INF280
Graph Traversals & Paths

Florian Brandner

7 March 2017

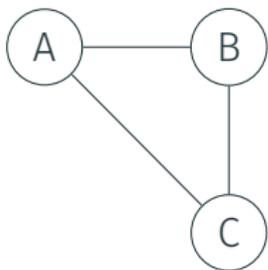## Contents

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

Contents

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

```cpp
bool Visited[MAXN] = {};
void DFS(int u) {
  if (Visited[u])
    return;
  Visited[u] = true;
  // maybe do something with u (pre-order) ...
  for (auto v : Adj[u])
    DFS(v);
  // or do something here (post-order)
}
```

Applications of DFS

- Determine a topological order of nodes
  - Only works for graphs without cycles (i.e., $x \rightarrow y \rightarrow z \rightarrow x$)
  - Add visited node at the head of an ordered list
    (at the end of DFS: `ordering.push_front(u)`)
- Detect if a cycle exists:
  - Check if the currently visited node is on the stack
    A) Use three states for `Visited` array:
       `UNVISITED`, `ONSTACK`, `VISITED`
    B) Explicitly search in the stack of the iterative algorithm
- Examples: https://visualgo.net/dfsbfs

Breadth-First Search

Visit each node in the graph once:

- Similar to DFS, but replaces stack by queue

```cpp
queue<int> Q;
bool Visited[MAXN] = {};
void BFS(int root) {
  Q.push(root);
  while (!Q.empty()) {
    int u = Q.front();
    Q.pop();
    if (Visited[u])
      continue;
    Visited[u] = true;
    for (auto v : Adj[u])
      Q.push(v);            // usually do something with 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
  - Generalization follows next
- Examples: `https://visualgo.net/dfsbfs`

Contents

Dijkstra

BFS can only be used if edge weights are uniform

- Dijkstra's algorithm generalizes this
- Constraint: all edges need to have non-negative weights
- Use a priority queue to choose which node to examine next
  - Would require a function to **update** the priority of an element
    - Would need to update order in the priority queue
  - We'll use the standard **priority_queue** of STL
    - Simply insert a new element in the queue (no update)
    - Ok since priorities decrease monotonically
    - This slightly diverges from Dijkstra's algorithm
- May revisit nodes several times

Dijkstra's Algorithm

```cpp
unsigned int Dist[MAXN];
typedef pair<unsigned int, int> WeightNode;              // weight goes first
priority_queue<WeightNode, std::vector<WeightNode>,
               std::greater<WeightNode> > Q;
void Dijkstra(int root) {
  fill_n(Dist, MAXN, MAXLEN);
  Dist[root] = 0;
  Q.push(make_pair(0, root));
  while (!Q.empty()) {
    int u = Q.top().second;                // get node with least priority
    Q.pop();
    for (auto tmp : Adj[u]) {
      int v = tmp.first;
      unsigned int weight = tmp.second;
      if (Dist[v] > Dist[u] + weight) {              // shorter path found?
        Dist[v] = Dist[u] + weight;
        Q.push(make_pair(Dist[v], v));     // simply push, no update here
      }
    }
  }
}
```

<div align="center">https://visualgo.net/sssp</div>

Bellman-Ford

- Dijkstra's algorithm is limited to non-negative edge weights
- Bellman-Ford extends this to general edge weights
- Constraint: no cycle with negative total costs
- May again revisit nodes several times

Bellman-Ford Algorithm

```cpp
unsigned int Dist[MAXN];
void BellmanFord(int root) {
  fill_n(Dist, MAXN, MAXLEN);
  Dist[root] = 0;
  for(int u=0; u < N - 1; u++) {  // just iterate N - 1 times
    for (auto tmp : Adj[u]) {
      int v = tmp.first;          // same as Dijkstra before
      unsigned int weight = tmp.second;
      Dist[v] = min(Dist[v], Dist[u] + weight);
    }
  }
}
                https://visualgo.net/sssp
```

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 Algorithm

```cpp
int Dist[MAXN][MAXN];
void FloydWarshall() {
  fill_n((int*)Dist, MAXN*MAXN, MAXLEN);
  for(int u=0; u < N; u++) {
    Dist[u][u] = 0;
    for (auto tmp : Adj[u])
      Dist[u][tmp.first] = tmp.second;
  }
  for(int k=0; k < N; k++)        // check sub-path combinations
    for(int i=0; i < N; i++)
      for(int j=0; j < N; j++)              // concatenate paths
        Dist[i][j] = min(Dist[i][j], Dist[i][k] + Dist[k][j]);
}
```

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)

Contents

Eulerian Circuits

Some definitions related to **undirected** graphs:

- 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**

- Conditions to find Eulerian **path**:
  - All nodes have even degree **or**
  - Precisely two nodes have odd degree

- For Eulerian **circuit**, all nodes must have even degree

Hierholzer's Algorithm for Eulerian Paths (assuming they exist)

```cpp
vector<int> Circuit;
void Hierholzer() {
  int v = 0;           // find node with odd degree, else start with node 0
  for (int u=0; u < N && v == 0; u++)
    if (Adj[u].size() & 1)
      v = u;                                // node with odd degree
  stack<int> Stack;
  Stack.push(v);
  while (!Stack.empty()) {
    if (!Adj[v].empty()) {               // follow edges until stuck
      Stack.push(v);
      int tmp = *Adj[v].begin();
      Adj[v].erase(tmp);                 // remove edge, modifying graph
      Adj[tmp].erase(v);
      v = tmp;
    } else {                           // got stuck: stack contains a circuit
      Circuit.push_back(v);            // append node at the end of circuit
      v = Stack.top();       // backtrack using stack, find larger circuit
      Stack.pop();
    }
  }
            https://www-m9.ma.tum.de/graph-algorithms/hierholzer
}
```