INF280: Competitive programming

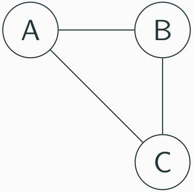
Basic graph traversals

Louis Jachiet

Introduction

You all know graphs:

- Set of nodes N
- Set of edges $E \subseteq N \times N$
- Edges can be undirected or directed, i.e., $(a, b) \neq (b, a)$



$$N \quad \{A, B, C\}$$
$$E \quad \{(A, B), (A, C), (B, C)\}$$

Data Structures

Several options to represent graphs:

- Adjacency matrix:
  - bool G[MAXN][MAXN];
  - G[x][y] is true if an edge between node x and y exists
  - Replace bool by int to represent weighted edges
- Adjacency list:
  - vector<int> Adj[MAXN];
  - y is in Adj[x] if an edge between node x and y exists
  - Pairs to represent weights
- Edge list:
  - vector<pair<int, int> > Edges;
  - Edges contains a pair of nodes if an edge exists between them
- Nodes and edges may also be custom structs or classes

Simple Traversals

Simple Traversals

Depth-First Search

Depth-First Search

Visit each node in the graph once:

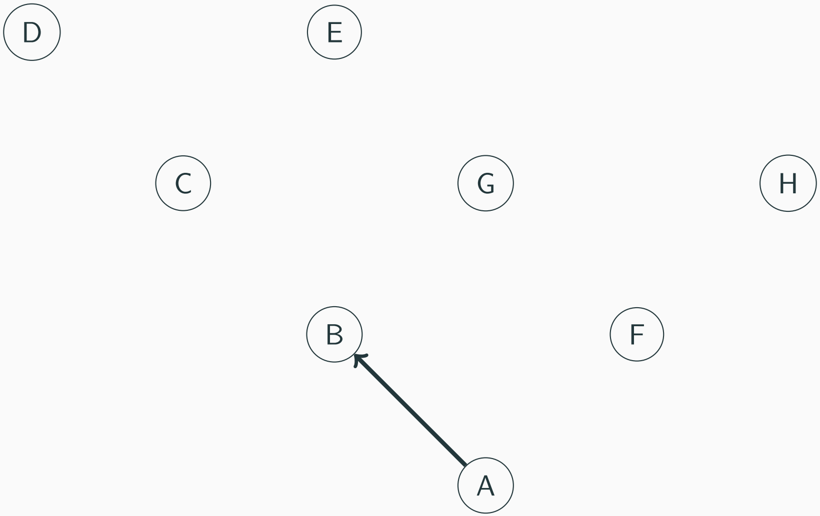
- Recursive implementation below
- Manage stack yourself for iterative version
- First visit child nodes then siblings

```
int state[ID_NODE_MAX] ;
const int NOT_VISITED = 0, IN_VISIT = 1 , VISITED = 2 ;
void dfs(int node) {
  if(state[node] == NOT_VISITED) {
    state[node] = IN_VISIT ;
    for(auto v : nxt[node])
      dfs(v);
    state[node] = VISITED ;
  }
}
```

Applications of DFS

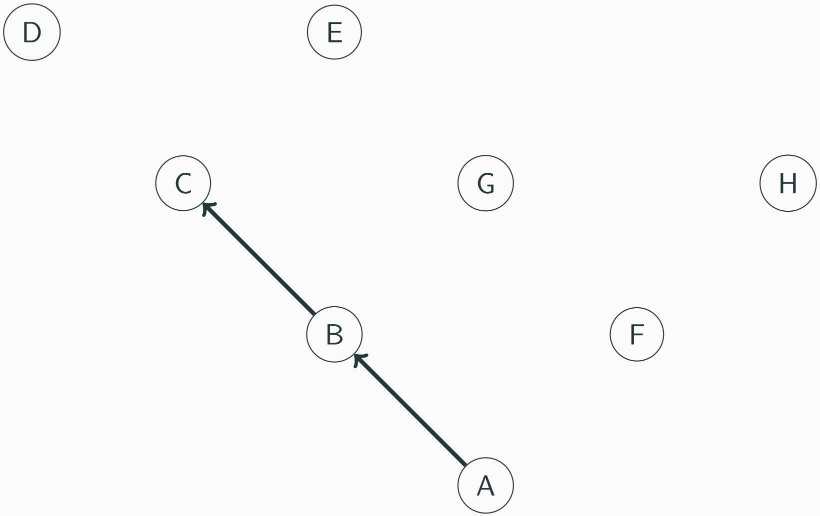
- Determine a topological order of nodes
- Detect if a cycle exists
- Check reachability between nodes
- Decompose graph into connected components
- Decompose graph in strongly connected components
- Examples: `https://visualgo.net/dfsbfs`

Tarjan representation of DFS



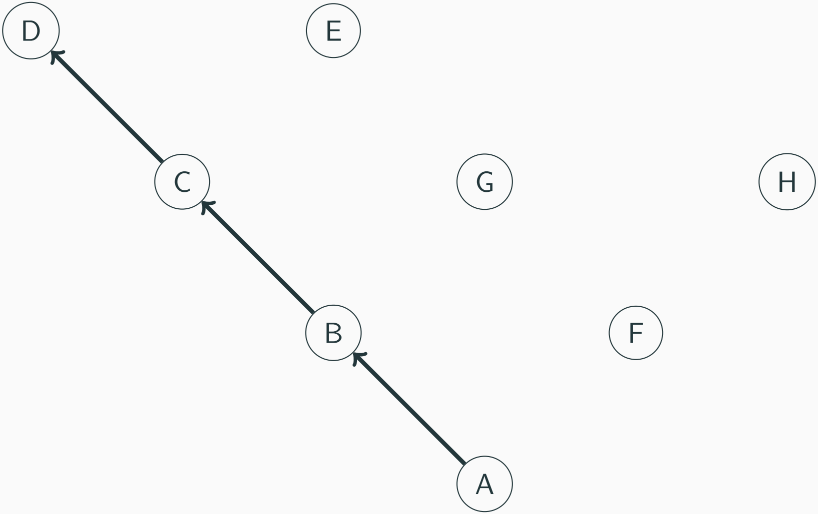
Useful to understand what happens...

Tarjan representation of DFS



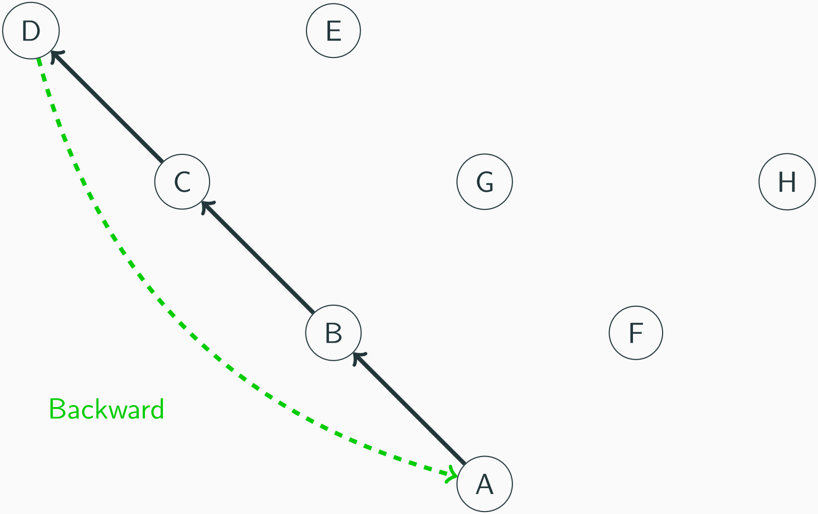
Useful to understand what happens...

Tarjan representation of DFS



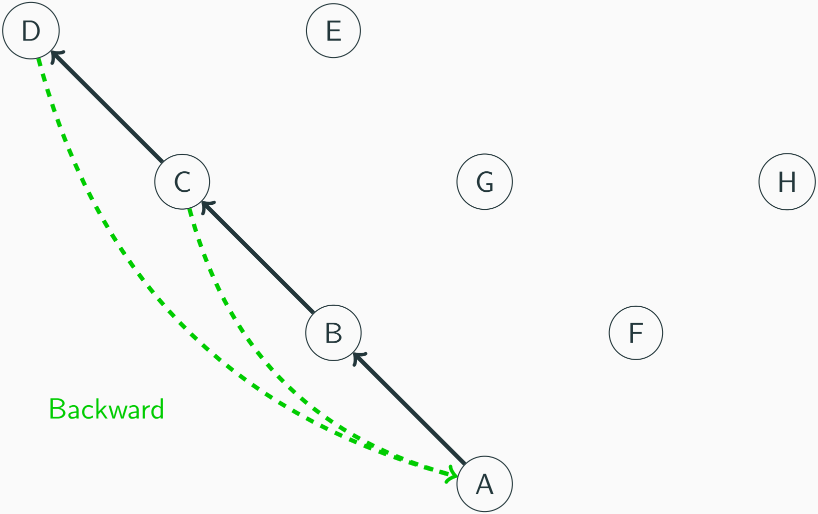
Useful to understand what happens...

Tarjan representation of DFS



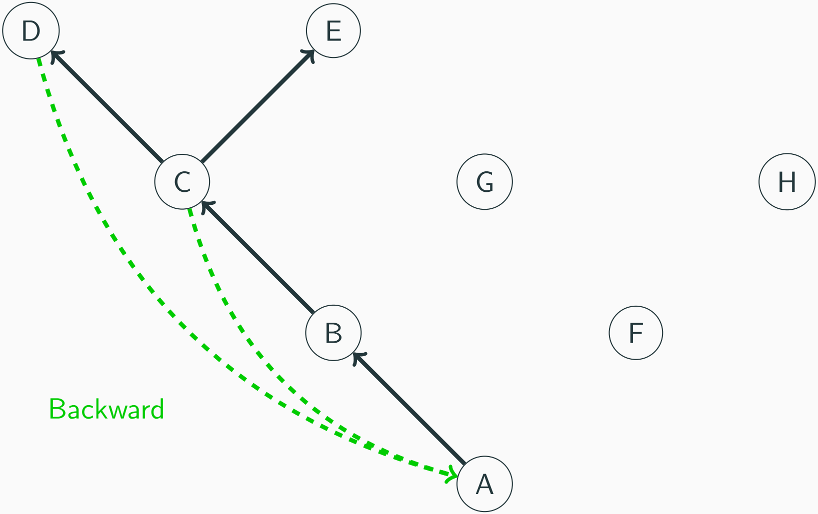
Useful to understand what happens...

Tarjan representation of DFS



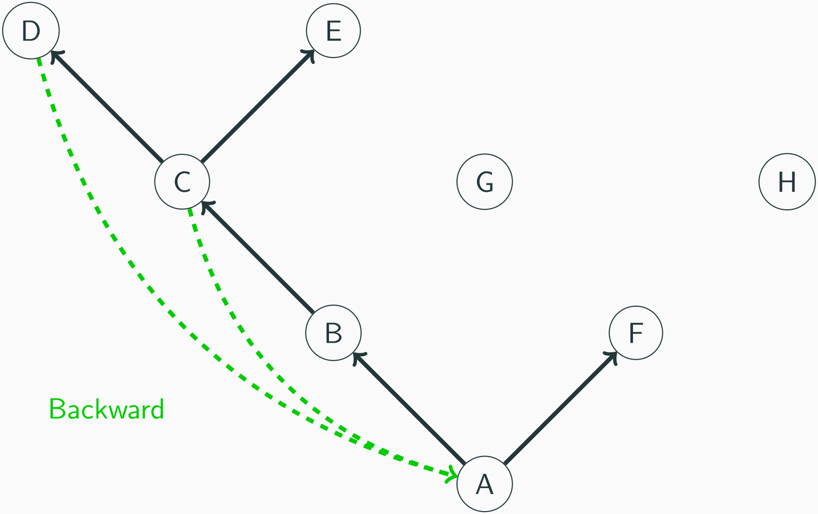
Useful to understand what happens...

Tarjan representation of DFS



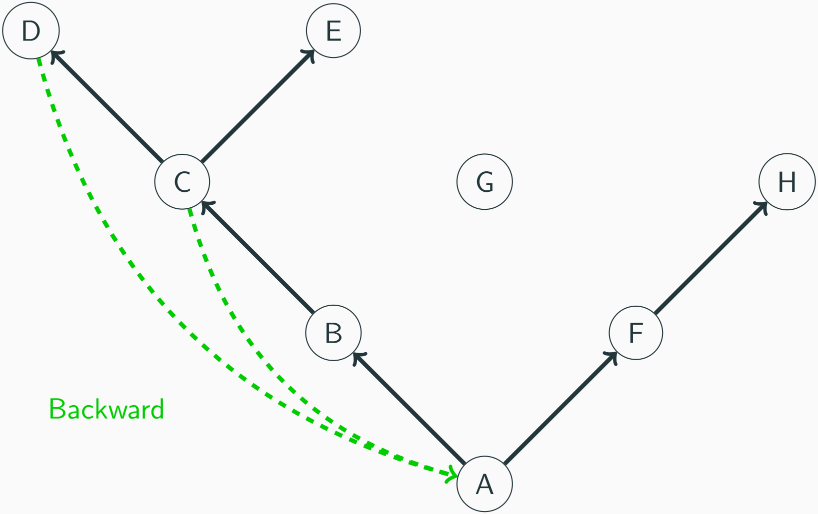
Useful to understand what happens...

Tarjan representation of DFS



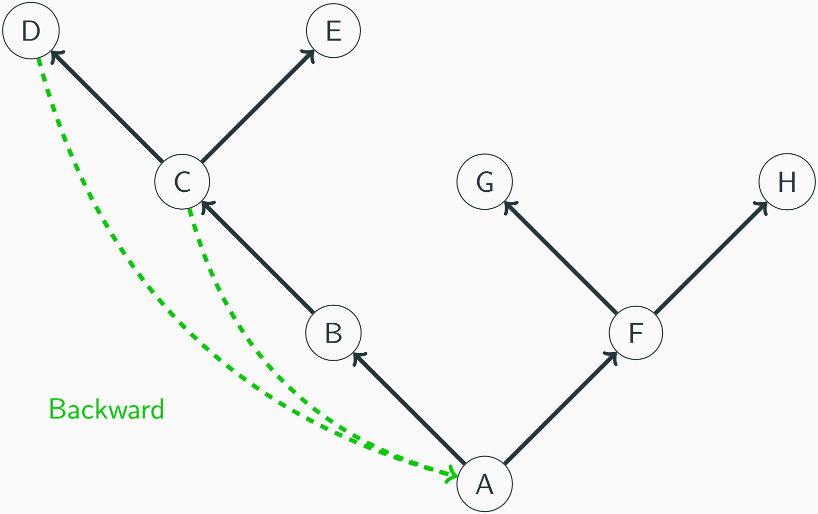
Useful to understand what happens...

Tarjan representation of DFS



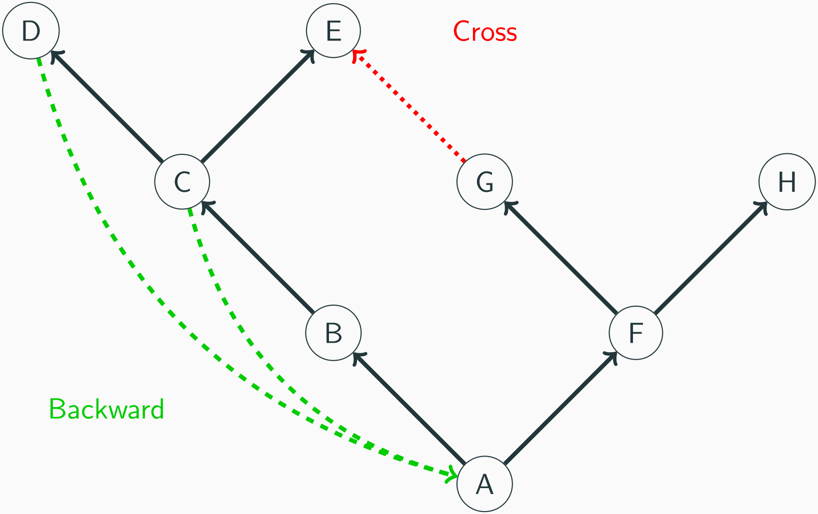
Useful to understand what happens...

Tarjan representation of DFS



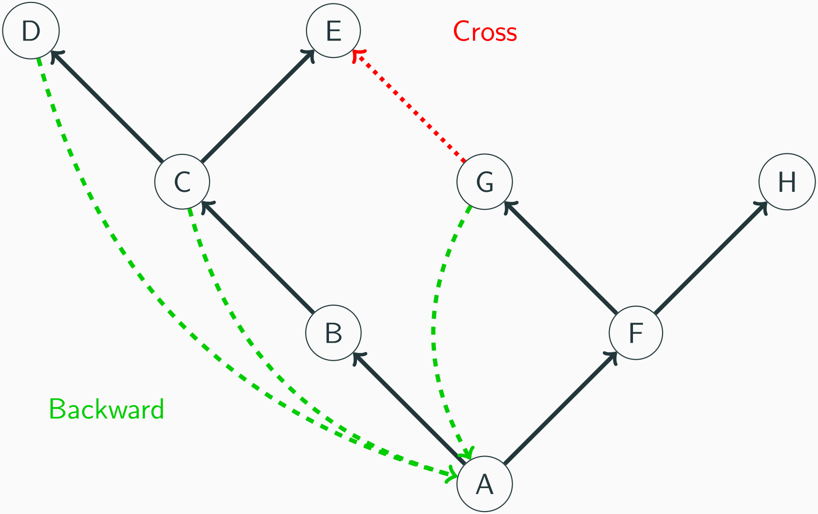
Useful to understand what happens...

Tarjan representation of DFS



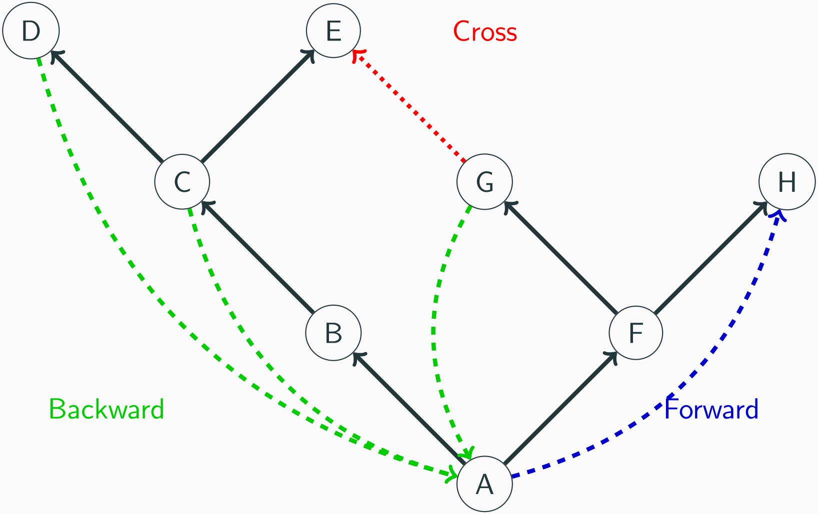
Useful to understand what happens...

Tarjan representation of DFS



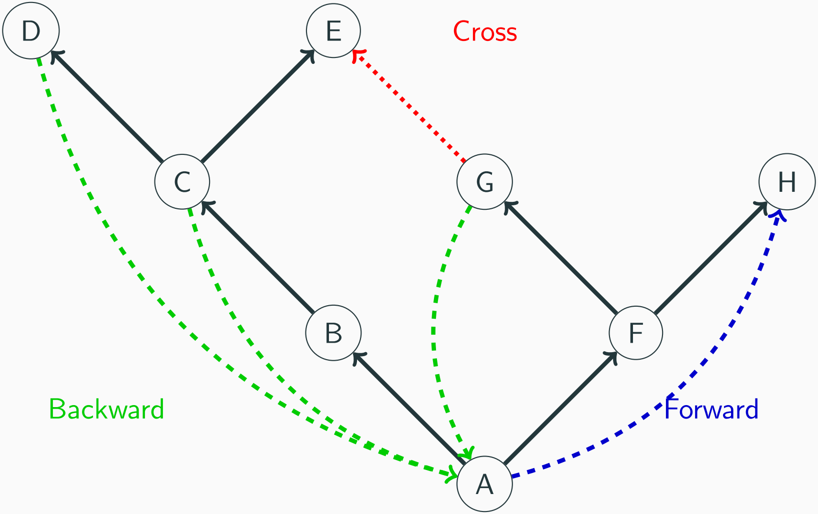
Useful to understand what happens...

Tarjan representation of DFS



Useful to understand what happens...

Tarjan representation of DFS



Exercise: compute Strongly Connected Component

Simple Traversals

Breadth-First Search

Breadth-First Search

Visit each node in the graph once:

- Similar to DFS, but replaces stack by queue

```
int seen[NB_NODE_MAX] ;
void bfs(int start) {
  vector<int> todo = {start} ;
  seen[start] = true ;
  for(size_t id = 0 ; id < todo.size() ; id++)
    for(auto v : nxt[todo[id]])
      if(!seen[v]) {
        seen[v] = true;
        todo.push_back(v);
      }
}
```

Applications of BFS

- Shortest path search
  - Stop processing when a given node d was found
  - Minimizes number of hops, i.e., all edges have same weight or 0-1 Weights
  - Generalization follows next
- Examples: `https://visualgo.net/dfsbfs`

Simple Traversals

0-1 Breadth-First Search

Breadth-First Search with edges of bounded distance

```
vector<int> nodes_at[MAX_DISTANCE];
void bfs(int start) {
  fill(dist,dist+NB_NODES_MAX,INF);
  nodes_at[0] = {start} ;
  dist[start] = 0 ;
  for(int cur_dist = 0 ; cur_dist < MAX_DISTANCE ; cur_dist++ )
    for(size_t id = 0 ; id < nodes_at[cur_dist].size() ; id++) {
      const int node = nodes_at[cur_dist][id] ;
      if(dist[node] == cur_dist)
        for(auto [neigh,len] : nxt[node])
          if(dist[neigh] > cur_dist+len) {
            dist[neigh] = cur_dist+len ;
            nodes_at[dist[neigh]].push_back(neigh);
          }
    }
}
```

Finding Paths

Finding Paths

Dijkstra

Dijkstra

- Dijkstra's algorithm generalizes BFS
- Constraint: all edges need to have non-negative weights
- Use a priority queue to choose which node to examine next

Finding Paths

Bellman-Ford

Bellman-Ford

- Dijkstra's algorithm is limited to non-negative edge weights
- Bellman-Ford extends this to general edge weights

Bellman-Ford

- Dijkstra's algorithm is limited to non-negative edge weights
- Bellman-Ford extends this to general edge weights

Bellman-Ford DP problem: "$q(n, k)$ is the minimal distance of n from the source node using k intermediate edges"

Bellman-Ford

- Dijkstra's algorithm is limited to non-negative edge weights
- Bellman-Ford extends this to general edge weights

Bellman-Ford DP problem: "$q(n, k)$ is the minimal distance of n from the source node using k intermediate edges"

Bellman-Ford can also be seen as a way to solve a linear system with inequalities of the form: $x_i + c_i \leq y_i$

Bellman-Ford Algorithm

```
int from[MAX_NB_EDGES], to[MAX_NB_EDGES],weight[MAX_NB_EDGES];
int dist[MAX_PATH_LENGTH+1][MAX_NB_NODES];
bool BellmanFord(int root) {
  fill(dist[0],dist[MAX_PATH_LENGTH],INF);
  dist[0][root] = 0;
  for(int len = 0 ; len < MAX_PATH_LENGTH ; len++)
    for (int e = 0 ; e < nb_edges ; e++)
      dist[len+1][to[e]] = min(dist[len+1][to[e]],
                               dist[len][from[e]]+weight[e]);
  // to be explained later; check for negative cycles
  return dist[MAX_PATH_LENGTH][target];
}
```

- replace `dist[l][n]` with `dist[n]` $= min_l($`dist[l][n]`$)$
- MAX_PATH_LENGTH is at most nb_nodes long

Bellman-Ford Algorithm

```
int dist[MAX_NB_NODES];
void BellmanFord(int root, int target) {
  fill(dist, dist+MAX_NB_NODES, INF);
  dist[root] = 0;
  for(int k = 0 ; k < nb_nodes - 1 ; k++)  // N - 1 times
    for (int i = 0 ; i < nb_edges ; i++)
      dist[to[i]] = min(dist[to[i]], dist[from[i]]+weight[i]);
}
```

Bellman-Ford Algorithm

```
bool detect_negative_cycle_BellmanFord(int root, int target) {
  fill(dist, dist+MAX_NB_NODES, INF);
  dist[root] = 0;
  for(int k = 0 ; k < nb_nodes - 1 ; k++)  // N - 1 times
    for (int i = 0 ; i < nb_edges ; i++)
      dist[to[i]] = min(dist[to[i]], dist[from[i]]+weight[i]);
 // now time to check for negative cycles:
  int dist_target = dist[target]; // copy distance
  for(int k = 0 ; k < nb_nodes - 1 ; k++)  // N - 1 times
    for (int i = 0 ; i < nb_edges ; i++)
      dist[to[i]] = min(dist[to[i]], dist[from[i]]+weight[i]);
  return dist[target] < dist_target ; // negative cycle?
}
```

Finding Paths

Floyd-Warshall

Floyd-Warshall

- Dijkstra and Bellman-Ford compute shortest paths
    - From a single source (root)
    - To all other (reachable) nodes
    - This is known as: single-source shortest path problem
- Floyd-Warshall extends this to compute the shortest paths between all pairs of nodes
- This is known as: all-pairs shortest path problem

Floyd-Warshall

- Dijkstra and Bellman-Ford compute shortest paths
  - From a single source (root)
  - To all other (reachable) nodes
  - This is known as: single-source shortest path problem
- Floyd-Warshall extends this to compute the shortest paths between all pairs of nodes
- This is known as: all-pairs shortest path problem

Floyd-Warshall answers the DP problem: "q(start,end,pivot): what is the shortest path between `start` and `end` going through intermediate nodes 1..`pivot`?"

Floyd-Warshall Algorithm

```
int dist[MAX_NB_NODES][MAX_NB_NODES];
// We store q(start,end,pivot) in dist[start][end]
void FloydWarshall() {
  // initialize distance
  fill(dist[0],dist[MAX_NB_NODES],INF);
  for (int e = 0 ; e < nb_edges ; e++)
    dist[fr[e]][to[e]] = min(dist[fr[e]][to[e]], weight[e]);
  // now compute
  for(int pivot = 0 ; pivot < nb_nodes ; pivot++)
    for(int start = 0 ; start < nb_nodes ; start++)
      for(int end = 0 ; end < nb_nodes ; end++)
        dist[start][end] = min(dist[start][end],
                    dist[start][pivot]+dist[pivot][end]);
}
// WARNING, the order of the loops is important!!!
// for french speakers Pivot Début Fin => PDF algorithm
```

Finding Paths

Improvements

Keeping track of the path

We only considered the length of the path so far:

- All of the above algorithms can track the actual shortest path
- This allows to *print* each edge/node along the path
- Basic idea:
  - Introduce an array int Predecessor[MAXN]
    (Use two-dimensional array for Floyd-Warshall)
  - Updated whenever Dist[v] changes
  - Simply set to the new predecessor u

Heuristics – A* Search

Heuristics may speed-up the path search

- Bellman-Ford and Floyd-Warshall equally explore all possibilities
- Dijkstra *prefers* nodes with shorter distance
- Basic idea behind A* Search:
  - Extension to Dijkstra's algorithm
  - Introduce an estimation of the remaining distance
  - Prefer nodes with minimal estimated *remaining* distance
- Advantages
  - May converge faster than Dijkstra
  - Can be used to compute approximate solutions (trading speed for precision)

Eulerian Circuits

Eulerian path

Use every edge of a graph exactly once. Start and end may differ

Eulerian circuit

Use every edge exactly once. Start and end at the same node

Idea of the algorithm

If you enter a node of even degree you are sure that you can go out, decreasing the degree of unused by 2. This gives a first path/circuit. If your graph is connected, you can have remaining edges unexplored, but at least one in your current path, so you can re-explore them.

Eulerian Circuits

```
vector<int> path, nxt[NB_NODES_MAX] ;
set<pair<int,int>> used ;
void eulerian_path(int cur_node, int target) {
  if(cur_node != target)
    for(int n : nxt[cur_node])
      if(!used.count({n,cur_node})) {
        used.insert({cur_node,n});
        used.insert({n,cur_node});
        eulerian_path(n,target) ;
        target = cur_node ;
      } // target is cur_node after first pass
  path.push_back(cur_node) ;
}
```

We will see more graph algorithms next week...