Graph Algorithms

DFS, BFS, Dijkstra

Benjamin Schmid

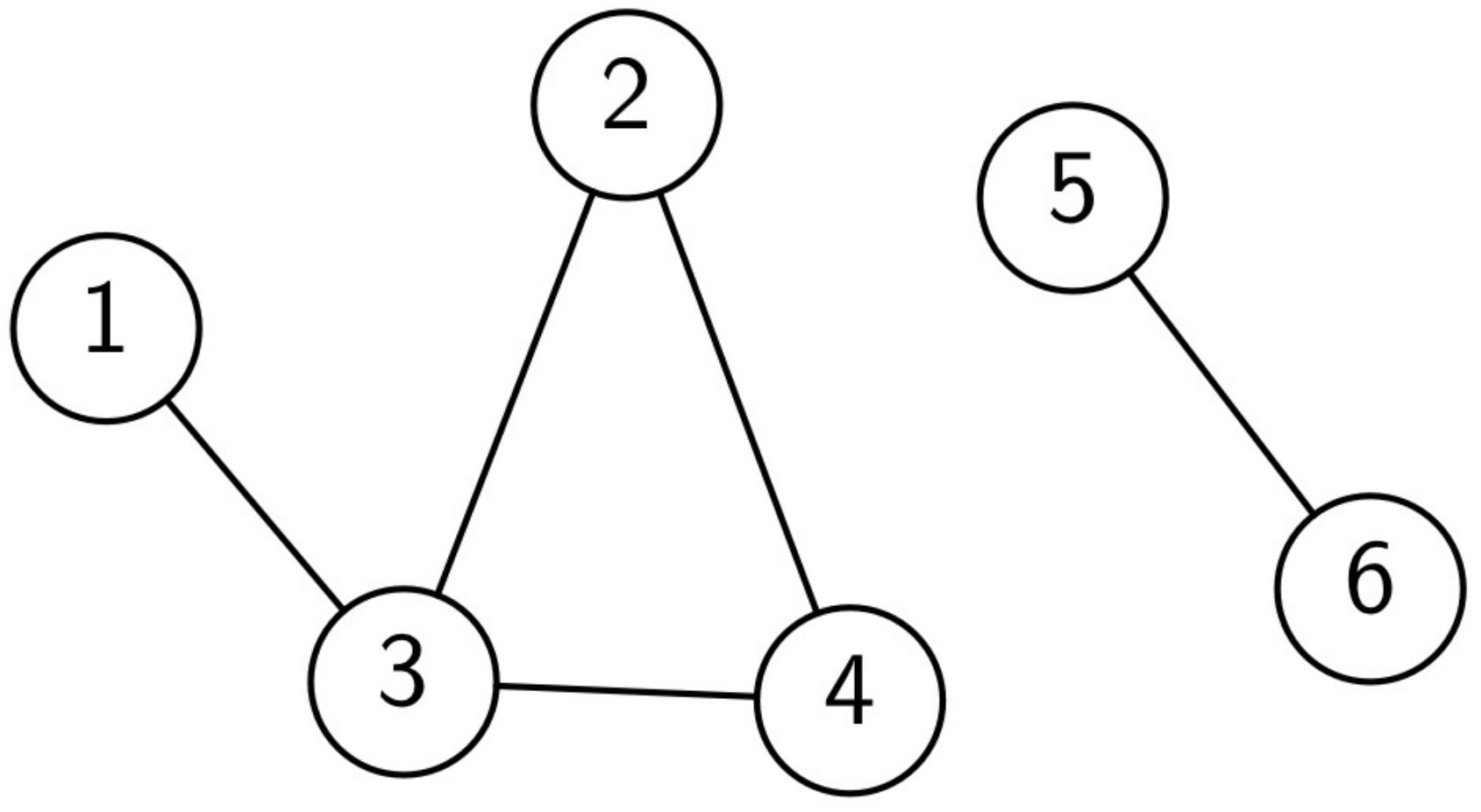
February 14, 2017

Swiss Olympiad in Informatics

Table of Contents

What is a graph

What is a graph



Graph representation

- Set of nodes and edges
- Sometimes edges have a "weight"
- Sometimes edges are directed
- Store for each node each neighbours (adjacent list)
- `vector<vector<int> >` (unweighted)
- `vector<vector<pair<int, int> > >` (weighted)

Read weighted, undirected graph

```
// n nodes, m edges
int n, m;
cin >> n >> m;
vector<vector<pair<int, int> >> graph(n);
for(int i = 0; i < m; i++){
  // edge between a and b, weight c
  int a, b, c;
  cin >> a >> b >> c;
  graph[a].push_back({b, c});
  graph[b].push_back({a, c});
}
```

DFS

DFS

- Traverse graph (i.e. visit all nodes)
- Start at any node
- Follow any edge that we didn't visit yet
- Repeat as long as possible
- If no new node can be reached: go back to previous node
- Similar to how you might search exit in labyrinth

Visulization

Visualization

DFS Implementation

```
vector<vector<int>> graph;
vector<int> visited;
void dfs(int v) {
  // We are at node v and explore neighbors
  for(int e = 0; e < graph[v].size(); e++) {
    int w = graph[v][e];
    // Check whether neighbor w
    // is not yet visited
    if(!visited[w]) {
      visited[w] = true;
      dfs(w); // recursive call
    }
  }
}
```

BFS

BFS

- Traverse graph or find shortest path in unweighted graph
- Start at any node
- Add all undiscovered, connected nodes to queue
- All newly discovered nodes have distance to start of `dist[current] + 1`
- Take next node from queue, repeat

Visulization

Visualization

BFS Implementation

```
void bfs(int start) {
  queue<int> Q;
  Q.push(start);
  while(!Q.empty()) {
    int v = Q.front(); Q.pop();
    for(int e = 0; e < graph[v].size(); e++) {
      int w = graph[v][e];
      if(!visited[w]) {
        visited[w] = true;
        Q.push(w); // replaces recurcive call
      }
    }
  }
}
```

Dijkstra

Dijkstra's shortest path algorithm

- BFS can only calculate the shortest path if all edges have the same length (so called unweighted graph)
- Dijkstra's Algorithm can be used on a weighted graph
- Calculates all shortest paths from a given start point
- Often only path from A to B is of interest

Ants

- Imagine an ant colony
- All ants walk at the same speed
- Thousands of ants start at the anthill (start node)
- Each time there are multiple paths the ants split
- If a path was already taken by an other group it is ignored
- If an ant reaches an undiscovered node it writes the passed time next to the node
- This ant took the shortest path (and therefore knows the shortest distance)
- Otherwise another ant would have discovered the node earlier

Visualization

Visualization

Ants to Code

How do we efficiently simulate the ants?

- Only arrivals of ants at a node are of interest
- Once an ant starts walking we know when it will arrive

Algorithm

1. Place ant at start
2. Take ant that will arrive next (use `priority_queue`)
3. If node was undiscovered yet note the found shortest path, add an ant for each edge to an undiscovered node and calculate the arrival time. Add the ant at the corresponding arrival time to the queue
4. Go to 2 as long as there are still ants walking

Visulization

Visualization

Dijkstra's shortest path algorithm

Time Complexity (easy to implement)

$\mathcal{O}(E \log V)$

Time Complexity (using fancy data structures)

$\mathcal{O}(V \log V + E)$

Implementation

```
int dijkstra(int start, int destination){
  vector<int> mindist(graph.size(), INT_MAX);
  vector<bool> visited(graph.size(), false);
  priority_queue<pair<int, int>> pq;
  pq.push({0, start});
  mindist[start] = 0;
  while(!pq.empty()){
    int active = pq.top().second;
    pq.pop();
    if(visited[active]){
      continue;
    }
    if(active == destination){
      return mindist[destination];
    }
    visited[active] = true;
    for(auto edge : graph[active]){
      if(mindist[active] + edge.second < mindist[edge.first]){
        mindist[edge.first] = mindist[active] + edge.second;
        pq.push({-mindist[edge.first], edge.first});
      }
    }
  }
  return INT_MAX;
}
```

"Beautiful" Implementation by Timon

```cpp
#include <vector>
#include <queue>
using namespace std;

#define OG(T) bool operator>(const T& o)const
struct E{ int t,w; OG(E){ return w>o.w; } };
using VE=vector<E>;
struct V{ VE in; bool v; };
using VV=vector<V>;
using HE=priority_queue<E,VE,greater<E>>;

int dijkstra(VV& g,int s,int t){
  HE q; q.push({s,0});
  while(!q.empty()){
    E c=q.top(); q.pop();
    if(g[c.t].v) continue;
    g[c.t].v=1;
    if(c.t==t) return c.w;
    for(auto&e:g[c.t].in)
      q.push({e.t,c.w+e.w});
  }
  return INT_MAX;
}
```

Implementation using Set by Michal Forišek

```
struct edge { int to, length; };

int dijkstra(const vector< vector<edge> > &graph, int source, int target) {
  vector<int> min_distance( graph.size(), INT_MAX );
  min_distance[ source ] = 0;
  set< pair<int,int> > active_vertices;
  active_vertices.insert( {0,source} );

  while (!active_vertices.empty()) {
    int where = active_vertices.begin()->second;
    if (where == target) return min_distance[where];
    active_vertices.erase( active_vertices.begin() );
    for (auto ed : graph[where])
      if (min_distance[ed.to] > min_distance[where] + ed.length) {
        active_vertices.erase( { min_distance[ed.to], ed.to } );
        min_distance[ed.to] = min_distance[where] + ed.length;
        active_vertices.insert( { min_distance[ed.to], ed.to } );
      }
  }
  return INT_MAX;
}
```

Tasks

Tasks

Training tasks online on grader.soi.ch:

- BFS (2H)
- DFS (2H)
- Dijkstra (2H)
- Signposting (Finals 16)