# Lecture 14

## Shortest Path Review

1. We are going to build a DP single source shortest path algorithm as follows (weighted directed graph with negative edges). The dp array **double**[] dists will store the length of the shortest path from the starting node $v$ to vertex $i$ using any walk of length $k$ or less. The algorithm iterates $k = 1, 2, \ldots$ updating dists at each step. Answer the following questions:

   (a) How should we initialize the array dists?

   (b) Explain how to update dists during each iteration.

   (c) When can we stop iterating through $k$-values?

   (d) How can we use the above algorithm to find negative cycles?

2. We are going to build a DP all pairs shortest path algorithm as follows (weighted directed graph with negative edges). The dp array **double**[][] dists will store the length of the shortest path from vertex $i$ to vertex $j$ only using vertices numbered less than $k$. The algorithm iterates $k = 1, 2, \ldots$ updating dists at each step. Answer the following questions:

   (a) How should we initialize the array dists?

   (b) Explain how to update dists during each iteration.

   (c) When can we stop iterating through $k$-values?

   (d) How can we use the above algorithm to find negative cycles?

3. Will Dijkstra's algorithm work on all graphs with negative edges, but no negative cycles?

4. The diameter of a graph is the longest distance between any pair of vertices. Explain how to find the diameter of a graph.

5. Explain how to find the strongly connected components of a graph in $O(V^3)$ time (not the best possible algorithm, but very easy to implement).

## Solutions

1. (a) Set dists[i] = i==v ? 0 : Double.POSITIVE_INFINITY.

   (b) For every neighbor $i$ of every vertex $j$ set dists[i] = min(dists[i], dists[j]+$w_{ij}$) where $w_{ij}$ is the weight of the edge from $i$ to $j$.

(c) When $k = v - 1$, the longest possible path length. Hence the runtime is $O(EV)$.

(d) If another iteration of the algorithm improves dists, then a walk of length $k$ is shorter than a walk of length $k-1$ implying a negative cycle by the pigeonhole principle.

2. (a) Set dists[i][i]=0 for all $i$, dists[i][j]=$w_{ij}$ if there is an edge from $i$ to $j$ with weight $w_{ij}$, or $\infty$ otherwise.

(b) On iteration $k$ set dists[i][j] = min(dists[i][j], dists[i][k]+dists[k][j]).

(c) The last value of $k$ is $V - 1$. Hence the runtime is $O(V^3)$.

(d) Check if dists[i][i]<0 for any $i$.

3. No. Consider a graph with 3 nodes and $w_{01} = 100$, $w_{02} = 2$, $w_{12} = -99$. If you allow Dijkstra to keep revisiting nodes when they are improved, you can get exponential runtimes.

4. Run all pairs shortest path, and take the maximum entry.

5. Run all pairs shortest path, build a union-find data structure, and union any pair such that dists[i][j]<$\infty$ && dists[j][i]<$\infty$.

## DAGs

Almost every DP problem can be thought of as a DAG traversal. For example, the Spreadsheet problem on the homework was possible since there were no cycles in the formulae. There are a few DP problems solvable on DAGs that are purer forms of problems we have already solved. For example, given a DAG count the number of paths from a starting node to an ending node. This can be solved with a simple memoized DFS. Performed on the correct graph, this is the same as computing the number of ways to produce exact change.

Another DAG specific DP problem is to find the longest path between two vertices. This is also solvable using a simple memoized DFS. Both counting paths and longest path are not efficiently solvable on general graphs, but can be solved quickly on DAGs.

## DAG Problems

1. Give an algorithm for counting the number of paths between two given nodes of a DAG.

2. Give an algorithm for finding the length of the longest path between two given nodes of a DAG.

3. ($\star$) Suppose you have a directed graph (not a DAG) with 2 positive integer weights $(w_1, w_2)$ associated with each edge. Give an algorithm to find a path between two given nodes $u, v$ that maximizes the sum of the $w_1$'s used, but keeps the sum of the $w_2$'s used below a given threshold $T$. Here $|E|, T \leq 1000$.

## Solutions

1. The number of paths from $u$ to $v$ is the sum of the number of paths from each of $u$'s neighbors to $v$. Runtime $O(E)$.

```
//Count paths from u to v
long count(ArrayList<Integer>[] adj, int u, int v)
{
  if (u == v) return 1;
  if (cache[u] != -1) return cache[u];
  long ret = 0;
  for (int i = 0; i < adj[u].size(); ++i)
    ret += count(adj,adj[u].get(i),v);
  return cache[u] = ret;
}
```

2. The length of the longest path from $u$ to $v$ is 1 more than the max of the lengths of the longest paths from each of $u$'s neighbors to $v$. Runtime $O(E)$.

```
//Length of longest path from u to v; returns -1 if no path
   exists
int len(ArrayList<Integer>[] adj, int u, int v)
{
  if (u == v) return 0;
  if (cache[u] != -2) return cache[u];
  int max = -1;
  for (int i = 0; i < adj[u].size(); ++i)
  {
    int l = len(adj,adj[u].get(u),v);
    if (l > -1) max = Math.max(max, l+1);
  }
  return cache[u] = max;
}
```

3. Create a new graph where each node is a pair $(i, t)$ where $i$ is a vertex number of the original graph and $t$ is the $w_2$-cost of arriving at that node. Then run the above longest path algorithm. Runtime is $O(TE)$.

## Eulerian Graphs

A graph is Eulerian if there is an Euler circuit: a walk that uses each edge once and ends where it starts. The test whether an undirected graph is Eulerian is easy: a graph is Eulerian iff it is connected, and every vertex has even degree.

*Proof.* To see this, first notice that an Euler circuit must leave every node the same number of times it returns to it. Hence Eulerian implies even degree. For the reverse direction,

consider the maximum length walk $w$ that doesn't reuse an edge. Call the $v_s$ and $v_e$ the starting and ending vertices of $w$. As this walk is of maximum length, every edge at $v_s$ has been used. Hence $v_e = v_s$. If this doesn't contain every edge, there must be a vertex $u$ on the circuit with an unused edge to a vertex $x$ by connectedness. Then assume our maximum length walk started and ended at $u$, and append $x$ to create a longer walk, a contradiction. □

The book has code for printing the Euler circuit of a graph (it is a bit trickier than you might expect). Effectively, you just traverse every edge and recursively print the tour on the new node, crossing out each edge you traverse (as the graph is undirected, make sure to cross out both directions of the edge).

## Euler Problems

1. Prove the number of vertices of odd degree in an undirected graph is even.
2. Prove that a connected graph with $2k$ ($k \geq 1$) odd vertices has $k$ edge-disjoint trails that cover all edges (a trail is a walk that doesn't repeat edges).
3. ($\star\star$) Consider a rectangle $R$ in the plane with corners $(0, 0)$ and $(a, b)$ with $a, b$ positive real numbers. Assume $R$ can be tiled with rectangles (may be all distinct) such that each tile has at least 1 integer side. Then $R$ has an integer side.

## Solutions

1. The sum of the degrees is twice the number of edges.
2. Pair up the odd vertices and connect them with $k$ edges. The resulting graph is Eulerian. Remove the $k$ edges to get the trails.
3. This problem has many solutions. Here are a few:
   (a) Create a graph using tile corners as vertices. Denote each tile as an $H$-tile or a $V$-tile if its horizontal or vertical edge is an integer (if both, just choose one label). For each $H$-tile, draw an edge along its horizontal sides, and do the same for the vertical edges of the $V$-tiles. Each vertex is either touching 2 or 4 rectangles, unless it is an outer corner touching only one. The resulting graph has all even degree, except the outer corners, and hence there is a path from one outer corner to another.
   (b) The integral
   $$\int_c^d \int_e^f \sin(2\pi x) \sin(2\pi y)\, dy\, dx$$
   is zero iff $c - d$ or $e - f$ is an integer. This will be true for all the tiles, and hence true for the large rectangle.

## Trees

Trees are undirected acyclic connected graphs. Firstly, many of the problems we have solved thus far are just faster when run on trees (explored in exercises). A particularly cool application of things we have learned is as follows. Consider a rooted tree $T$. The least common ancestor of two nodes $v, u$ is the deepest (w.r.t. the root) ancestor of both nodes. We want to do some precomputation that will allow us to compute, for every pair of nodes in the tree, the least common ancestor in $O(1)$ time. The idea is as follows. We will build a list of the nodes in the graph using a depth first traversal. A node is added to the list when it is first visited, and after visiting each of its children. For each node we store its first occurrence in the list, and for each list entry we store its depth in the tree. LCA is then just a min range query on the static list of depths between two nodes. This is solved in $O(1)$ time using the sparse table data structure we studied.

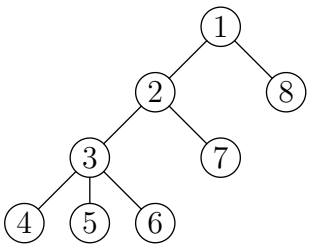
## Exercises

1. For the given tree below, and the DFS order written on the nodes (which for simplicity is the same as the vertex numbers), do the following:

   (a) Write the list of data structures stored by each node in the LCA algorithm above.

   (b) Using the lists, answer the following LCA queries:

      i. LCA$(4, 6)$,
      ii. LCA$(4, 2)$,
      iii. LCA$(3, 8)$.




2. Give an algorithm for single source shortest path on a weighted tree.

3. Give an algorithm for all pairs shortest path on a tree in $O(V^2)$.

4. ($\star$) Show that after $O(V \lg V)$ precomputation time, you can get the length of the shortest path between any pair of vertices on a tree in $O(1)$ time.

5. ($\star\star$) Suppose you are given a set $P$ of LCA queries you want to perform on a tree $T$. Give an algorithm that answers these queries in time $O(P + \alpha(V)V)$ where $\alpha$ is the inverse Ackermann's function.

## Solutions

1. (a)

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| depth | 0 | 1 | 2 | 3 | 2 | 3 | 2 | 3 | 2 | 1 | 2 | 1 | 0 | 1 | 0 |
| visit | 1 | 2 | 3 | 4 | 3 | 5 | 3 | 6 | 3 | 2 | 7 | 2 | 1 | 8 | 1 |
| first | X | 0 | 1 | 2 | 3 | 5 | 7 | 10 | 13 | X | X | X | X | X | X |

   (b) i. $\text{LCA}(4, 6) = 3$,
   ii. $\text{LCA}(4, 2) = 2$,
   iii. $\text{LCA}(3, 8) = 1$.

2. Breadth first search. Runs $O(V)$.

3. Run SSSP on each vertex.

4. Compute the sparse LCA data structure described above, and store the depth of each node. The distance from $a$ to $b$ is $\text{dist}(a, \text{LCA}(a, b)) + \text{dist}(\text{LCA}(a, b), b)$. Note the distance from a node to an ancestor is the difference in depth.

5. The algorithm is as follows:

```
//P stores for each vertex v, the list of vertices we want to
   compute the LCA for
//The ancestor array is used to associate LCAs with the sets
   of uf
void LCA(ArrayList<Integer>[] adj, UnionFind uf, int v, int[]
   ancestor,
         int[] state, ArrayList<Integer>[] P)
{
  if (state[v] != INIT) return;
  state[v] = PROCESSING;
  for (int i = 0; i < adj[v].size(); ++i)
  {
    int w = adj[v].get(i);
    LCA(adj,uf,w,state,P);
    uf.union(v,w);
    ancestor[uf.find(v)] = v;
  }
  state[v] = FINISHED;
  for (int i = 0; i < P[v].size(); ++i)
  {
    int w = P[v].get(i);
    if (state[w] != FINISHED) continue;
    System.out.printf("LCA of %d and %d is
       %d\n",v,w,ancestor[uf.find(w)]);
```

```
    }
}
```

The idea is as follows: Firstly, note that the LCA is only established at the moment both arguments are finished. Hence we are always computing the LCA with an earlier finished node. We maintain a disjoint-set data structure that groups the finished nodes by the LCA they will have with future finishing nodes. When a child of a node $n$ is finished, all future descendents of $n$ will have LCA $n$ with the earlier finished descendents. Hence all finished descendents have been unioned into 1 set.