Shortest Paths

COMP4128 Programming Challenges

School of Computer Science and Engineering
UNSW Australia

- 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 graph.
- Note that the unweighted case is solved by BFS.
- 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 graphs where all edges are positive and graphs with negative weight edges! Why?
- Imagine a graph with a cycle whose total weight is negative.
- Even if there are no negative cycles, this may still cause problems depending on your algorithm choice!
- If the graph is acyclic, negative weight edges generally don't cause problems, but care should be taken regardless.

- Most single source shortest paths algorithms rely on the basic idea of building shortest paths iteratively. At any point, we keep what we think is the shortest path to each vertex and we update this by "relaxing" 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.
- For graphs with non-negative weights, we can get away with only relaxing vertices for which we know the optimal distance. But with negative weights, this becomes trickier.

- 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?

- For now, suppose there are no negative edges.
- 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 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. Prove this inductively!

- 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).
- In short: don't use Dijkstra's if there's any negative edges! (But most graphs you see won't have them).

- **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));
  }
}
```

- How do we handle negative edges in a more efficient way?
- How do we handle negative cycles?
- **Key Observation:** If a graph has no negative cycle then all shortest paths from u have length $\leq |V| - 1$. Conversely, a negative cycle implies there is a shortest path of length $|V|$ better than any path of length $|V| - 1$.
- So we should instead build up shortest paths by number of edges, not just from u outwards.
- 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.

- 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.
- Sometimes called Shortest Path Faster Algorithm (SPFA) for some reason...

- **Implementation**

```
struct edge {
  int u, v, w;
  edge(int _u, int _v, int _w) : u(_u), v(_v), w(_w) {}
};
vector<int> dist(n);
vector<edge> edges;
// global relaxation: try to relax every edge in the graph
// Returns if any distance was updated.
bool relax() {
  bool relaxed = false;
  for (auto e = edges.begin(); e != edges.end(); ++e) {
    // we don't want to relax an edge from an unreachable vertex
    if (dist[e->u] != INF && dist[e->v] > dist[e->u] + e->w) {
      relaxed = true;
      dist[e->v] = dist[e->u] + e->w;
    }
  }
  return relaxed;
}
```

- **Implementation (continued)**

```
// Puts distances from source (n-1) in dist
// Returns true if there is a negative cycle, false otherwise.
// NOTE: You can't trust the dist array if this function returns True.
vector<int> find_dists_and_check_neg_cycle() {
  fill(dist.begin(), dist.end(), INF);
  dist[n-1] = 0;
  // |V|-1 global relaxations
  for (int i = 0; i < n - 1; i++) relax();

  // If any edge can be relaxed further, there is a negative cycle
  for (auto e = edges.begin(); e != edges.end(); ++e) {
    if (dist[e->u] != INF &&
        dist[e->v] > dist[e->u] + e->w) {
      return true;
    }
  }
  // Otherwise, no negative cycle, that condition is actually a iff
  return false;
}
```

- Slight technical note: Due to how we coded `relax`, after $n-1$ iterations, `dist[i]` may be the distance of a path of length $> n-1$. But this does not matter, everything still holds.
- If there is a negative cycle, you can't trust the distances computed. Call a vertex v ruined if its shortest distance from u is actually $-\infty$.
- For every negative cycle, in every relaxation round at least one of its vertices will be updated.
- Hence, to find all ruined vertices, DFS out of all vertices who were relaxed in the V-th round.
- To find a specific negative cycle, backtrack from any 'ruined' vertex.

- 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, by dynamic programming.

- 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.
- The key is to build this up for increasing values of i.
- **Base Case:** 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.

- 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 $i \to v$.

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

- **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] is the length of the shortest path from u to v
           using only 0 to i-1 as intermediate vertices
      // now that we're allowed to also use i, the only new path that
           could be shorter is u -> i -> v
      dist[u][v] = min(dist[u][v], dist[u][i] + dist[i][v]);
    }
  }
}
```

What if there is a negative cycle?

- If there is a negative-weight cycle, our invariant is instead $f(u, v, i) \leq$ shortest simple path from $u \to v$ only using the first i vertices as intermediaries.
- Hence $f(u, u, n)$ will be negative for vertices in negative cycles. Also you can't trust the calculated distances, same as Bellman-Ford.
- Note that if there *are* negative-weight edges, but *no negative cycles*, Floyd-Warshall will correctly find all distances.
  - Every *undirected* graph with a negative-weight edge contains a negative cycle.

- 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.

- 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.
- Also, sometimes modifying the graph you consider is helpful!

- **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 ($1 \leq n, m \leq 10^7$).
- **Output** A single number, the smallest number of button presses required to get from n to m.

- In this example, we should think of our button presses as transitions.
- Hence our graph has numbers as its vertices and edges representing which numbers can reach each other through button presses.
- The graph is unweighted, so we just need to do a simple BFS to find the answer.

- **Implementation**

```
#include <bits/stdc++.h>
using namespace std;

const int MAXVAL=20*1000*1000+5;
const int INF = 1000*1000*1000+7;
int n, m, v[MAXVAL];
queue<int> q;

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

- **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.

- 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;
        }
      }
    }
  }
}
```

- **Problem Statement** There are N cities, each in one of C countries. There are two modes of travel.
  - There are A unidirectional direct flight routes connecting two cities, u_i, v_i with weight w_i.
  - There are B unidirectional inter-country flights, connecting two countries, a_i, b_i with a weight w_i. These flights can be boarded from any city in the source country and disembarked from in any city in the destination country.

  Find the shortest distance from city 1 to city N.
- **Input** First line, 4 integers N, C, A, B. $1 \leq N, C, A, B \leq 100,000$. Next line, N integers, c_i, denoting the country the i-th city is in. Next A lines each with 3 integers $u_i, v_i, w_i, 1 \leq u_i, v_i \leq N, 1 \leq w_i \leq 10^9$. Next B lines each with 3 integers $a_i, b_i, w_i, 1 \leq a_i, b_i \leq C, 1 \leq w_i \leq 10^9$.
- **Output** A single integer, shortest distance from city $1 \to N$. -1 if impossible.

- **Sample Input:**

```
4 3 2 2
1 2 2 3
1 4 100
2 4 20
1 2 50
1 3 80
```

- **Sample Output**: 70
- **Explanation**: Fly with an intercountry flight of cost 50 to city 2. Then take a direct flight with cost 20 to city 4.

- How to view this as a graph? Without inter-country flights, this is routine.
- With inter-country flights, we could just generate all edges between cities in country a_i and b_i.
- But this is too many edges.

- **Observation:** We should only consider inter-country flights originating from A the first time we reach a city in A. Similarly, we should only consider inter-country flights to B once.
- This sounds just like how we treat cities in Dijkstra's.
- To encode this, we should treat countries just like cities in our graph, they should have nodes. Be careful, we need 2 nodes per country, one to encode outgoing flights and one to encode incoming flights. What are the edges though?
- The natural ones.
  - Cities go to the "outgoing" node for their country.
  - The "incoming" node for a country goes to all cities in that country.
  - "Outgoing" country nodes connect by inter-country flights to "incoming" country nodes.
- $O(N + C)$ nodes, $O(N + A + B)$ edges. Okay!

Example Problem: Intercountry


```cpp
#include <bits/stdc++.h>
using namespace std;

const int MAXN = 100005;
const int MAXC = 100005;
int N, C, A, B;
// (dest, dist)
// "outgoing" country nodes are at MAXN + (country id)
// "incoming" country nodes are at MAXN + MAXC + (country id)
vector<pair<int, long long>> allE[MAXN+2*MAXC];

int main() {
    cin >> N >> C >> A >> B;
    for (int i = 1; i <= N; i++) {
        int cC; cin >> cC;
        allE[i].emplace_back(MAXN + cC, 0);
        allE[MAXN+MAXC+cC].emplace_back(i, 0);
    }
    for (int i = 0; i < A; i++) {
        int a, b; long long w;
        cin >> a >> b >> w;
        allE[a].emplace_back(b, w);
    }
    for (int i = 0; i < B; i++) {
        int a, b; long long w;
        cin >> a >> b >> w;
        allE[MAXN+a].emplace_back(MAXN+MAXC+b);
    }
    // Run your favorite shortest dist algo!
}
```

Example Problem: Escape From Enemy Territory41

- **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 Territory42

- **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 Territory43

- 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 Territory44

- 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 Territory45

- 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.

- 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 Territory47

- **Implementation**

```
const int di[4] = { -1, 1, 0, 0 };
const int dj[4] = { 0, 0, -1, 1 };

vector<pair<int,int>> enemies;

// 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 = enemies.begin(); it != enemies.end(); ++it) {
  q.push(*it);
}
while (!q.empty()) {
  pair<int, int> enemy = q.front(); q.pop();
  int i = enemy.first, j = enemy.second;
  // try all neighbours
  for (int d = 0; d < 4; ++d) {
    int ni = i + di[d], nj = j + dj[d];
    // if off board, ignore
    if (ni < 0 || ni >= N || nj < 0 || nj >= M) continue;
    if (dist_to_enemy[ni][nj] != -1) continue;
    dist_to_enemy[ni][nj] = dist_to_enemy[i][j] + 1;
    q.push(make_pair(ni, nj));
  }
}
```

Example Problem: Escape From Enemy Territory48

- **Implementation (continued)**

```
// binary search
int lo = -1, hi = min(dist_to_enemy[i1][j1], dist_to_enemy[i2][j2]),
      sol = -1;
while (lo != hi) {
  int X = (lo + hi + 1) / 2;
  // BFS, since the edges are unit weight
  vector<vector<int> > d2(N, vector<int>(M, -1));
  d2[i1][j1] = 0;
  q.push(make_pair(i1, j1));
  while (!q.empty()) {
    int i = q.front().first, j = q.front().second; q.pop();
    for (int d = 0; d < 4; ++d) {
      int ni = i + di[d], nj = j + dj[d];
      if (ni < 0 || ni >= N || nj < 0 || nj >= M) continue;
      if (dist_to_enemy[ni][nj] < X) continue;
      if (dist[ni][nj] != -1) continue;
      dist[ni][nj] = dist[i][j] + 1;
      q.push(make_pair(ni, nj));
    }
  }

  if (dist[i2][j2] == -1) hi = X - 1;
  else lo = X, sol = dist[i2][j2];
}
```

- Given M inequalities of the form $x_i - x_j \leq c_k$, are there real numbers $x_1, x_2, \ldots, x_N$ that satisfy those inequalities?
- What does this have to do with shortest paths??
- This can actually be solved in $O(MN)$ using Bellman-Ford!

- We need to transform the problem into a graph problem, by creating a graph from the difference constraints.
- Create a vertex for every variable, and for every constraint $x_i - x_j \leq c_k$, create an edge from vertex j to vertex i with weight c_k.
- Then, create a source s that has a zero weight edge to every other vertex.

- If we run Bellman-Ford from this source vertex and examine the shortest paths from the source, we obtain a solution.
- For every edge from j to i, from its definition we know that the length of the shortest path to i, say d_i, and the length of the shortest path to j, d_j, satisfy the inequality $d_i \leq d_j + c_k$.
- We can rewrite $d_i \leq d_j + c_k$ as $d_i - d_j \leq c_k$, which means that the distances from the source satisfy the difference constraints.

- What happens when there is a negative weight cycle?
- What happens when there is no solution to our system of difference constraints?

- When we have a negative weight cycle, we have a set of edge inequalities with the variables in that cycle:

$$x_1 - x_2 \leq c_1$$
$$x_2 - x_3 \leq c_2$$
$$x_3 - x_1 \leq c_3$$

- If we add all these inequalities together, because they form a cycle, all the variables cancel out, giving

$$0 \leq c_1 + c_2 + \ldots + c_k,$$

  where $c_1, c_2, \ldots, c_k$ are the edge weights on our cycle.
- Since the cycle is negative, the RHS adds up to a negative number. Thus we have a contradiction, so there is no solution.

- **Problem Statement** Given a weighted directed graph with non-negative edge weights, what is the length of the second shortest path on the graph from s to t? The second shortest path is the walk with the shortest length that is distinct from the shortest path; in particular, it is the same length as the shortest path if and only if there are multiple shortest paths.
- **Input** A weighted directed graph with N vertices and M edges ($1 \leq N, M \leq 100,000$).
- **Output** A single integer, the length of the second shortest walk.

- Observe that the second shortest walk will differ from a given shortest path by at least one edge.
- Then after finding the shortest path, we can iterate over all edges on the shortest path, and see what happens if we go along a different edge instead.

- Let the shortest path from u to v have length d_{uv}. If some edge (u, v) is blocked, and we take some other edge (u, w) instead, with weight e_{uw} we get a path of length $d_{su} + e_{uw} + d_{wt}$.
- Since we're iterating over all smallest possible differences to the shortest path, one of these distances will be our answer.
- So we just try them all (i.e. all choices of (u, w)) and take the one of minimum length.
- How do we efficiently compute all the d_{wt}?

- **Implementation**

```
// dist1[v] = shortest distance from v to 1
vector<int> dist1(n+1, -1);
vector<int> prev1(n+1, -1);
{
  set<edge> s;
  vector<bool> seen(n+1);
  prev1[1] = -2;
  for (s.insert(edge(0, 1)); !s.empty(); s.erase(s.begin())) {
    edge cur = *s.begin();
    if (seen[cur.y]) continue;
    dist1[cur.y] = cur.x;
    seen[cur.y] = true;
    foreach(v, adj[cur.y]) {
      if (!seen[v->y])
        if (dist1[v->y] == -1 || dist1[v->y] > cur.x+v->x) {
          prev1[v->y] = cur.y;
          s.insert(edge(cur.x+v->x, v->y));
        }
    }
  }
}

// to be continued
```

- **Implementation (continued)**

```
// dist2[v] = shortest distance from v to n
vector<int> dist2(n+1, -1);
vector<int> prev2(n+1, -1);
{
  set<edge> s;
  s.insert(edge(0, n));
  vector<bool> seen(n+1);
  prev2[n] = -2;
  for (s.insert(edge(0, n)); !s.empty(); s.erase(s.begin())) {
    edge cur = *s.begin();
    if (seen[cur.y]) continue;
    dist2[cur.y] = cur.x;
    seen[cur.y] = true;
    foreach(v, adj[cur.y]) {
      if (!seen[v->y])
        if (dist2[v->y] == -1 || dist2[v->y] > cur.x+v->x) {
          prev2[v->y] = cur.y;
          s.insert(edge(cur.x+v->x, v->y));
        }
    }
  }
}

int res = 1<<30;
for (int v = n; v != -2; v = prev1[v])
  foreach(e, adj[v])
    res = min(res, dist1[v] + e->x + dist2[e->y]);
```