Traveling Salesman

Find a minimum-cost hamiltonian path or cycle in a complete graph.

Held-Karp algorithm

Time	$2^n n^2$
Space	$2^n \sqrt{n}$ (store $\mathrm{dp}[y][S]$ for $\lvert S\rvert = n-1$)

Let $\mathrm{dp}[y][S]$ be the minimum cost of a path from 0 to y through the intermediate vertices in S.

- $\mathrm{dp}[y][\varnothing] = W[0 \to y]$
- $\mathrm{dp}[y][S] = \min_{x \in S} \{\mathrm{dp}[x][S - \{x\}] + W[x \to y]\}$

Notes

- If $V \subseteq \{0, \ldots, 63\}$, use an integer bitset for S.
- To reconstruct the hamiltonian path or cycle, track $\mathbf{prev}[y][S]$.
- For the hamiltonian cycle, compute $\min_{x \in V} \{\mathrm{dp}[x][V - \{x\}] + W[x \to 0]\}$.

Knapsack

Given $0 \leq W$, $[1 \leq v_1, \ldots, v_N]$, and $[1 \leq w_1, \ldots, w_N]$, maximize $\sum v_i x_i$ with $\sum w_i x_i \leq W$.

0/1 Knapsack

When $0 \leq x_i \leq 1$.

Time	NW
Space	W (store $\text{dp}[n-1]$)

Let $\text{dp}[n][w]$ be the solution when $W = w$ and $N = n$.

- $\text{dp}[0][w] = 0$
- $\text{dp}[n][w] = \max\left(\text{dp}[n-1][w], \max\limits_{1 \leq i \leq N} \{\text{dp}[n-1][w - w_i] + v_i : w \geq w_i\}\right)$

Bounded Knapsack

When $0 \leq x_i \leq k$.

Time	kNW
Space	W (store $\text{dp}[n-1]$)

Solve the 0/1 knapsack problem where each v_i, w_i implicitly appears k times ($N' = kN$).

Unbounded Knapsack

When $0 \leq x_i < \infty$.

Time	NW
Space	W

Let $\text{dp}[w]$ be the solution when $W = w$.

- $\text{dp}[0] = 0$
- $\text{dp}[w] = \max\limits_{1 \leq i \leq N} \{\text{dp}[w - w_i] + v_i : w \geq w_i\}$

Notes

- Divide $w_1, \ldots, w_n, W$ by their GCD to improve complexity in some cases.

Polygon

```
// TODO
```

Point

```
#define _sca(op) \
Point &operator op##=(double t) { x op##= t; y op##= t; return *this; } \
Point operator op(double t) { return Point(*this) op##= t; }

#define _vec(op) \
Point &operator op##=(Point p) { x op##= p.x; y op##= p.y; return *this; } \
Point operator op(Point p) { return Point(*this) op##= p; }

struct Point {
    double x, y;

    Point(double x, double y): x(x), y(y) {}

    Point operator-() { return Point(-x, -y); }

    _vec(+)
    _vec(-)
    _sca(*)
    _sca(/)
};

Point operator*(double t, Point p) { return p * t; }
string _s(Point p,...) { return _s(make_pair(p.x, p.y)); }

/* Dot product */
double dot(Point p, Point q) { return p.x * q.x + p.y * q.y; }
double norm(Point p) { return dot(p, p); }
double abs(Point p) { return sqrt(norm(p)); }
double dist(Point p, Point q) { return abs(p - q); }
Point unit(Point p) { return p / abs(p); }
bool eq(Point p, Point q) { return dist(p, q) < eps; }
double proj(Point p, Point q) { return dot(p, q) / abs(q); }
Point project(Point p, Point q) { return dot(p, q) / norm(q) * q; }
double angle(Point p, Point q) { return acos(proj(p, q) / abs(p)); }

/* Cross product */
double cross(Point p, Point q) { return p.x * q.y - p.y * q.x; }
double pgram(Point p, Point q) { return abs(cross(p, q)); }
double triangle(Point p, Point q) { return pgram(p, q) / 2; }
```

Line

```
#define _orient(u, v) \
double dist(u x, v y) { return abs(sdist(x, y)); } \
int side(u x, v y) { double s = sdist(x, y); return (s > eps) - (s < -eps); } \
bool incident(u x, v y) { return side(x, y) == 0; }

struct Line {
    double a, b, c;

    Line(double a, double b, double c): a(a), b(b), c(c) { normalize(); }
    Line(Point p, Point q): a(p.y - q.y), b(q.x - p.x), c(-a * p.x - b * p.y) { normalize(); }

    void normalize() {
        double z = sqrt(a * a + b * b);
        if (a < 0 && b < 0) z = -z;
        a /= z; b /= z; c /= z;
    }
};

/* Line */
string _s(Line m,...) { return '(' + _s(m.a) + "x + " + _s(m.b) + "y + " + _s(m.c) + " = 0)"; }
Point normal(Line m) { return Point(m.a, m.b); }
Point tangent(Line m) { return Point(m.b, -m.a); }

/* Point, Line */
double sdist(Point p, Line m) { return m.a * p.x + m.b * p.y + m.c; }
_orient(Point, Line)
Point project(Point p, Line m) { return p - normal(m) * sdist(p, m); }

/* Line, Line */
bool eq(Line m, Line n) { return eq(normal(m), normal(n)) && abs(m.c - n.c) < eps; }
double cross(Line m, Line n) { return cross(normal(m), normal(n)); }
double angle(Line m, Line n) { double a = angle(normal(m), normal(n)); return min(a, pi - a); }
bool parallel(Line m, Line n) { return abs(cross(m, n)) < eps; }
Point intersect(Line m, Line n) {
    return Point(m.b * n.c - m.c * n.b, m.c * n.a - m.a * n.c) / cross(m, n);
}
```

Convex Hull

TODO

Circle

```
struct Circle {
    Point o; double r;

    Circle(Point o, double r): o(o), r(r) {}
}

/* Circle */
string _s(Circle c) { return _s(make_pair(c.o, c.r)); }
double area(Circle c) { return pi * c.r * c.r; }
double circum(Circle c) { return 2 * pi * c.r; }

/* Point, Circle */
double sdist(Point p, Circle c) { return dist(c.o, p) - r; }
_orient(Point, Circle)
Point project(Point p, Circle c) { return c.o + unit(p - c.o) * r; }

/* Line, Circle */
double sdist(Line l, Circle c) { return dist(c.o, l) - r; }
_orient(Line, Circle)
void intersect(Line l, Circle c, Point &p, Point &q) {
    Point m = project(c.o, l);
    Point v = tangent(l) * sqrt(c.r * c.r - norm(m - c.o));
    p = m + v;
    q = m - v;
}
```

Notes

- A point is inside a circle when `side(p, c) == -1`.
- A line intersects a circle when `side(l, c) <= 0` (in two points when `side(l, c) == -1`).
- A line is tangent to a circle when `incident(l, c)` (at `project(c.o, l)`).

BFS

Time	$\|V\| + \|E\|$
Space	$\|V\|$

Data Structures

Name	Type	Initial Value
`front`	`Queue<Vertex>`	`[start]`
`seen`	`Set<Vertex>`	`{start}`
`prev?`	`Map<Vertex, Vertex>`	`{}`

Algorithm

```
while (!front.empty()) {
    Vertex u = front.top();
    front.pop();

    // Visit u

    for (Vertex v : E[u]) {
        if (seen.has(v)) continue;
        seen.add(v);
        prev[v] = u;

        // See u → v

        front.push(v);
    }
}
```

Results

- `seen` is the set of vertices connected to start.
- `prev[v]` is the penultimate vertex on **some** shortest path from `start` to `v` (if they are connected).

Notes

- `seen` may be redundant if `prev` is used.
- Finds shortest paths by number of edges (not weight).

Prim's Algorithm

Time	$(\lvert E\rvert + \lvert V\rvert) \log \lvert V\rvert$
Space	$\lvert V\rvert^2$

Data Structures

Name	Type	Initial Value
`front`	`PriorityQueue<(Weight, Vertex)>`	`[(0, start)]`
`visited`	`Set<Vertex>`	`{}`
`parent`	`Map<Vertex, Vertex>`	`{}`
`cost`	`Map<Vertex, Weight>`	`{}`
`tree`	`Map<Vertex, List<(Weight, Vertex)>>`	`{}`

Algorithm

```
while (!front.empty()) {
    (Weight w, Vertex u) = front.top();
    front.pop();

    if (visited.has(u)) continue;
    visited.add(u);

    // Visit u

    if (parent.has(u)) {
        tree[u].push((w, parent[u]));
        tree[parent[u]].push((w, u));

        // Connect parent[u] to u
    }

    for ((Vertex v, Weight x) : E[u]) {
        if (!cost.has(v) || cost[v] > x) {
            cost[v] = x;
            parent[v] = u;

            // Relax u → v

            front.push((x, v));
        }
    }
}
```

Results

- `tree` is **some** MST of `start`'s connected component.

Notes

- Fails on directed graphs.

DFS

Time	$|V| + |E|$
Space	$|V|$

Data Structures

Name	Type	Initial Value
`back`	`Stack<Vertex>`	`[start]`
`visited`	`Set<Vertex>`	`{}`

Algorithm

```
while (!back.empty()) {
    Vertex u = back.top();

    if (!visited.has(u)) {
        visited.add(u);

        // Start visiting u
    } else {
        // Backtrack to u
    }

    bool follow = false;
    for (Vertex v : E[u]) {
        if (visited.has(v)) continue;

        // Follow u → v

        back.push(v);
        follow = true;
        break;
    }
    if (follow) continue;

    // Finish visiting u

    back.pop();
}
```

Results

- `visited` is the set of vertices connected to `start`.

Floyd-Warshall

Time	$\lvert V \rvert^3$
Space	$\lvert V \rvert^2$

Data Structures

Name	Type	Initial Value
`next?`	`Map<(Vertex, Vertex), Vertex>`	`{E(u, v): v}`
`dist`	`Map<(Vertex, Vertex), Distance>`	`{E(u, v): w, V(v, v): 0}`

Algorithm

```
for (Vertex m : V) {
    for (Vertex u : V) {
        for (Vertex v : V) {
            if (!dist.has((u, m)) || !dist.has((m, v))) continue;

            if (dist[u, v] > dist[u, m] + dist[m, v]) {
                dist[u, v] = dist[u, m] + dist[m, v];
                next[u, v] = next[u, m];

                // Relax u → v through m
            }
        }
    }
}

for (Vertex v : V) {
    if (dist[v, v] < 0) {
        return false; // Negative cycle detected
    }
}

return true;
```

Results

- `dist[u, v]` is the distance from `u` to `v` (if they are connected).
- `next[u, v]` is the second vertex on **some** shortest path from `u` to `v` (if they are connected and distinct).

Notes

- Johnson's Algorithm is faster for sparse graphs.
- Fails on negative cycles (detected).

Bellman-Ford

Time	$\lvert V \rvert \cdot \lvert E \rvert$
Space	$\lvert V \rvert$

Data Structures

Name	Type	Initial Value
`prev?`	`Map<Vertex, Vertex>`	`{}`
`dist`	`Map<Vertex, Distance>`	`{start: 0}`

Algorithm

```
for (|V| - 1) {
    for ((Vertex u, Vertex v, Weight w) : E) {
        if (!dist.has(u)) continue;

        if (!dist.has(v) || dist[v] > dist[u] + w) {
            dist[v] = dist[u] + w;
            prev[v] = u;

            // Relax u → v
        }
    }
}

// Extra iteration
for ((Vertex u, Vertex v, Weight w) : E) {
    if (dist.has(u) && dist[v] > dist[u] + w) {
        return false; // Negative cycle detected
    }
}

return true;
```

Results

- `dist[v]` is the distance from `start` to `v` (if they are connected).
- `prev[v]` is the penultimate vertex on **some** shortest path from `start` to `v` (if they are connected).

Notes

- The extra iteration will relax some vertex iff a negative cycle is reachable from `start`.
- `|V| - 1` extra iterations will relax `v` iff there is a negative cycle between `start` and `v`.
- Fails on negative cycles (detected).

Johnson’s Algorithm

Time	$(\lvert E\rvert + \lvert V\rvert)\lvert V\rvert \log \lvert V\rvert$
Space	$\lvert V\rvert^2$

Data Structures

Name	Type	Initial Value
`adjusted`	`Map<Vertex, List<(Weight, Vertex)>>`	`G + {q: [V(0, v)]}`
`height`	`Map<Vertex, Distance>`	`{}`
`prev?`	`Map<Vertex, Map<Vertex, Vertex>>`	`{}`
`dist`	`Map<Vertex, Map<Vertex, Distance>>`	`{}`

Algorithm

```
if (!BellmanFord(adjusted, &height, q)) return false;
adjusted.remove(q);

// Reweighting
for (Vertex u : V) {
    for ((Weight w, Vertex v) : adjusted[u]) {
        w += height[u] - height[v];
    }
}

// Repeated Dijkstra
for (Vertex v : V) {
    (dist[v], prev[v]) = Dijkstra(adjusted, v);
}

return true;
```

Results

- `dist[u][v] - height[u] + height[v]` is the distance from `u` to `v` (if they are connected).
- `prev[u][v]` is the penultimate vertex on **some** shortest path from `u` to `v` (if they are connected).

Notes

- Bellman-Ford & reweighting can be skipped for graphs with non-negative edges.
- Fails on negative cycles (detected during Bellman-Ford).

Dijkstra's Algorithm

Time	$(\|E\| + \|V\|) \log \|V\|$
Space	$\|V\|^2$

Data Structures

Name	Type	Initial Value
`front`	`PriorityQueue<(Distance, Vertex)>`	`[(0, start)]`
`visited`	`Set<Vertex>`	`{}`
`prev?`	`Map<Vertex, Vertex>`	`{}`
`dist`	`Map<Vertex, Distance>`	`{start: 0}`

Algorithm

```
while (!front.empty()) {
    (Distance d, Vertex u) = front.top();
    front.pop();

    if (visited.has(u)) continue;
    visited.add(u);

    // Visit u

    for ((Vertex v, Weight w) : E[u]) {
        Distance r = d + w;
        if (!dist.has(v) || dist[v] > r) {
            dist[v] = r;
            prev[v] = u;

            // Relax u → v

            front.push((r, v));
        }
    }
}
```

Results

- `dist[v]` is the distance from `start` to `v` (if they are connected).
- `prev[v]` is the penultimate vertex on **some** shortest path from `start` to `v` (if they are connected).

Notes

- Fails on graphs with negative edges (use Bellman-Ford).

Edmonds-Karp

Time	$\lvert V\rvert \cdot \lvert E\rvert^2$
Space	$\lvert V\rvert$

Data Structures

Name	Type	Initial Value
`residual`	`Map<Vertex, Map<Vertex, Capacity>>`	`G + E{v: {u: 0}}`

Algorithm

```
Flow flow = 0;
for (;;) {
    // Find an augmenting path
    // PathBFS skips (u → v) where residual[u][v] == 0 and stops at sink
    Map<Vertex, Vertex> prev;
    if (!PathBFS(residual, source, sink, &prev) break;

    // Find the bottleneck capacity of the augmenting path
    Capacity cap = INF;
    for (Vertex v = sink, u = v; prev.has(u); v = u) {
        u = prev[u];
        cap = min(cap, residual[u][v]);
    }

    // Send flow down the augmenting path
    for (Vertex v = sink, u = v; prev.has(u); v = u) {
        u = prev[u];
        residual[u][v] -= cap;
        residual[v][u] += cap;
    }

    flow += cap;
}
```

Results

- `flow` is the maximum flow from `source` to `sink`.
- `residual[v][u]` is the flow through `u → v` in **some** maximum flow.

Notes

- Fails on graphs with self-loops, parallel edges, and bidirectional edges.
- Can be fixed for those graphs by defining `struct Edge { Vertex v, Capacity, Flow, Edge *rev }`.

Dinitz's Algorithm

Time	$\|V\|^2 \cdot \|E\|$
Space	$\|V\|$

Data Structures

Name	Type	Initial Value
`residual`	`Map<Vertex, Map<Vertex, Capacity>>`	`G + E{v: {u: 0}}`

Algorithm

```
Flow flow = 0;
for (;;) {
    // LevelBFS skips (u → v) where residual[u][v] == 0 and stops at sink
    // Returns a DAG of edges seen from each vertex traversed
    Map<Vertex, Stack<Vertex>> layered;
    if (!LevelBFS(residual, source, sink, &layered)) break;

    for (;;) {
        // PathDFS pops edges when it backtracks and stops at sink
        Map<Vertex, Vertex> prev;
        if (!PathDFS(layered, source, sink, &prev)) break;

        // Get the bottleneck capacity of the augmenting path
        Capacity cap = INF;
        for (Vertex v = sink, u = v; prev.has(u); v = u) {
            u = prev[u];
            cap = min(cap, residual[u][v]);
        }

        // Send flow down the augmenting path
        for (Vertex v = sink, u = v; prev.has(u); v = u) {
            u = prev[u];
            residual[u][v] -= cap;
            residual[v][u] += cap;

            if (residual[u][v] == 0) layered[u].pop();
        }

        flow += cap;
    }
}
```

Results

- `flow` is the maximum flow from `source` to `sink`.
- `residual[v][u]` is the flow through `u → v` in **some** maximum flow.

Notes

- Fails on graphs with self-loops, parallel edges, and bidirectional edges.
- Can be fixed for those graphs by defining `struct Edge { Vertex v, Capacity, Flow, Edge *rev }`.

Topological Sort

Time	$\lvert V\rvert + \lvert E\rvert$
Space	$\lvert V\rvert$

Data Structures

Name	Type	Initial Value
`sorted`	`Set<Vertex>`	`{}`
`topo`	`List<Vertex>`	`[]`

Algorithm

```
for (Vertex start : V) {
    if (sorted.has(start)) continue;

    Set<Vertex> visited = {};
    Stack<Vertex> backtrack = [start];

    while (!backtrack.empty()) {
        Vertex u = backtrack.top();
        visited.add(u);

        bool follow = false;
        for (Vertex v : E[u]) {
            if (sorted.has(v)) continue;
            if (visited.has(v)) return false; // Cycle detected

            backtrack.push(v);
            follow = true;
            break;
        }
        if (follow) continue;

        sorted.add(u);
        topo.push(u);
        backtrack.pop();
    }
}

return true;
```

Results

- `i < j` implies there is no path from `topo[i]` to `topo[j]` (reverse topological order).

Notes

- Impossible with cycles (detected).

C++ Tricks

- `set` is better than `priority_queue` for Prim's and Dijkstra's:

	`set<T>`	`priority_queue<T, vector<T>, greater<T>>`
Insert	`q.insert(x)`	`q.push(x)`
Top	`*q.begin()`	`q.top()`
Pop	`q.erase(q.begin())`	`q.pop()`
Delete	`q.erase(q.find(x))`	N/A

C++ Prelude

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

#define DEBUG 1
#define dbg(x) (DEBUG ? _d((#x), (x)) : (x))
#define mod(x, m) ((((x) % (m)) + (m)) % (m))
#define _f (k ? '\n' + string(f, ' ') : "")

template <class T> auto _s(T x,...) -> decltype(to_string(x)) { return to_string(x); }
string _s(char x,...) { return string("'") + x + "'"; }
string _s(string x,...) { return '"' + x + '"'; }

template <class P, class Q> string _s(pair<P, Q> x, int f=0, int k=0) {
    return _f + '(' + _s(x.first) + ", " + _s(x.second) + ')';
}

template <class T> auto _s(T x, int f=0, int k=0) -> decltype(end(x), string()) {
    string s; int i = 0; auto b = begin(x), e = end(x);
    while (b != e) s += _s(*b++, f+1, i++), s += (b == e ? "" : ", ");
    return _f + '[' + s + ']';
}

template <class T> T& _d(string s, T&& x) {
    cout << s + " = " + _s(x, s.size() + 3) + '\n'; return x;
}

typedef vector<int> vi;
typedef pair<int, int> ii;
typedef int64_t i64;
typedef uint64_t u64;

const double eps = 1e-9;
const double pi = 2 * acos(0);
const double dinf = 1 / 0.0;
const int inf = numeric_limits<int>::max() >> 2;
const long linf = numeric_limits<i64>::max() >> 2;
```

Sample Graphs

```
10
0 0 (no vertices)
9 0 (no edges)
1 1 (self-loop)
0 0 5
2 2 (parallel edges)
0 1 0
0 1 2
3 3 (cycle)
0 1 0
1 2 2
2 0 4
4 2 (disconnected)
0 1 0
2 3 2
3 3 (non-consecutive)
9 3 0
7 3 2
7 9 4
3 3 (negative edge)
0 1 -2
1 2 4
2 0 0
3 3 (negative cycle)
0 1 -4
1 2 2
2 0 0
5 10 (complete graph)
0 1
0 2
0 3
0 4
1 2
1 3
1 4
1 5
2 3
2 4
2 5
3 4
3 5
4 5
```

Other possibilities

- Dense or sparse
- Multiple solutions (e.g., shortest paths, MSTs, topological sorts)