

INF280

Graph Algorithms

Florian Brandner

Contents

Union-Find

Minimum Spanning Trees

Flows in Graphs

Assignment Problems and Matchings

Union-Find

A data structure to track equivalence relations between elements:

- Elements are partitioned into non-overlapping sets
 - Initially only pairwise relations are known (i.e., X and Y are in the same set)
 - From pairwise relations, deduce the global partitioning step-wise
- Basic idea:
 - Represent partitions as trees
 - Merge trees when a new pairwise relation is discovered

Union-Find (2)

Two operations to update/query the union-find data structure:

- Union(x, y):
Add a new pairwise relation between x and y and update the Union-Find structure to put them in the same set
- Find(x):
Get the (current) representative of the set for element x

<https://visualgo.net/ufds>

Union-Find using Ranks and Path Compression

```
map<int, pair<int, unsigned int> > Sets; // map to parent & rank
void MakeSet(int x) {
    Sets.insert(make_pair(x, make_pair(x, 0)));
}
int Find(int x) {
    if (Sets[x].first == x) return x; // Parent == x ?
    else return Sets[x].first = Find(Sets[x].first); // Get Parent
}
void Union(int x, int y) {
    int parentX = Find(x), parentY = Find(y);
    int rankX = Sets[parentX].second, rankY = Sets[parentY].second;
    if (parentX == parentY) return;
    else if (rankX < rankY)
        Sets[parentX].first = parentY;
    else
        Sets[parentY].first = parentX;
    if (rankX == rankY)
        Sets[parentX].second++;
}
```

Contents

Union-Find

Minimum Spanning Trees

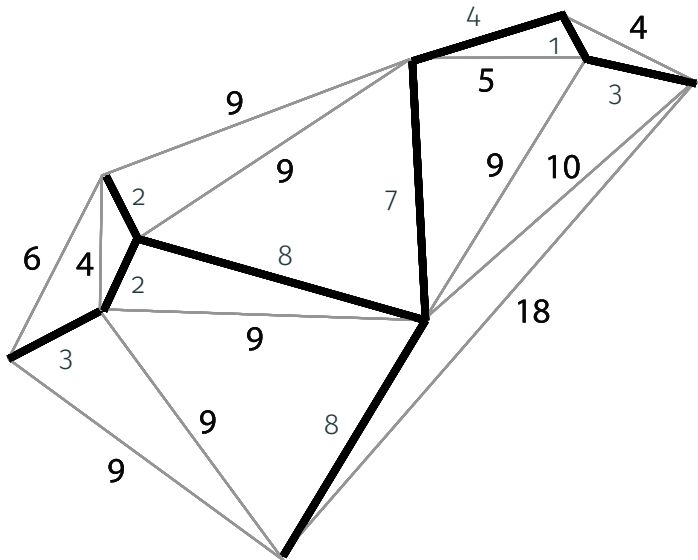
Flows in Graphs

Assignment Problems and Matchings

Spanning Trees

- Subgraph of undirected connected graph that forms a tree
- The subgraph contains each node of the original graph
- Computing spanning trees
 - Many possible spanning trees exist
 - Any depth-first traversal gives a spanning tree
- In weighted graphs one might need a **minimum spanning tree**
 - A spanning tree as defined above
 - Require that the total sum of edge weights is **minimal** (i.e., no other spanning tree with a lower total sum exists)

Example: Minimum Spanning Trees



Kruskal's Algorithm

```
vector<pair<int, pair<int, int> > > Edges;
set<pair<int,int> > A; // Final minimum spanning tree

void Kruskal() {
    for(int u=0; u < N; u++)
        MakeSet(u); // Initialize Union-Find

    sort(Edges.begin(), Edges.end()); // Sort edges by weight

    for(auto tmp : Edges) {
        auto edge = tmp.second;
        if (Find(edge.first) != Find(edge.second)) {
            Union(edge.first, edge.second); // update Union-Find
            A.insert(edge); // include edge in MST
        }
    }
}
```

Contents

Union-Find

Minimum Spanning Trees

Flows in Graphs

Assignment Problems and Matchings

Flow Networks

- Flow networks are weighted directed graphs
- Edge weights denote the capacity of edges
 - Current in an electric circuit
 - Water in pipes
 - Trains on a railroad
- Question: What is the **maximum flow** between nodes s and t ?
 - Assign flow to each edge respecting the edge's capacity
 - For each node, the combined in/out flows must be equal

Maximum Flow / Minimum Cut

Maximum flow is limited by a **cut** separating s and t

- Basic idea behind cuts:
 - Partition the network's nodes into two sets S and T
 - S contains s while T contains t
 - Edges (u, v) with $u \in S$ and $v \in T$ are in the **cut** between S and T
 - The edges in the cut completely separate the two sets
 \implies Removing those edges gives a maximum flow of zero
- Link between cuts and flows:
 - The combined edge weights of a cut bound the flow from s to t
 - The **maximum flow** is thus limited by a **minimum cut**

Edmonds-Karp Algorithm

```
// find path from s to t in G, return true if such a path exists
bool BFS(int G[MAXN][MAXN], int s, int t, int Predecessor[MAXN]);

int EdmondsKarp(int G[MAXN][MAXN], int s, int t) {
    int GRes[MAXN][MAXN]; // residual graph
    copy_n((int*)G, MAXN*MAXN, (int*)GRes); // copy original graph

    int Predecessor[MAXN];
    int Maxflow = 0;
    while (BFS(GRes, s, t, Predecessor)) { // find residual path
        int Bottleneck = MAXFLOW; // get minimal flow of residual path
        for (int v = t; v != s; v = Predecessor[v])
            Bottleneck = min(Bottleneck, GRes[Predecessor[v]][v]);
        for (int v = t; v != s; v = Predecessor[v]) {
            // decrease capacity along residual path
            GRes[Predecessor[v]][v] -= Bottleneck;
            GRes[v][Predecessor[v]] += Bottleneck;
        }
        Maxflow += Bottleneck;
    }
    return Maxflow;
}

https://visualgo.net/maxflow
```

Contents

Union-Find

Minimum Spanning Trees

Flows in Graphs

Assignment Problems and Matchings

Assignment Problems and Matchings

- Represented as **bipartite graphs**:
 - Nodes are partitioned into two disjoint sets X and Y
 - Edges always connect nodes from both sets
 - $(G = (X \cup Y, E),$ where $E \subseteq X \times Y$)
- Basic idea:
 - Search the best assignment of elements in X to elements in Y
 - Each element may appear only in one assignment
- Problem variants
 - Maximize matching cardinality (Hopcroft-Karp – on next slide)
 - Maximize matching cost in weighted graphs

Hopcroft-Karp (data structures)

```
// Artificial node (unused otherwise) -- end of augmenting path
#define NIL 0

// "Infinity", i.e., value larger than min(|X|, |Y|)
#define INF numeric_limits<unsigned int>::max()

// Partitions X and Y
vector<int> X, Y;

// Neighbors in Y of nodes in X
vector<int> Adj[MAXX];

// Matching X-Y and Y-X
int PairX[MAXX];
int PairY[MAXY];

// Augmenting path lengths
unsigned int Dist[MAXX];
```


Hopcroft-Karp (main)

```
int HopcroftKarp() {
    fill_n(PairX, MAXX, NIL);    // initialize: empty matching
    fill_n(PairY, MAXY, NIL);

    int Matching = 0;          // count number of edges in matching

    while (BFS()) {           // find all shortest augmenting paths
        for(auto x : X)       // update matching cardinality
            if (PairX[x] == NIL && // node not yet in matching?
                DFS(x))         // does an augmenting path start at x?
                Matching++;
    }
    return Matching;
}
```

Hopcroft-Karp (BFS)

```
bool BFS() {
    queue<int> Q;
    Dist[NIL] = INF;
    for(auto x : X) { // start from nodes that are not yet matched
        Dist[x] = (PairX[x] == NIL) ? 0 : INF;
        if (PairX[x] == NIL)
            Q.push(x);
    }
    while (!Q.empty()) { // find all shortest paths to NIL
        int x = Q.front(); Q.pop();
        if (Dist[x] < Dist[NIL]) // can this become a shorter path?
            for (auto y : Adj[x])
                if (Dist[PairY[y]] == INF) {
                    Dist[PairY[y]] = Dist[x] + 1; // update path length
                    Q.push(PairY[y]);
                }
    }
    return Dist[NIL] != INF; // any shortest path to NIL found?
}
```

Hopcroft-Karp (DFS)

```
bool DFS(int x) {
    if (x == NIL)
        return true; // reached NIL

    for (auto y : Adj[x])
        if (Dist[PairY[y]] == Dist[x] + 1 &&
            DFS(PairY[y])) { // follow trace of BFS
            PairX[x] = y; // add edge from x to y to matching
            PairY[y] = x;
            return true;
        }
    Dist[x] = INF;
    return false; // no augmenting path found
}
```