Competitive Programming Week 7

Graph Algorithms: SSSP + MST

Alberta Collegiate Programming Contest (ACPC):

This year ACPC will take place on November 25th
We will be hosting a location here at U of C
(MS 1st floor)
There will be $2550 in prizes across two divisions
+ an open division (no registration required)

Sign Up + More Info:
https://www.eventbrite.ca/e/alberta-collegiate-programming-contest-2023-tickets-750029397117?aff=oddtdtcreator

Week 6 Solutions

Grid

Problem: Given a nxm grid where each cell contains an integer. Determine a path from the top left to bottom right cell only moving in the cardinal directions exactly the number of cells as the integer in the current cell

Solution: Create a graph where each cell is a vertex and cells are connected by the rule above. Perform bfs to find a shortest path

```cpp
typedef struct {
    int i, j, d;
} Entry;

int main() {
    ios::sync_with_stdio(false);
    cin.tie(NULL);

    int n, m;
    char v;
    cin >> n >> m;
    vector<vector<int>> grid(n);
    vector<vector<bool>> seen(n);
    for (int i = 0; i < n; i++) {
        grid[i].reserve(m);
        seen[i].reserve(m);
        for (int j = 0; j < m; j++) {
            cin >> v;
            grid[i][j] = (int) (v - '0');
            seen[i][j] = false;
        }
        cout << endl;
    }
    queue<Entry> queue;
    queue.push({0, 0, 0});
    while (!queue.empty()) {
        Entry p = queue.front();
        queue.pop();
        if (p.i < 0 || p.i >= n || p.j < 0 || p.j >= m) continue;
        if (seen[p.i][p.j]) continue;
        if (p.i == n - 1 && p.j == m - 1) {
            cout << p.d << endl;
            return 0;
        }
        seen[p.i][p.j] = true;
        int k = grid[p.i][p.j];
        int nd = p.d + 1;
        queue.push({p.i + k, p.j, nd});
        queue.push({p.i - k, p.j, nd});
        queue.push({p.i, p.j + k, nd});
        queue.push({p.i, p.j - k, nd});
    }
    cout << -1 << endl;
    return 0;
}
```

Flip Five

Problem: Given a 3x3 grid where each tile is either white or black. Determine a sequence of *clicks* where each click flips the tile clicked and the neighbouring tiles that will result in a grid that is all white

Solution: Let each configuration of the tiles define a state. Create a graph where the states are vertices and *clicks* are edges between states. The minimum clicks is then the shortest path from the start state to the end state (use bfs)

Note: defining an encoding for the states can make it easier code and potentially faster

```
typedef struct {
    short i;
    int d;
} Entry;

int main() {
    ios::sync_with_stdio(false);
    cin.tie(NULL);
    int n;
    cin >> n;

    while (n--) {
        short g = 0b111111111, s = 0;
        char c;
        for (int i = 0; i < 9; i++) {
            cin >> c;
            if (c == '.') g ^= 1 << i;
        }
        queue<Entry> q;
        unordered_set<short> seen;
        q.push({g, 0});
        while (!q.empty()) {
            short ng = q.front().i;
            int d = q.front().d;
            q.pop();
            if (seen.find(ng) != seen.end()) continue;
            if (ng == 0) {
                cout << d << endl;
                break;
            }
            seen.insert(ng);
            for (int i = 0; i < 9; i++) {
                short nng = ng;
                if (i / 3 > 0) nng ^= 1 << (i - 3);
                if (i / 3 < 2) nng ^= 1 << (i + 3);
                if (i % 3 > 0) nng ^= 1 << (i - 1);
                if (i % 3 < 2) nng ^= 1 << (i + 1);
                nng ^= 1 << i;
                q.push({nng, d + 1});
            }
        }
    }
    return 0;
}
```

Popularity Contest

Problem: Given a set of your friends, ordered by how successful you think they are, and the friends they are also friends with, determine the difference between each friends popularity and success

Solution: Kind of a trick question. No graphs are needed (although I think the intended solution does use a graph). You can just count the occurrences of each friend in the input and subtract the index of their success (1-indexed)

```
int main() {
    ios::sync_with_stdio(false);
    cin.tie(NULL);
    int n, m;
    cin >> n >> m;
    vector<int> v(n);
    int a, b;
    while (m--) {
        cin >> a >> b;
        v[a - 1]++;
        v[b - 1]++;
    }
    for (int i = 0; i < n; i++) {
        cout << v[i] - (i + 1);
        if (i != n - 1) cout << ' ';
    }
    cout << endl;
    return 0;
}
```

Prime Path

Problem: Given an 4-digit prime integer s and a 4-digit prime integer t determine the minimum sequence of prime integers where each stage differs by a single digit that starts at s and ends at t.

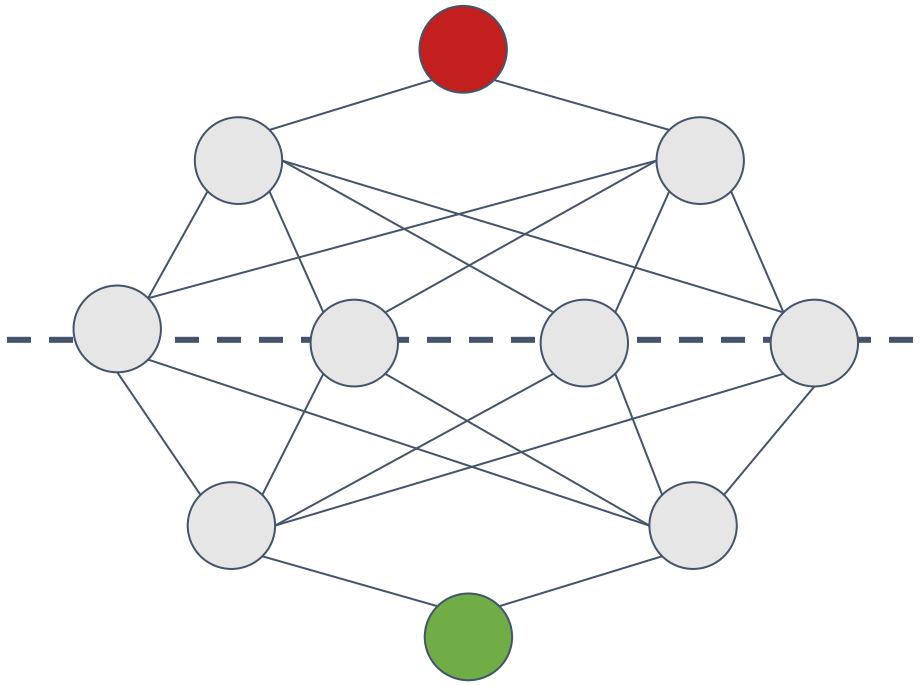
Solution: Use a prime sieve to find all 4-digit primes. Define a graph where each vertex is a 4-digit number where 2 vertices are connected if they are both prime and differ by one bit. Perform bfs on this graph

```
typedef struct {
    int i, d;
} Entry;

bool notprime[10000];

int bfs(int s, int t) {
    queue<Entry> q;
    unordered_set<int> seen;
    q.push({s, 0});
    while (!q.empty()) {
        int i = q.front().i;
        int d = q.front().d;
        q.pop();
        if (seen.find(i) != seen.end()) continue;
        if (i == t) {
            return d;
        }
        seen.insert(i);
        for (int j = 3; j >= 0; j--) {
            int p = pow(10, j);
            int e = (i % (10 * p) / p) * p;
            for (int k = 0; k < 10; k++) {
                if (k * p == e) continue;
                if (j == 3 && k == 0) continue;
                int n = i - e + k * p;
                if (!notprime[n]) q.push({n, d + 1});
            }
        }
    }
    return -1;
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(NULL);
    for (int i = 2; i <= 100; i++) {
        for (int j = 2 * i; j < 10000; j += i) {
            notprime[j] = true;
        }
    }
    int n, s, t;
    cin >> n;
    while (n--) {
        cin >> s >> t;
        int d = bfs(s, t);
        if (d >= 0) cout << d << endl;
        else cout << "Impossible" << endl;
    }

    return 0;
}
```

Pathfinding Algorithms

Meet in the Middle BFS

- Basic idea: BFS struggles on graphs that expand quickly/exponentially. When doing pathfinding through states, like many of last week's problems, this can become an issue. We can reduce the expansion by doing a modified version of BFS where we essentially do 2 in parallel, starting from both the start and target nodes. We then are looking for nodes that both BFS's have encountered instead of the target node. When the node is found we can reconstruct the paths to it, invert the one starting from the target node, and concatenate them



Dijkstra's algorithm

- Basic idea: a more powerful version of BFS that uses edge weight information
- Uses a priority queue of closest known vertices to the start
- Similarly to BFS, keep popping the priority queue to find the closest unprocessed vertex
  - For each neighbour, update the distance to it to the minimum of the current distance to it and the distance to the current vertex + edge weight connect current vertex to neighbour (DP-style "relaxation" update)
- https://visualgo.net/en/sssp
- Complexity: O((V + E) log V)

Bellman-Ford and Floyd-Warshall Algorithms

- Simpler algorithms based on nested for loops to find single source shortest path in the case of negative weights (Bellman-Ford) or to find the shortest path between all pairs of vertices (Floyd-Warshall)
- Not as common as BFS and Dijkstra's algorithm, but handy to know about

```
void bellman_ford(vector<vector<pair<int, int>>> adj_list, int s)
{
  vector<int> dist(adj_list.size(), 1000000000); dist[s] = 0;
  for (int i = 0; i < adj_list.size()-1; ++i)
    for (int u = 0; u < adj_list.size(); ++u)
      if (dist[u] != 1000000000)
        for (auto &[v, w] : adj_list[u])
          dist[v] = min(dist[v], dist[u] + w);
}

void floyd_warshall(vector<vector<int>> adj_matrix) {
  for (int k = 0; k < adj_matrix.size(); ++k)
    for (int i = 0; i < adj_matrix.size(); ++i)
      for (int j = 0; j < adj_matrix.size(); ++j)
        adj_matrix[i][j] = min(adj_matrix[i][j],
                               adj_matrix[i][k] + adj_matrix[k][j]);
}
```

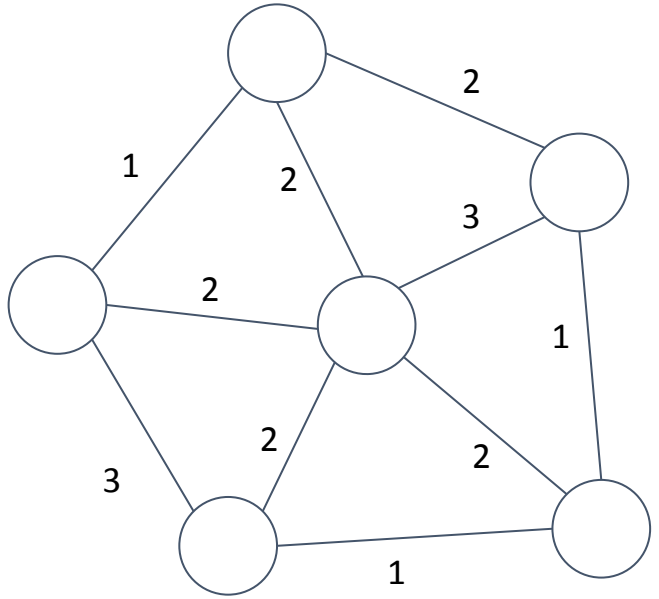
Comparison (Competitive Programming 4 Book Table 4.4)

Graph → Criteria ↓	BFS O(V + E)	Dijkstra's O((V + E) log V)	Bellman-Ford O(V*E)	Floyd-Warshall $O(V^3)$
Max Size	$V + E \leq 10^8$	$V + E \leq 10^6$	$V*E \leq 10^8$	$V \leq 450$
Unweighted	Best	Okay	Bad	Bad in general
Weighted	Wrong Answer	Best	Okay	Bad in general
Negative weight	Wrong Answer	Okay but slower	Okay	Bad in general
Negative cycle	Cannot detect	Cannot detect	Can detect	Can detect
Small graph	W. A. if weighted	Overkill	Overkill	Best

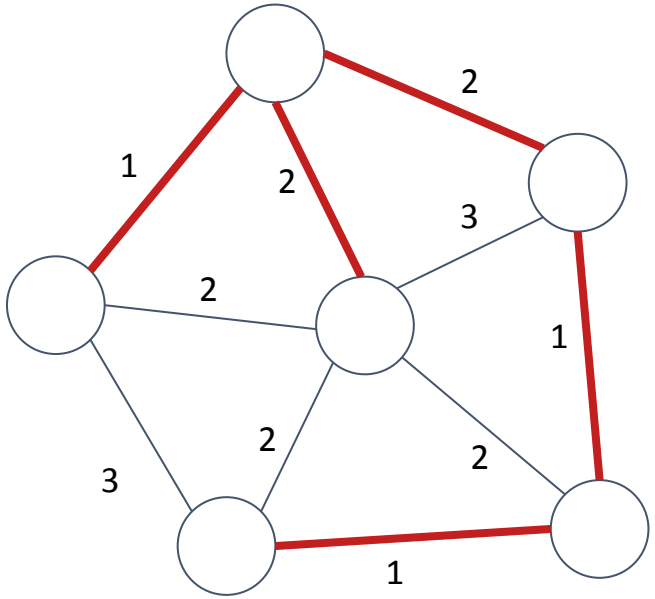
Minimum Spanning Trees

MST's

- A Spanning Tree is a subtree of a graph that contains all the vertices of the graph
- A Minimum Spanning Tree of a connected weighted graph is a weighted spanning tree of the graph where the sum of all of the weights in the tree is minimal
- There are two main algorithms for constructing MST, which we will cover, both are **greedy** algorithms



Original Graph



Original Graph with an MST highlighted red

Kruskal's Algorithm

Idea: Starting with an empty graph choose min cost edges to join trees together until a single tree is formed

- **Greedy Heuristic:**
  Add the lowest cost edge that does **not** introduce a cycle to the current graph

When no more edges can be added you have an MST

Runtime: O(|E|log|E|) / O(|E|log|V|)

```
def kruskalMST(adjMatrix: int[][]):
    res = int[][]
    edges = pair<int>[]
    sort(edges by weight)
    for edge (u, v) in edges:
        if dfs(res, u,v) == null: // pick any traversal
            res[u][v] = adjMatrix[u][v]
            res[v][u] = adjMatrix[u][v]
    return res
```

Prim's Algorithm

Idea: Construct a tree and grow it while ensuring min cost

- **Greedy Heuristic:**
  Connect a vertex not in the current tree with minimum cost edge to the current tree

Do this for each vertex and you have a MST

Implementation details:

- you can assign each vertex a value being the current min cost edge from the current tree to the vertex.
- you will likely have to track the parent of the min cost edge when storing that information
- you have to use a heap implementation that allows for updating costs

Runtime: O(|V|log|V|) / O(|E|log|V|)

Can be improved to O(E + log |V|) using a Fibonacci Heap (will discuss later)

```
def primMST(adjMatrix: int[][]):
    res = int[][]
    mins = int[]
    parent = int[]
    for i from 1 to adjMatrix.size:
        mins[i] = inf
    mins[1] = 0
    seen = bool[]
    minheap = MinHeap(sort by mins[val])
    minHeap.push(1)
    while !minHeap.empty:
        v = minHeap.pop()
        seen[v] = true
        res[parent[v]][v] = adjMatrix[parent[v]][v]
        res[v][parent[v]] = adjMatrix[parent[v]][v]
        for i from 1 to adjMatrix.size:
            if !seen[i]
            if adjMatrix[v][i] < mins[i]:
                mins[i] = adjMatrix[v][i]
                parent[i] = v
                minHeap.push(i)
    return res
```

https://open.kattis.com/contests/h6zwy2

(or look up “CPC Fall 2023 Practice Contest Week 7” in the Kattis contest list)

Feel free to ask questions until 7pm, and then throughout the week on Discord!

Register for ACPC!