Shortest Paths

COMP4128 Programming Challenges

School of Computer Science and Engineering
UNSW Australia

Table of Contents

- Given a weighted directed graph G with two specific vertices s and t, we want to find the shortest path that goes between s and t on the path.
- Generally, algorithms which solve the shortest path problem also solve the single source shortest path problem, which computes shortest paths from a single source vertex to every other vertex.
- You can represent all the shortest paths from the same source as a tree.

- It's very important to distinguish between directed graphs and graphs with negative weight edges! Why?
- Imagine a graph with a cycle whose total weight is negative.
- If the graph is acyclic, negative weight edges generally don't cause problems, but care should be taken regardless.
- Also, the properties of shortest paths are very different from the properties of minimum spanning trees (although the problems are similar and have similar solutions).

- Most single source shortest paths algorithms, like minimum spanning tree algorithms, rely on the basic idea of keeping around tentative shortest paths (that may be incorrect) and the relaxation of edges.
- Relax(u, v): if the currently found shortest path from our source s to a vertex v could be improved by using the edge (u, v), update it.

- If we keep track for each v of its most recently relaxed incoming edge, we can find the actual path from the source to v. How?
- For each v, we know the vertex we would've come from if we followed the shortest path from the source.
- We can work backwards from v to the source to find the shortest path *from* the source *to* v.

- If we keep relaxing our edges until we can't anymore, then we will have a shortest path.
- How do we choose which edges to relax?

Dijkstra's Algorithm

- Visit each vertex u in turn, starting from the source s. Whenever we visit the vertex u, we relax all of the edges coming out of u.
- How do we decide the order in which to visit each vertex?
- We can do something similar to Prim's algorithm and breadth-first search.
- The next vertex we process is always the unprocessed vertex with the smallest distance from the source.
- This ensures that we only need to process each vertex once: by the time we process a vertex, we have definitely found the shortest path to it.

- To decide which vertex we want to visit next, we can either just loop over all of them, or use a priority queue keyed on each vertex's current shortest known distance from the source.
- Since we know that we have a complete shortest path to every vertex by the time we visit it in Dijkstra's algorithm, we know we only visit every vertex once.
- **Complexity** Dijkstra's Algorithm is $O(E \log V)$ using a binary heap as a priority queue, or $O(V^2)$ with a loop.

- The above only holds for graphs without negative edges!
- With negative edges, we may need to visit each vertex more than once, and it turns out this makes the runtime exponential in the worst case (and it's even worse with negative cycles).

- **Implementation**

```
#include <queue>

typedef pair<int, int> edge; // (distance, vertex)
priority_queue<edge, vector<edge>, greater<edge> > pq;

// put the source s in the queue
pq.push(edge(0, s));
while (!pq.empty()) {
  // choose (d, v) so that d is minimal,
  // i.e. the closest unvisited vertex
  edge cur = pq.top();
  pq.pop();
  int v = cur.second, d = cur.first;
  if (seen[v]) continue;

  dist[v] = d;
  seen[v] = true;

  // relax all edges from v
  for (int i = 0; i < edges[v].size(); i++) {
    edge next = edges[v][i];
    int u = next.second, weight = next.first;
    if (!seen[u]) pq.push(edge(d + weight), u);
  }
}
```

Bellman-Ford Algorithm

- How do we handle negative edges in a more efficient way?
- How do we handle negative cycles?
- Bellman-Ford involves trying to relax every edge of the graph (a *global relaxation*) $|V| - 1$ times and update our tentative shortest paths each time.
- Because every shortest path has at most $|V| - 1$ edges, if after all of these global relaxations, relaxations can still be made, then there exists a negative cycle.

Bellman-Ford Algorithm

- The time complexity of Bellman-Ford is $O(VE)$.
- However, if we have some way of knowing that the last global relaxation did not affect the tentative shortest path to some vertex v, we know that we don't need to consider edges coming out of v in our next global relaxation.
- This heuristic doesn't change the overall time complexity of the algorithm, but makes it run very fast in practice on random graphs.

- **Implementation**

```
struct edge {
  int u, v;
  int w;
  edge(int _u, int _v, int _w) : u(_u), v(_v), w(_w) {}
};

vector<int> dist(n);
vector<edge> edges;

// global relaxation
bool relax() {
  bool relaxed = false;
  for (auto e = edges.begin(); e != edges.end(); ++e) {
    if (dist[e->v] > dist[e->u] + e->w) relaxed = true;
    dist[e->v] = min(dist[e->v], dist[e->u] + e->w);
  }
  return relaxed;
}

vector<int> check_neg_cycle() {
  fill(dist.begin(), dist.end(), INF);
  for (int i = 0; i < n - 1; i++) edges.push_back(edge(n - 1, i, 0));
  dist[n - 1] = 0;
  // |V|-1 global relaxations
  for (int i = 0; i < n - 1; i++) relax();
  // to be continued
```

- **Implementation (continued)**

```
  // if edges can be relaxed further, there is a negative cycle
  vector<int> res;
  for (auto e = edges.begin(); e != edges.end(); ++e) {
    if (dist[e->v] > dist[e->u] + e->w) res.push_back(e->v);
  }
  // start a BFS from every vertex in a negative cycle
  queue<int> q;
  vector<bool> seen(n);
  for (auto it = res.begin(); it != res.end(); ++it) {
    q.push(*it);
    seen[*it] = true;
  }
  // every vertex reachable from a negative cycle is affected too
  vector<int> real_res;
  while(!q.empty()) {
    int u = q.front();
    real_res.push_back(u);
    for (auto e = edges.begin(); e != edges.end(); ++e) {
      if (e->u == u && !seen[e->v]) {
        seen[e->v] = true;
        q.push(e->v);
      }
    }
    q.pop();
  }
  return real_res;
}
```

Table of Contents

All Pairs Shortest Paths

- The all pairs shortest path problem involves finding the shortest path between every pair of vertices in the graph.
- Surprisingly, this can be found in $O(V^3)$ time and $O(V^2)$ memory.

- Let $f(u, v, i)$ be the length of the shortest path between u and v using only the first i vertices (i.e. the vertices with the i smallest labels) as intermediate vertices.
- Then $f(u, u, 0) = 0$ for all vertices u, and $f(u, v, 0) = w_e$ if there is an edge e from u to v, and infinity otherwise.
- Then, we build up our solution, one new vertex at a time.
- Say we have already calculated $f(u, v, i-1)$ for all pairs u, v and some i. Then

$$f(u, v, i) = \min(f(u, v, i-1), f(u, i, i-1) + f(i, v, i-1)).$$

- The solution we already had, $f(u, v, i-1)$, definitely doesn't use i as an intermediate vertex.
- If i is the only new intermediate vertex we can use, the only new path that could be better is the shortest path $u \to i$ concatenated with the shortest path $i \to v$.

Floyd-Warshall Algorithm

- $f(u, v, i) = \min(f(u, v, i-1), f(u, i, i-1) + f(i, v, i-1))$
- Thus, $f(u, v, n-1)$ will give the length of the shortest path from u to v.
- Noting that to calculate the table for the next i, we only need the previous table, we see that we can simply overwrite the previous table at each iteration, so we only need $O(V^2)$ space.
- But what if $f(u, i, i-1)$ or $f(i, v, i-1)$ is overwritten in the table before we get to use it?
- Allowing the use of i as an intermediate vertex on a path to or from i is not going to improve the path, unless we have a negative weight cycle (Floyd-Warshall does not work on graphs that have negative weight edges)

- **Implementation**

```
// the distance between everything is infinity
for (int u = 0; u < n; ++u) for (int v = 0; v < n; ++v) {
  dist[u][v] = 2e9;
}

// update the distances for every directed edge
for (/* each edge u -> v with weight w */) dist[u][v] = w;

// every vertex can reach itself
for (int u = 0; u < n; ++u) dist[u][u] = 0;

for (int i = 0; i < n; i++) {
  for (int u = 0; u < n; u++) {
    for (int v = 0; v < n; v++) {
      dist[u][v] = min(dist[u][v], dist[u][i] + dist[i][v]);
    }
  }
}
```

- How can we find the actual shortest path?
- As well as keeping track of a `dist` table, any time the improved path (via i) is used, note that the next thing on the path from u to v is going to be the next thing on the path from u to i, which we should already know because we were keeping track of it!
- When initialising the table with the edges in the graph, don't forget to set v as next on the path from u to v for each edge $u \to v$.
- Implementing this functionality is left as an exercise.

Table of Contents

- Although some graph interpretations are obvious (e.g. cities and highways), it's often the case that the graph you must run your algorithm on is non-obvious.
- Often this doesn't admit a clean implementation using something like an explicit adjacency list.
- In many cases like this, it may be a better idea to compute the adjacencies on the fly.

- **Problem Statement** You have found a strange device that has a red button, a blue button, and a display showing a single integer, initially n. Pressing the red button multiplies the number by two; pressing the blue button subtracts one from the number. If the number stops being positive, the device breaks. You want the display to show the number m. What is the minimum number of button presses to make this happen?
- **Input** Two space-separated integers n and m ($1 \leq n, m \leq 10^8$).
- **Output** A single number, the smallest number of button presses required to get from n to m.

Example Problem: Two Buttons

- In this example, the graph is fairly obvious: we have an edge for each number we can reach from a given number.
- The graph is unweighted, so we just need to do a simple BFS to find the answer.
- However, there are so many positive integers! It'll take too long to build a graph over all of the possible numbers we could visit.
- If you think about what the best solution usually looks like, you can guess that the answer is at most $O(\log |n - m|)$ in all cases.
- If we avoid constructing a graph upfront, and just push vertices into the queue as we go (since it's easy to see what the outgoing edges from each vertex are), we can stop as soon as we reach m from n, and we will have the correct solution without needing to construct or even look at the entire graph.

- **Implementation**

```
#include <iostream>
#include <queue>
#include <algorithm>

using namespace std;

int n, m, v[200000001];
queue<int> q;

int main () {
  cin >> n >> m;
  fill(v, v + 200000001, 1e9);
  q.push(n);
  v[n] = 0;
  while (q.size()) {
    int i = q.front(); q.pop();
    if (i > 0 && v[i] + 1 < v[i-1]) {
      v[i-1] = v[i] + 1;
      q.push(i - 1);
    }
    if (i <= 100000000 && v[i] + 1 < v[i*2]) {
      v[i*2] = v[i] + 1;
      q.push(i * 2);
    }
  }
  cout << v[m];
}
```

- **Problem Statement** You are a rock climber trying to climb a wall. On this wall, there are N rock climbing holds for you to use. Whenever you are on the wall, you must be holding on to exactly three holds, each of which can be at most D distance from the other two. To move on the wall, you can only disengage from one of the holds and move it to another hold that is within D distance of the two holds that you are still holding onto. You can move from hold to hold at a rate of 1m/s. How can you reach the highest hold in the shortest amount of time, starting from some position that includes the bottom hold?
- **Input** A set of up to N ($1 \leq N \leq 50$) points on a 2D plane, and some integer D ($1 \leq D \leq 1000$). Each point's coordinates will have absolute value less than 1,000,000.
- **Output** A single number, the least amount of time needed to move from the bottom to the top.

Example Problem: Rock Climbing

- If there was no restriction that required you to always be using three holds, then this would just be a standard shortest path problem that is solvable using Dijkstra's algorithm.
- We would just need to take the points as the vertices and the distance between points as the edge weights.

- However, we need to account for the fact that we must be using three holds clustered together at any time.
- But there is a natural interpretation of the hold restriction in terms of a graph: when we move from some position that uses holds $\{a, b, c\}$ to some position where we use holds $\{a, b, d\}$, we can say that we are moving from some vertex labelled $\{a, b, c\}$ to some vertex labelled $\{a, b, d\}$.
- It can be determined whether or not such a move is allowed, i.e. if there is an edge between these vertices, in constant time.

- Now, we have a graph where we have $O(N^3)$ vertices and $O(N^4)$ edges.
- Running our shortest path algorithm on this graph directly will give us the answer we want, by definition.
- So we can solve this problem in $O(E \log V) = O(N^4 \log N^3) = O(N^4 \log N)$ time.

- **Implementation**

```
struct state {
  int pid[3];
  int dist;
};

bool operator< (const state &a, const state &b) {
  return a.dist > b.dist;
}

priority_queue<state> pq;
pq.push(begin);
bool running = true;
while (!pq.empty() && running) {
  state cur = pq.top();
  pq.pop();
  // check if done
  for (int j = 0; j < 3; j++) {
    if (cur.pid[j] == n) {
      running = false;
      break;
    }
    // to be continued
```

- **Implementation (continued)**

```
    // try disengaging our jth hold
    for (int j = 0; j < 3; j++) {
      // and moving to hold number i
      for (int i = 1; i <= n; i++) {
        // can't reuse existing holds
        if (i == cur.pid[0] || i == cur.pid[1] || i == cur.pid[2])
            continue;

        state tmp = cur;
        tmp.dist += dist(cur.pid[j], i);
        tmp.pid[j] = i;
        sort(tmp.pid, tmp.pid + 3);

        // try to move if valid
        if (valid(tmp) &&
            (!seen[tmp.pid[0]][tmp.pid[1]][tmp.pid[2]] ||
             seen[tmp.pid[0]][tmp.pid[1]][tmp.pid[2]] > tmp.dist)) {
          pq.push(tmp);
          seen[tmp.pid[0]][tmp.pid[1]][tmp.pid[2]] = tmp.dist;
        }
      }
    }
  }
}
```

Example Problem: Escape From Enemy Territory33

- **Problem Statement** You are at some position on a grid and wish to reach your safe house at some other location on the grid. However, also on certain cells on the grid are enemy safe houses, which you do not want to go near. What is the maximum possible distance you can stay away from every enemy safe house, and still be able to reach your own safe house? When there are multiple paths that keep the same distance from the enemy safe houses, print the shortest one. Distance in this problem is measured by Manhattan distance.

Example Problem: Escape From Enemy Territory34

- **Input** An $N \times M$ grid ($1 \leq N, M, \leq 1000$), and the location of your starting point, your safe house, and all the enemy safe houses. There are up to 10,000 enemy safe houses.
- **Output** Two integers, the maximum distance that you can stay away from every enemy safe house and still be able to reach your safe house, and the shortest length of such a path.

Example Problem: Escape From Enemy Territory35

- If there was no restriction stating that you must stay as far away from the enemy safe houses as possible, this would be a simple shortest path problem on a grid.
- What if we already knew how far we need to stay away from each enemy safe house?

Example Problem: Escape From Enemy Territory36

- Call the distance that we know we need to stay away from the enemy safe houses X.
- We just need to BFS out from every enemy safe house to a distance of X squares, marking all of those squares as unusable. Just marking them as seen will suffice.
- Then we can find the answer with a simple BFS from the starting point. It will ignore the squares that are too close to enemy safe houses because we've marked them as seen.

Example Problem: Escape From Enemy Territory37

- How do we view our original optimisation problem in terms of this decision problem?
  - Our simpler problem is a decision problem because we answer whether or not it's possible to get from the starting point to the safe house with distance X.
  - The original problem is an optimisation problem because it requires a 'best' answer.
- Observe that if we can stay X distance away from the enemy safe houses, then any smaller distance is also feasible, and if we cannot stay X distance away, then any larger distance is also infeasible.

Example Problem: Escape From Enemy Territory38

- This monotonicity allows us to binary search for the largest X such that we can still reach our safe house from our starting point, which we check using the BFS procedure outlined earlier.
- **Complexity** Each check takes $O(NM)$ time, and we need to perform $\log X_{MAX} = \log(N + M)$ of these checks in our binary search, so this algorithm takes $O(NM \log(N + M))$ total.

Example Problem: Escape From Enemy Territory39

- **Implementation**

```
const int dx[4] = { -1, 1, 0, 0 };
const int dy[4] = { 0, 0, -1, 1 };

vector<pair<int,int> > enemy;

// search from all enemy safe houses to find
// each square's minimum distance to an enemy safe house
queue<pair<int, int> > q;
for (auto it = enemy.begin(); it != enemy.end(); ++it) {
  q.push(*it);
}
while (!q.empty()) {
  pair<int, int> enemy = q.front(); q.pop();
  int x = enemy.first, y = enemy.second;
  // try all neighbours
  for (int i = 0; i < 4; ++i) {
    int nx = x + dx[i], ny = y + dy[i];
    // if off board, ignore
    if (nx < 0 || nx >= X || ny < 0 || ny >= Y) continue;
    if (d[nx][ny] != -1) continue;
    d[nx][ny] = d[x][y] + 1;
    q.push(make_pair(nx, ny));
  }
}
```

Example Problem: Escape From Enemy Territory40

- **Implementation (continued)**

```
// binary search
int lo = -1, hi = min(d[x1][y1], d[x2][y2]), sol = -1;
while (lo != hi) {
  int mid = (lo + hi + 1) / 2;
  // BFS, since the edges are unit weight
  vector<vector<int> > d2(X, vector<int>(Y, -1));
  d2[x1][y1] = 0;
  q.push(make_pair(x1, y1));
  while (!q.empty()) {
    int x = q.front().first, y = q.front().second; q.pop();
    for (int i = 0; i < 4; ++i) {
      int nx = x + dx[i], ny = y + dy[i];
      if (nx < 0 || nx >= X || ny < 0 || ny >= Y) continue;
      if (d[nx][ny] < mid) continue;
      if (d2[nx][ny] != -1) continue;
      d2[nx][ny] = d2[x][y] + 1;
      q.push(make_pair(nx, ny));
    }
  }

  if (d2[x2][y2] == -1) hi = mid - 1;
  else lo = mid, sol = d2[x2][y2];
}
```