Graph Algorithms

COMP4128 Programming Challenges

School of Computer Science and Engineering
UNSW Australia

Table of Contents

Graphs

- A graph is a collection of *vertices* and *edges* connecting pairs of vertices.
- Generally, graphs can be thought of as abstract representations of objects and connections between those objects, e.g. intersections and roads, people and friendships, variables and equations.
- Many unexpected problems can be solved with graph techniques.

- Many different types of graphs
  - Directed graphs
  - Acyclic graphs, including trees
  - Weighted graphs
  - Flow graphs
  - Other labels for the vertices and edges

Graph Representations

- You'll usually want to either use an *adjacency list* or an *adjacency matrix* to store your graph.
- An adjacency matrix is just a table (usually implemented as an array) where the jth entry in the ith row represents the edge from i to j, or lack thereof.
- Adjacency lists are a little more complex, but can be faster.

Adjacency List

- The simplest way to implement adjacency lists is to create a vector for every vertex.
- There exist other representations that are slightly faster and more efficient, like implementing your own linked list, but the time complexity of these is the same.
- The simplest implementation with the best time complexity, even if it slower by a constant factor, is usually the best choice.

Adjacency List

- **Implementation for undirected graph**

```
#include <iostream>
#include <vector>

int N = 1e6 + 5; // number of vertices in graph

vector<int> edges[N]; // each vertex has a list of connected vertices
void add(int u, int v) {
  // add from u to v
  edges[u].push_back(v);
  // add from v to u
  edges[v].push_back(u);
}

...

// iterate over edges from u (since C++11)
for (int v : edges[u]) cout << v << '\n';

// iterate over edges from u (before C++11)
vector<int>::iterator it = edges[u].begin();
for (; it != edges[u].end(); ++it) {
  int v = *it;
  cout << v << '\n';
}
```

Table of Contents

Graph Traversals

- There are two main ways to traverse a graph, which differ in the order in which they visit new vertices:
  - Depth-first search (DFS) - visit the first vertex in some vertex's adjacency list, and then recursively DFS on that vertex, then move on;
  - Breadth-first search (BFS) - visit the entire adjacency list of some vertex, then recursively visit every unvisited vertex in the adjacency list of those vertices.
- Both can be implemented in $O(|V| + |E|)$ time.

Depth-First Search

- Depth-first search is a simple idea that can be extended to solve a huge amount of problems.
- Basic idea: for every vertex, recurse on everything it's adjacent to that hasn't already been visited.

- **Implementation**

```
// global arrays are initialised to zero for you
bool seen[N];

void dfs(int u) {
  if (seen[u]) return;
  seen[u] = true;
  for (int v : edges[u]) dfs(v);
}
```

Depth-First Search

- In its simple form, it can already be used to solve several problems - undirected cycle detection, connectivity, flood fill, etc.
- It's very useful, however, to introduce some more terminology to describe the parts of the DFS:
  - *Preorder(u)* - the procedure to execute when u is reached, but before recursing on any of its neighbours;
  - *Inorder(u)* - the procedure to execute after each neighbour of u has finished its recursion;
  - *Postorder(u)* - the procedure to execute after u has finished completely.
- Where in the implementation would each one of these go?
- How would we modify the implementation to return a result found from the graph?

Table of Contents

Directed Graph Cycle Detection

- A simple cycle is a path through a graph consisting of $n \geq 1$ distinct vertices and n distinct edges connecting them (no vertex or edge is visited twice), where the start and end of the path are the same vertex.
- Given a graph, report whether or not this graph contains a cycle. Assume that the graph is connected — if not, we can solve separately on each connected component.
- If the graph is undirected, we can simply run a DFS on the graph, and return true if any vertex marked seen is visited again. (note: every undirected acyclic graph is a tree)
- However, this doesn't work for directed graphs, such as the diamond graph ($1 \rightarrow 2 \rightarrow 3 \leftarrow 4 \leftarrow 1$).

Directed Graph Cycle Detection

- This doesn't work for directed graphs, such as the diamond graph ($1 \to 2 \to 3 \leftarrow 4 \leftarrow 1$). Why not?
- What does the DFS actually do?
- When visiting each vertex in our DFS, we check to see if any of the vertices we're recursing on has been visited before.
- This doesn't work because even if we're visiting some vertex that has been visited before, there's no guarantee that that vertex can visit us.

- However, we can see that the only time that some already-visited vertex that we encounter can reach us is when we're currently within that vertex's recursion stack.
- Now, if we had some easy way to check if we're currently inside the recursion stack of a specific vertex, we'd have a complete algorithm.
- It turns out this is easy to do — just mark each vertex "active" in a table during its preorder step, and unmark it during its postorder step.

- **Implementation**

```
// the vertices that are still marked active when this returns are the
      ones in the cycle we detected
bool has_cycle(int u) {
  if (seen[u]) return false;
  seen[u] = true;
  active[u] = true;
  for (int v : edges[u]) {
    if (active[v] || has_cycle(v)) return true;
  }
  active[u] = false;
  return false;
}
```

Topological Sort

- A topological sort of a graph is an ordering of the vertices that has the property that if some vertex u has a directed edge pointing to another vertex v, then v comes after u in the ordering.
- Clearly, if the graph has a cycle, then there does not exist a valid topological ordering of the graph.

Topological Sort

- Assuming the graph is acyclic (we refer to directed acyclic graphs as DAGs), how do we compute a topological ordering via DFS?
- We can directly use the reverse of the postorder sequence of the graph.
- The postorder sequence of the graph is an ordering of the vertices of the graph in the order that each vertex reaches its postorder procedure.
- A vertex is only added after its children have been visited (and thus added), so the reverse order is a valid topological ordering.

- **Implementation**

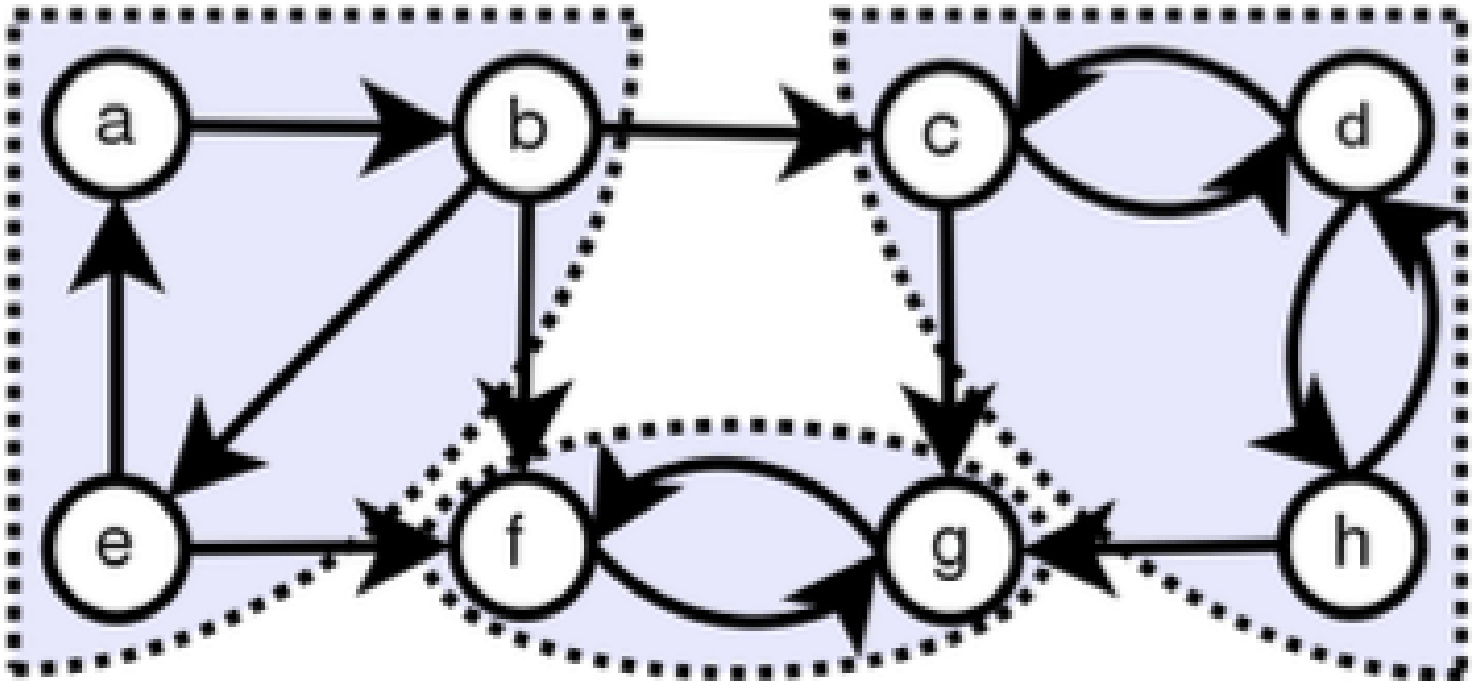
```
// if the edges are in ASCENDING order of node number,
// this produces the lexicographically GREATEST ordering

void dfs(int u, vector<int>& postorder) {
  if (seen[u]) return;
  seen[u] = true;
  for (int v : edges[u]) dfs(v);
  postorder.push_back(u);
}

vector<int> topsort() {
  vector<int> res;
  for (int i = 0; i < n; i++) dfs(i, res);
  reverse(res.begin(), res.end()); // #include <algorithm>
  return res;
}
```

Table of Contents

- A *strongly connected component* (SCC) is a maximal subset of the vertices of a directed graph such that every vertex in the subset can reach every other vertex in that component.
- Condensing every strongly connected component to a single vertex results in a directed acyclic graph (DAG).



Source: Wikipedia

- There are a few linear time algorithms known to compute the strongly connected components of a graph based on DFS.
- Kosaraju's algorithm is simple to implement but hard to understand.
- Another popular choice in these contests is Tarjan's algorithm.

Tarjan's Algorithm

- Do a regular DFS on the graph, but with an explicit stack.
- When an item is pushed onto the stack, mark it as "in-stack", and unmark it as such when it is popped.
- If we want to push a vertex that is already "in-stack", then we've found a strongly connected component.
- Each item on the stack after this vertex can be reached from it, and can also reach that vertex.
- Simply pop everything off the stack including that vertex, and combine it into an SCC.
- The actual algorithm is slightly more complicated because some bookkeeping is required.

- **Implementation**

```
// we will number the vertices in the order we see them in the DFS
int dfs_index[MAX_VERTICES];
// for each vertex, store the smallest number of any vertex we see
// in its DFS subtree
int lowlink[MAX_VERTICES];

// explicit stack
stack<int> s; // #include <stack>
bool in_stack[MAX_VERTICES];

// arbitrarily number the SCCs and remember which one things are in
int scc_counter;
int which_scc[MAX_VERTICES];

void connect(int v) {
  // a static variable doesn't get reset between function calls
  static int i = 1;
  // set the number for this vertex
  // the smallest numbered thing it can see so far is itself
  lowlink[v] = dfs_index[v] = i++;
  s.push(v);
  in_stack[v] = true;

  // continued
```

- **Implementation**

```
  for (auto w : edges[v]) { // for each edge v -> w
    if (!dfs_index[w]) {  // w hasn't been visited yet
      connect(w);
      // if w can see something, v can too
      lowlink[v] = min(lowlink[v], lowlink[w]);
    }
    else if (in_stack[w]) {
      // w is already in the stack, but we can see it
      // this means v and w are in the same SCC
      lowlink[v] = min(lowlink[v], dfs_index[w]);
    }
  }
  // v is the root of an SCC
  if (lowlink[v] == dfs_index[v]) {
    ++scc_counter;
    int w;
    do {
      w = s.top(); s.pop();
      in_stack[w] = false;
      which_scc[w] = scc_counter;
    } while (w != v);
  }
}

// call connect for each vertex once
for (int v = 0; v < n; ++v) if (!dfs_index[v]) connect(v);
```

Table of Contents

Satisfiability

- Satisfiability (SAT) is the problem of determining, given some Boolean formula, if there exists some truthiness assignment of variables which would result in the formula evaluating to true.
- Satisfiability is NP-hard in the general case.

- 2-satisfiability (2SAT) is the problem of determining, given a set of constraints on pairs of Boolean variables, if there exists some truthiness assignment of variables which would result in the conjunction of all the constraints evaluating to true.
- Unlike general satisfiability, 2-satisfiability can be solved in linear time

- The inputs to a 2SAT problem are a set of constraints on Boolean variables with standard Boolean operators.
- In this context, only these make sense:
  - $x_1 \vee x_2$ (disjunction)
  - $\neg x_1$ (negation)

- We can write these in equivalent implicative normal form:
  - $x_1 \vee x_2 \equiv (\neg x_1 \rightarrow x_2) \wedge (\neg x_2 \rightarrow x_1)$
  - $\neg x_1 \equiv (x_1 \rightarrow \neg x_1)$

- These implications now form a directed graph with the Boolean variables (and their negations) as vertices, called the implication graph.
- What does it mean when some variable x can reach some other variable y in this implication graph?
  - If x can reach y in this graph, then $x \rightarrow y$.

- When do we have a valid solution to our 2SAT instance?
- As long as we don't have any contradictions (i.e. $x \to \neg x$ and $\neg x \to x$, we can solve our 2SAT instance.
- In our implication graph, this is exactly the same as checking to see if x and $\neg x$ are in the same strongly connected component!

- We can then easily construct an actual solution to our 2SAT instance after computing the strongly connected components by assigning to each variable whichever truthiness value comes second in the topological ordering of the SCC condensed graph.
- If x comes after $\neg x$ in the topological ordering of the condensed implication graph, then we say x is true. Otherwise, we say it's false.

Table of Contents

- A *tree* is a connected acyclic graph.
- There are several equivalent definitions, all of which are useful:
  - A connected graph with $|V| - 1$ edges
  - An acyclic graph with $|V| - 1$ edges
  - There exists exactly one path between every pair of vertices
  - An acyclic graph where adding any edge results in a cycle
  - A connected graph where removing any edge would disconnect it

- A *spanning tree* for some graph G is a subgraph of G that is a tree, and also connects (spans) all of the vertices of G.
- A *minimum spanning tree* (MST) is a spanning tree with minimum sum of edge weights.
- There are several similar algorithms to solve this problem.

- To construct a minimum spanning tree of some graph G, we maintain a set of spanning forests, initially composed of just the vertices of the graph and no edges, and we keep adding edges until we have a spanning tree.
- Clearly, if we add $|V| - 1$ edges and we avoid constructing any cycles, we'll have a spanning tree.

- How do we decide which edges to add, so that we end up with a minimum spanning tree?
- We can't add any edges to our spanning forest that has its endpoints in the same connected component of our spanning forest, or we'll get a cycle.

- We can restrict ourselves to only the edges that cross components that we haven't connected yet.
- Furthermore, to connect some component C and some other component D, we should always choose the minimum weight edge e that crosses from one to the other; if we instead choose some higher weight edge f, then adding e and deleting the maximum weight edge from the resulting cycle will create a better spanning tree, contradicting minimality.

- The distinction between MST algorithms is in the way that they pick the next components to join together, and how they handle the joining.
- Prim's algorithm only ever connects one large connected component to single disconnected vertices in the spanning forest.
- Kruskal's algorithm connects the two components that contain the next globally minimum edge.

Prim's Algorithm

- Prim's algorithm is typically faster for dense graphs, while Kruskal's algorithm is typically faster for sparse graphs. Both can be implemented to run in $O(|E| \log |V|)$ time.
- Prim's algorithm:
  1. Mark all vertices as not in the tree.
  2. Pick any vertex and add it to the tree. Put all of its outgoing edges in a min-heap, ordered by weight (so that the lowest weight one is at the top).
  3. Remove the top edge $u \to v$ from the min-heap. If v is already in the tree, ignore it and repeat this step.
  4. Add v to the tree. Add all of v's outgoing edges to the min-heap.
  5. Repeat from step 3 until the min-heap is empty.

- **Implementation**

```
#include <vector>
#include <queue>

vector<pair<int, int>> edges[N]; // pairs of (weight, v)
bool in_tree[N];
// use greater as the comparator instead of the default less so the
      priority queue is a min-heap instead of a max-heap
// the vector<int> parameter is the container the queue is stored in, an
      implementation detail you will not need to change
priority_queue<pair<int, int>, vector<int>, greater<pair<int, int>>> pq;

int mst() {
  int total_weight = 0;
  in_tree[0] = true; // (2)
  for (auto edge : edges[0]) pq.emplace(edge.first, edge.second);
  while (!pq.empty()) { // (3)
    auto edge = pq.top(); pq.pop();
    // if this edge goes to somewhere already in the tree, it's useless (
          we've already done something better)
    if (in_tree[edge.second]) continue;
    in_tree[edge.second] = true;
    total_weight += edge.first;
    for (auto edge : edges[edge.second]) pq.emplace(edge.first, edge.
          second); // (4)
  }
  return total_weight;
}
```