwise). The rule is that you're required to pass through at least two checkpoints (it can be the same checkpoint multiple times) *before* you reach the destination (reaching the destination without having passed through two checkpoints is an instant loss).

25.A. (25 PTS.) Give an algorithm, as fast as possible, to find the fastest path from s to t in GraphCarts, given as input a directed graph describing the race course. Briefly explain why your algorithm is correct, and analyze its running time.

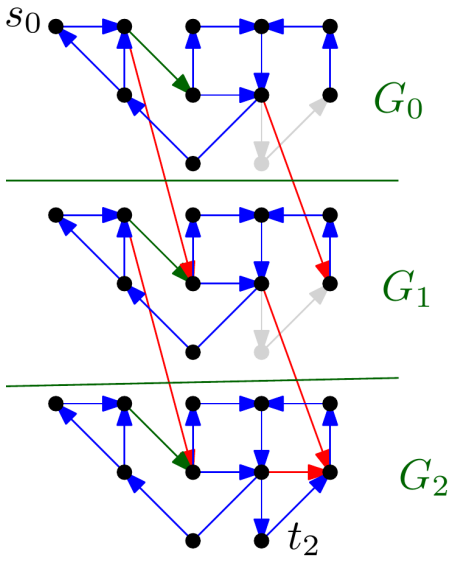
Solution:

We have to pass through a small constant number of checkpoints. We create a new graph G' consisting of three copies of G. For each intersection $u \in \mathsf{V}(G)$, we create three corresponding vertices u_0, u_1, u_2 in the new graph G', where u_i corresponds to being in position u and having passed through at least i checkpoints. For each checkpoint edge $u \to v \in G$ such that $\varphi(u, v) = 1$, we also add an edge $u_i \to v_{i+1}$ to G' with the same weight, for $i = 0$ and $i = 1$. The shortest path from s_0 to t_2 in G' gives us the desired answer, since t_2 corresponds to reaching the finish line having passed through at least 2 checkpoints. Since reaching the destination without having passed through two checkpoints is an instant loss, we should make sure t_0 or t_1 are removed from G'. As an example:

Input G:



"Tower" graph G':



The edges are non-negative, so Dijkstra is applicable here. The resulting cost is $O(n' \log n' + m') = O(n \log n + m)$, where $n = |\mathsf{V}(G)|$, $m = |\mathsf{E}(G)|$, $n' = |\mathsf{V}(G')|$ and $m' = |\mathsf{E}(G')|$. Indeed $|\mathsf{V}(G')| = 3n - 2$, and $\mathsf{E}(G') \leq 5m$.

25.B. (25 PTS.) In the downloadable content for GraphCarts, the player gets one free "Time Freeze" powerup that lets them traverse a single road segment in 0 time. This power-up can only be used once, so for example if they use it to travel along the road segment $u \to v$ in 0 time, and then they travel $u \to v$ again, the second takes the usual amount of time $w(u, v) > 0$. The rule about needing to pass through a checkpoint twice still applies. Modify your algorithm to find the fastest path from s to t under this new rule. Briefly explain why your algorithm is correct, and analyze its running time.

Solution:

We construct a new graph G'', which consists of two copies of the graph G', where $u_0 \in G''$ corresponds to state $u \in G'$ before having used the Time Freeze powerup, and u_1 corresponds to state $u \in G'$ after having used the powerup. For every edge $u \to v \in G'$, we add an edge $u_0 \to v_1 \in G''$ that has zero weight. Now the shortest path from s_0 to t_1 gives the desired answer. The running time is unchanged.

25.C. (50 PTS.) In the second downloadable content for GraphCarts, the player gets k free "Negative Time" powerup (you can assume $k < n$). A single such powerup lets them traverse a road segment in -5 time (they are allowed to use several powerups on the same edge – each such time is counted separately). They are allowed to use up to k powerups during the game.

Fortunately, the developers removed the checkpoint rule that everybody hated so much from this version of the game. Just get from s to t as fast as you can, no need to pass through any checkpoints.

Design an algorithm, as fast as possible, to compute the fastest path from s to t under this new rule. Briefly explain why your algorithm is correct, and analyze its running time.

Solution:

The expected solution. We create a tower graph with $k+1$ copies of G, with the powerup edges connecting two consecutive levels. Formally, let $V' = \left\{(v, i) \;\middle|\; v \in V, i \in [\![k]\!]\right\}$, where $[\![k]\!] = \{0, 1, \ldots, k\}$. Similarly, we define the inner level edges set E_i, for $i \in [\![k]\!]$,. to be

$$\mathsf{E}_i = \{(u, i) \to (v, i) \mid (u \to v) \in \mathsf{E}(\mathsf{G})\}.$$

We assign these edges their original weight $w\big((u, i) \to (v, i)\big) = w(u \to v)$. Similarly, we define cross "floor" edges

$$\mathsf{E}_{i,i+1} = \{(u, i) \to (v, i + 1) \mid (u \to v) \in \mathsf{E}(\mathsf{G})\},$$

for $i = 0, \ldots, k-1$. Such edges corresponds to a powerup, and as such has weight $w\big((u, i) \to (v, i + 1)\big) = -5$.

Let $\mathsf{H} = (\mathsf{V}(\mathsf{G}) \times [\![k]\!], \bigcup_i \mathsf{E}_i \cup \bigcup_i \mathsf{E}_{i,i+1})$ be the resulting "tower" graph.

Clearly, the resulting graph H has negative weight edges, but no negative cycles. It has $O(nk)$ vertices, and $O(km)$ edges. As such, one can use Bellman-Ford to compute the

shortest path from $(s,0)$ to all the vertices in H. Let $d\big(v,i\big)$ denote the computed distance from s to (v,i), for all v and i.

Clearly, the desired distance is $D(v) = \min_{i \in [\![k]\!]} d(v,i)$. Indeed, a path with $\ell \leq k$ powerups from s to v, corresponds to a path from $(s,0)$ to (v,ℓ) in H.

As such, the desired distances can be computed in $O(|\mathsf{V}(\mathsf{H})| \cdot |\mathsf{E}(\mathsf{H})|) = O(nk \cdot km) = O(k^2 nm)$.

Solution:

A slightly better solution.

Let repeat the above construction of the graph. In the ith iteration, assume we computed the distance from $(s,0)$ to all the vertices (u,i), for all $u \in \mathsf{V}(\mathsf{G})$. For $i = 0$ this just requires running Dijkstra on G. For $i > 0$, we construct the following new graph J_i, with a special source vertex $\widehat{s}_i$, with all the other vertices being $\{(v,i) \mid v \in \mathsf{V}(\mathsf{G})\}$. We set the weight of the edge $w\Big(\widehat{s}_i \to (v,i)\Big)$ to be $d(v,i-1) - 5$. All the other edges of J_i are just the edges of E_i, and they have their original weights. Clearly, computing the shortest path in J_i from $\widehat{s}_i$, corresponds to computing $d(v,i)$, for all $v \in \mathsf{V}(\mathsf{G})$. Using Bellman-Ford this takes $O(nm)$ time, and since we have to repeat it k times, we get overall running time $O(knm)$. Now that we computed $d(v,i)$ correctly for all $v \in \mathsf{V}(\mathsf{G})$ and for all i, we can compute $D(\cdot)$ as descried above. The overall running time is $O(knm)$.

Solution:

A somewhat better solution is possible, by observing the following.

Lemma 9.1. *Let G be a directed graph, with a special vertex s that has only outgoing edges. All the edges in the graph have weights on them, but only the outgoing edges from s might have negative weights. Then one compute the shortest path from s to all the vertices in G, in $O(n \log n + m)$ time, where $n = |\mathsf{V}(\mathsf{G})|$ and $m = |\mathsf{E}(\mathsf{G})|$.*

Proof: Let $\alpha = 1 + \max_{s \to v \in \mathsf{E}(\mathsf{G})} |w(s \to v)|$. Consider the modified weight function on the edges

$$\forall u \to v \in \mathsf{E}(\mathsf{G}) \qquad w'(u \to v) = \begin{cases} w(u \to v) + \alpha & u = s \\ w(u \to v) & \text{otherwise.} \end{cases}$$

For any $u \in \mathsf{V}(\mathsf{G})$, let $d(u)$ and $d'(u)$ be the shortest path in G from s to u, according to the weight function w and w' on the edges, respectively. Since $w'(\cdot) > 0$, one can compute $d'(\cdot)$ in $O(n \log n + m)$ time using Dijkstra. It is however clear that $d(u) = d'(u) - \alpha$, for all $u \neq s$. ∎

Clearly, the graph J_i constructed above has the desired properties, and we can apply the algorithm of the above lemma, as we compute the distances in J_i. This takes $O(n \log n + m)$ time, per "floor" i, and since we are repeating it k times, we get overall running time $O(kn \log n + km)$.

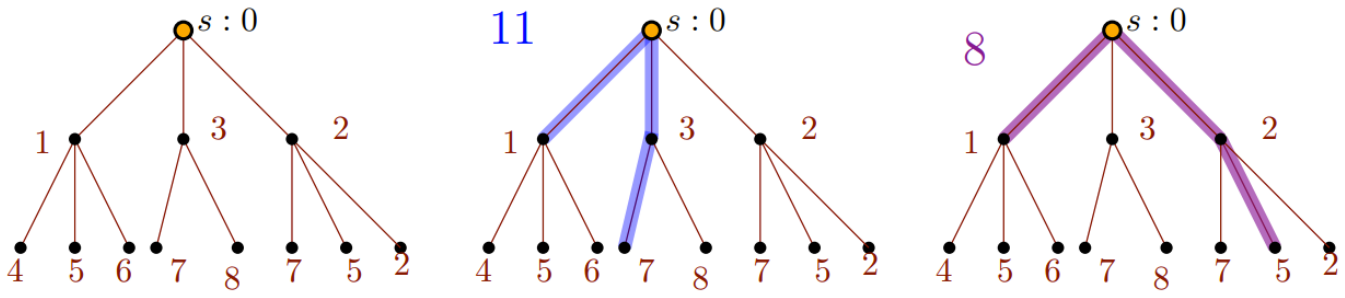
26 (100 PTS.) DynCraft

You're in charge of setting up some gold mining infrastructure within an underground cave system. The cave system can be modeled as a directed tree. It consists of N chambers. The root is the entrance to the cave system, the chamber s. Each chamber is connected to at most 3 subtrees (a ternary rooted tree!).

The quantity of gold in each chamber u is denoted by a function $g(u)$.

To mine for gold and bring it up to the entrance, you need to place conveyor belts in the corridors connecting the cave chambers. You have a limited number of L conveyor belts, $L = O(N)$. If you place a conveyor belt in a corridor connecting u to v, then it can carry gold from v to u towards the entrance. You can mine all the gold from all of the chambers that can reach the entrance through a path of conveyor belts.

The following is an example with three belts, and two different solutions. Here, the root has value 0 associated with it. Here, the first solution has a better value than the second solution.



Give a dynamic programming algorithm that determines the optimal way to place the L conveyor belts in order to mine the largest amount of gold.

Solution:

This is a standard dynamic programming problem on trees.

We can define each subproblem as $F(u, \ell)$, which denotes the amount of gold that can be brought up to chamber u by placing exactly ℓ conveyor belts, one of them has to connect u with its parent.

At each tree, we have to choose how to allocate conveyor belts to the subtrees, that is we have to choose among ℓ_1, ℓ_2, ℓ_3 such that $\ell_1 + \ell_2 + \ell_3 \leq \ell$. There are $O(\ell^2)$ such choices. We get the recurrence:

Assume the vertices of the tree are numbered 1 to N, and $s = 1$. Let $\texttt{NULL} = 0$. Let $\texttt{child}(v, i)$ be a function that returns the ith child of the node v. If no such node exists, it returns $\texttt{NULL}$.

$$F(u,\ell) = \begin{cases} 0 & \ell = 0 \\ 0 & u = \texttt{NULL} \\ g(u) + \max_{\ell_1+\ell_2+\ell_3=\ell-1} \left\{ \sum_{i=1}^{3} F(\texttt{child}(u,i), \ell_i) \right\} & \text{otherwise.} \end{cases}$$

Here is a pseudo-code for solving this using explicit-memoization:

```
V[0...N][0...N+1] ← −∞
compOpt(v, ℓ):
  if ℓ = 0 or v = NULL then
    V[v, ℓ] ← 0
    return 0
  if V[v, ℓ] ≥ 0 then
    return V[v, ℓ]
  β = −∞
  for ℓ1 = 0, ... ℓ − 1 do
    for ℓ2 = 0, ... (ℓ − 1) − ℓ1 do
      ℓ3 = (ℓ − 1) − ℓ1 − ℓ2
      if ℓ3 < 0
        continue
      β1 ← compOpt(child(v, 1), ℓ1)
      β2 ← compOpt(child(v, 2), ℓ2)
      β3 ← compOpt(child(v, 3), ℓ3)
      if β1 + β2 + β3 > β then
        β ← β1 + β2 + β3

  V[v, ℓ] ← β + g(v)
  return β
```

The total number of subproblems is $LN = O(N^2)$. The time to compute each subproblem will be $O(N^2)$ due to searching over $\ell_1 + \ell_2 + \ell_3 = \ell$. As such, the overall running time is

$$O\Big((\#\text{ subproblems}) \cdot (\text{time per subproblem})\Big) = O(N^2 \cdot N^2) = O(N^4).$$

The desired optimal solution value is $\alpha = F(s, L+1)$.

After computing all the optimal values, $V[\cdot, \cdot]$, recovering the optimal solution is easy.

```
recoverOpt(p, v, ℓ):
  if ℓ = 0
    return
  if v = NULL then
    return
  if p ≠ NULL then
    print pv
  for ℓ1 = 0, ... ℓ − 1 do
    for ℓ2 = 0, ... (ℓ − 1) − ℓ1 do
      ℓ3 = (ℓ − 1) − ℓ1 − ℓ2
      if ℓ3 < 0
        continue
      if V[v, ℓ] = g(v) + V[child(v, 1), ℓ1] + V[child(v, 2), ℓ2)] + V[child(v, 3), ℓ3] then
        recoverOpt(v, child(v, 1), ℓ1)
        recoverOpt(v, child(v, 2), ℓ2)
        recoverOpt(v, child(v, 3), ℓ3)
        return
```

We call **recoverOpt**(NULL, v, ℓ) to recover the optimal solution.

A faster solution is possible by rewriting the tree as a binary tree, with new fake edges being "free". The running time then improves to $O(N^3)$. An even faster algorithm should follow by using separator node in the tree, but this is outside the scope for this class (and the analysis looks messy in this case, but it probably leads to a near quadratic time algorithm in this case).

27 (100 PTS.) Max Norm Suppose you have a directed graph with n vertices and m edges, where each edge e has weight $w(e)$, and no two edges have the same weight. Here, you can assume that the graph is sparse, say, $m = O(n \log n)$. For a cycle C with edges $e_1 e_2 \cdots e_\ell$, define the *max-norm* of C to be

$$f(C) = \max\{w(e_1), w(e_2), ..., w(e_\ell)\}/\ell.$$

27.A. (20 PTS.) Let f^* be the maximum max-norm of any closed walk W in the graph. Show that there is a *simple* cycle C such that $f(C) = f^*$. A simple cycle can visit a vertex at most once.

Solution:

Let C be a cycle with the maximum $f(C)$. If there is more than one such optimal cycle, pick the one with the smallest number of edges. Let $e_1, e_2, \ldots, e_\ell, e_1$ be its edge sequence, where without loss of generality e_1 is the edge with the most weight (the cycle is the same if we simply rotate the indexes around). Suppose that a vertex is repeated. Then the end vertex of e_i is equal to the start vertex of e_j for some $j > i + 1$. The cycle $e_1, \ldots, e_i, e_j, e_{j+1}, \ldots, e_\ell, e_1$ has the same maximum edge weight, but fewer nodes, hence it is has a greater *max-norm*, a contradiction.

27.B. (80 PTS.) Give an algorithm that finds the cycle with largest *max-norm* as quickly as possible.

Solution: We compute for every pair of vertices $u, v \in \mathsf{V}(G)$ their *hop distance* $h(u, v)$ – that is, the minimal number of edges in a path from u to v. This can be computed in $O(n(n + m))$ time, by doing **BFS** from each vertex of the graph. Here $h(u, v) = \texttt{NULL}$, if there is not path from u to v, where `NULL` is a special value indicator.
Now, for each edge $u \to v$ in the graph, we compute the candidate value

$$\beta(u, v) = \frac{w(u \to v)}{h(v, u) + 1}.$$

Here if $h(u, v) = \texttt{NULL}$, then we set $\beta(u, v) = \texttt{NULL}$.
The algorithm returns the maximum value $h(u, v)$ computed that is not `NULL`, overall vertices $u, v \in \mathsf{V}(G)$. If all the computed values of $h(\cdot, \cdot)$ are `NULL`, the algorithm prints "Oh no", apologizes, throw a tantrum, and then do a core dump.
The running time of the algorithm is $O(n^2 + nm)$.

Lemma 9.2. *The above algorithm returns the correct value.*

Proof: If there is no cycle in the graph, then the algorithm detects it, and apologizes. Otherwise, observe that any $\beta(u, v)$ that is not `NULL`is a valid value of some cycle. Consider the optimal cycle C, with $e = u \to v$ be its heaviest edge. Clearly, $h(v, u) = |C| - 1$, which readily implies that the algorithm computes the right value. ■