DD2458, Problem Solving and Programming Under Pressure

Lecture 5: Graphs

Date: 2009-09-30
Scribe(s): Simon Ragnar, Per Eriksson and Chen Xing
Lecturer: Fredrik NiemelÃď

This lecture goes through some algorithms to solve different graph related problems.

1 Graphs

1.1 Definitions

A **graph** is an ordered pair G = (V, E), where V is a set of **vertices** or **nodes** and E is a set of **edges**. The edge set E consist of *ordered* pairs of vertices in a **directed graph** or **digraph**, and *unordered* pairs of vertices in an **undirected graph**.

Vertices are commonly denoted as $\{v_1, v_2, \ldots, v_n\} \in$ V , and edges as $(v_i, v_j) \in$ E, if there is an edge from vertex v_i to v_j. It is common for edges to hold additional information such as its capacity, weight or distance.

The **degree** of a vertex is the number of edges **incident on** the vertex. Directed graphs usually differ between **in-degree** and **out-degree**; the number of edges **incident to** or **entering** the vertex, and the number of edges **incident from** or **leaving** the vertex. A vertex with degree 0 is called an **isolated** vertex.

Graphs with vertices having high degrees are considered to be **dense** while graphs with low degrees are **sparse**.

A **path** of **length** k in a graph is a sequence $(u_0, u_1, \ldots, u_k) \in$ V such that $(u_{j-1}, u_j) \in$ E for all $j = 1, 2, \ldots, k$. Usually only non-zero length paths are considered.

A **cycle** in a directed graph is a path where $u_0 = u_k$. A **self-loop** is a single-edge cycle. A cycle in an undirected graph is usually a path where $u_0 = u_k$ *and* where each edge in the path is distinct.

Graphs without cycles are called **acyclic graphs** or **trees**.

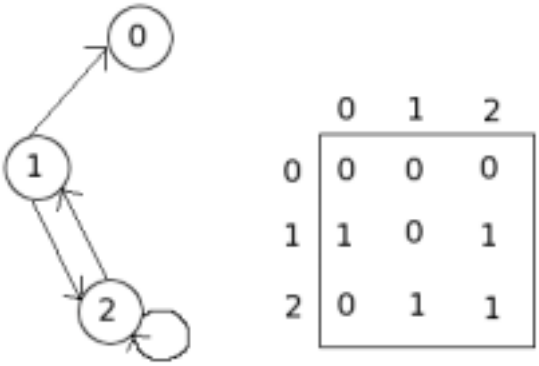
1.2 Representations

When representing a graph in a computer, the two most common structures used are the **adjacency matrix** and the **adjacency list**. It is also not unheard of representing a graph its (then usually sorted) list of edges.

Other representations do exist but are generally only used for a particular algorithm in mind.

1.2.1 Adjacency Matrix

An adjacency matrix represents a graph by a |V| * |V| matrix where element m_{ij} equals 1 if there exist an edge $(v_i, v_j) \in$ E , and 0 otherwise. It is typically used to represent dense graphs.



The picture above illustrates a directed graph with its corresponding adjacency matrix.

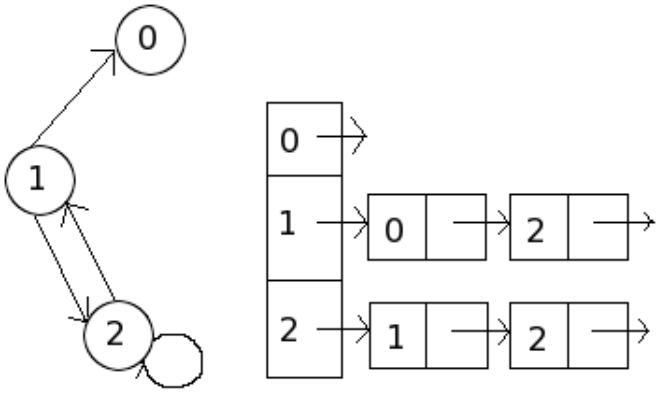
For directed graphs the first index usually indicates the edge's start vertex and the second index indicates the edge's end vertex.

For undirected graphs the upper or lower triangle of the matrix is enough to store all edge information since (v_i, v_j) equals (v_j, v_i).

A **distance matrix** is a special form of adjacency matrix where element D_{ij} instead contains the length of the shortest path from v_i to v_j. See the Floyd-Warshall's shortest path algorithm for an example of where it is commonly being used.

1.2.2 Adjacency List

An adjacency list represents the graph through a list of vertices. Every vertex has in turn a list of edges incidenting from it.



The picture above illustrates a directed graph with its corresponding adjacency list.

Adjacency lists are typically the data structure of choice, but more memory intensive than adjacency matrices for very dense graphs.

Performance characteristics varies greatly depending on the exact list-structures being used. Linked lists, vectors, sets/maps and hash sets/maps are all commonly seen.

Adjacency lists do not have adjacency matrices' limitation of only being able to represent a single edge between a pair of vertices.

1.3 Tips

When working with graph tasks it is often obvious what kind of graph it is and what sort of properties it holds by just reading the task description, but there are exceptions.

It is usually worth the time to closely analyze the graph properties before starting to write the data structure or choosing the algorithm to solve the problem. Is the graph directed or undirected and is it perhaps a good idea to first "reverse" all edges? Can the edge weights be negative, forcing us to use other algorithms? If so, then how will negative cycles influence the algorithm? Is it worth to look at the inverted/complemented graph (i.e. G' = (V, E^C))?

Especially watch out for the "tricky" cases such as the existence of self-loops or multiple edges between pairs of vertices. More often than not, those can be a source of very hard-to-spot bugs.

2 Algorithms

The lecture brought up the most common algorithms for solving the following graph related problems:

- The Shortest Path
- Minimal Spanning Tree
- Topological Sorting
- Strongly Connected Component
- Maximum Flow / Minimal Cut

2.1 The Shortest Path

3 Dijkstra's algorithm

Dijkstra's algorithm is a greedy algorithm for finding the shortest path from a source vertex to all other vertices in a graph.

3.1 Description

It works in this way: All vertexs are assigned two variables;

Definition 3.1 *Distance: the shortes distance from the start vertex to this vertex found so far.*

Definition 3.2 *Previous: the previous vertex on that path.*

Choose the vertex v with the shortest distance, dist[v], from a set of all non-used vertexs Q. For all neighbors of v, $\{u_1, u_2, \ldots, u_n\}$ calculate the new distance. If the new distance is shorter than the previous distance of u_i, then replace the old distance of u_i with the new distance and replace the old previous of u_i with v. Repeat this untill Q is empty.

The algorithm will fail on graphs which has negative edges. The reason for this is that one short cheap path might be chosen while a seemingly longer and more expensive path might include a negative edge, making it the shortest.

This is a greedy algorithm because it selects the vertex with minimum-weight not yet processed to work on. With a simple implementation (i.e. linear search for the vertex with smallest distance in Q) the algorithm has complexity O($|V|^2$) derived from the fact that it searches for the smallest dist[] in Q |Q|/2 times, |Q|=|V| thus the complexity.

With smarter implementations using adjacency lists and a heap/balanced tree based priority queue for Q, the complexity can be reduced to O($|E|log|V|$)

Fibonacci heaps further improves the asymptotic bounds to O($|E|+|V|*log|V|$), but due to the increased complexity of writing the data structure and worse constant term, it's only of practical interest for huge graphs (certainily larger than the ones ever appearing.

Algorithm 1: Dijkstra's algorithm
Input: A start vertex s and a graph G with non-negative edge weights, directed or undirected and if unweighed, all edges should be considered to have a weight of 1. The graph can be unconnected, however, if that is the case and if the algorithm is modified to search for a specific end vertex, then it should also be modified to be able to give the answer: impossible.
Output: The shortest path from s to all other vertexs.

```
DIJKSTRA(G, s, weight[])
(1)   foreach vetex v in G
(2)     dist[v] = infinity
(3)     prev[v] = undefined
(4)   dist[s] = 0
(5)   Q = all vertices in G
(6)   while Q not empty
(7)     u = vertex with minimal distance in Q
(8)     remove u from Q
(9)     if dist[u] = infinity
(10)      break
(11)    foreach neighbor v of u
(12)      if dist[u] + weight[u,v] < dist[v]
(13)        dist[v] = dist[u] + weight[u,v]
(14)        prev[v] = u
(15)  return dist[]
```

4 Bellman Ford

Bellman Fords algorithm for solving the shortest path problem works on the same principles as Dijkstra's algorithm except that it works through all the edges, instead of greedily choosing the one with best distance each time. As a result, Bellman Ford can handle graphs with negative weights because it takes all edges into account and it can also detect negative cucles. The complexity is thereby O($|V||E|$).

This algorithm uses the same definitions as Dijkstra.

Algorithm 2: Bellman Ford
Input: A start vertex s and a graph G, directed or undirected and if unweighed, all edges should be considered to have a weight of 1.
Output: The shortest path from s to all other vertexs.
BELLMAN FORD($G = (V, E)$, s, weight[])
(1) **foreach** vetex v in V
(2) dist[v] = $infinity$
(3) prev[v] = $undefined$
(4) dist[s] = 0
(5) **foreach** u in V
(6) **foreach** (v, w) in E
(7) **if** dist[v] + weight[v,w] < dist[w]
(8) dist[w] = dist[v] + weight[v,w]
(9) prev[w] = v
(10) **foreach** (v, w) in E
(11) **if** dist[v] + weight[v,w] < dist[w]
(12) Negative cycle
(13) **return** dist[]

5 Floyd-Warshall

Floyd warshall is a dynamic algorithm that produces the shortest path between all vertices in the graph. It does this by filling up a path-matrix with distances: The matrix is initialized to contain all edge weights. Then try to make new paths by using all pairs of edges and adding vertice nr 1 to each pair, then nr 2 and so on.

The complexity of this algorithm is O($|V|^3$) because we repeat the operation of trying to find the shortest path between all nodes |V| times producing the complexity.

Algorithm 3: Floyd-Warshall
Input: a 2-dimensional matrix path[][] initialized with edgeweights, number of vertices n
Output: The shortest path from all vertices to all other vertices.
FLOYD-WARSHALL(path[][])
(1) **for** k = 1 **to** $|V|$
(2) **foreach** (i, j) in $\{1, .., n\}^2$
(3) path[i][j] $\leftarrow$ min (path[i][j], path[i][k] + path[k][j])

5.1 Minimal Spanning Tree

Any connected graph without cycles is a tree.

Given an undirected, connected graph G. The minimum spanning tree (MST) of G is the tree that contains all vertices of G and has the lowest possible total edge weight.

The lecture presented two algorithms that finds the MST in a graph, Kruskal and Prim.

5.1.1 Kruskal

Kruskal's algorithm to find a MST in a graph is a greedy algorithm that runs in $O(|E|\log(|E|))$. It works by starting with all the nodes in clusters (one cluster for each vertex at the beginning), then for all edges it takes the cheapest and adds it to the tree unless it creates a cycle. In other words it adds the edge to the tree if the edge's start vertex is in a different cluster then the edge's end vertex. Then it merges the two clusters.

Algorithm 4: Kruskal's Algorithm.
Input: An undirected, connected graph G=(V,E)
Output: The minimum spanning tree of G.
KRUSKAL(G)
(1) **foreach** vertex $v \in$V
(2) Create cluster C(v)
(3) Define a tree T $\leftarrow \emptyset$
(4) Q contains all edges in G
(5) **while** T has fewer than |V|-1 edges
(6) (u, v) $\leftarrow$ Q.removeMinimum()
(7) **if** C(u) $\neq$ C(v)
(8) Add edge (u, v) to T
(9) Merge cluster C(u) and C(v) into one cluster
(10) **return** T

5.1.2 Prim

Prim's Algorithm is another algorithm for finding MST in an connected graph. Usually the graph is weighted aswell, if not every edge weight should be considered one and any spanning tree is the minimum spanning tree. The algorithm has time complexity for most implementations $O(|E|\log(|V|))$ (when a heap is used to pick the next vertex to consider). The algorithm can be improved further by implementing it with Fibonacci heap and bring the time complexity to $O(|E| + |V|\log(|V|))$ which is good when the graph is dense (has many edges).

It works by starting in an arbitrary vertex, u, and adding it to a new set of vertices, V_{new}. Then choose edge (u, v) with minimal weight such that u is in V_{new} but v is not in V_{new}. Now add v to V_{new} and (u, v) to E_{new}

When $V_{new} = V$ the algorithm is done, E_{new} and V_{new} now describes the MST.

(The algorithm bears a close resemblance to Djikstra's algorithm which is used to find the shortest path in a graph between two vertices.)

Algorithm 5: Prim's Algorithm.
Input: A connected graph G=(V,E)
Output: The minimum spanning tree of G.
PRIM(G)
(1) **foreach** vetex v in G
(2) dist[v] $\leftarrow$ $infinity$
(3) prev[v] $\leftarrow$ $undefined$
(4) visited[v] $\leftarrow$ $false$
(5) Choose an arbitrary vertex s to start in.
(6) dist[s] $\leftarrow$ 0
(7) **for** 1 **to** |V|
(8) Find u so that u is minimal and dist[u] = $infinity$ and visited[u] $\leftarrow$ $false$.
(9) visited[u] $\leftarrow$ $true$
(10) **foreach** neighbour v of u
(11) **if** visited[v] = $false$ and weight(u, v) $<$ dist[v]
(12) dist[v] $\leftarrow$ weight(u, v)
(13) prev[v] $\leftarrow$ u

5.2 Topological Sorting

Topological sorting of a directed acyclic graph is to order the vertices in such a way where each edge leaves from a vertex ordered earlier than the vertex being entered to

5.2.1 Khan's Algorithm

Perhaps the simplest and most intuitive algorithm is the greedy one by Khan.

Algorithm 6: Khan's Algorithm.
Input: A directed acylic graph graph G = (V, E).
Output: Topological sorted list of vertices.
KHAN(G)
(1) S $\leftarrow$ empty list
(2) Q $\leftarrow$ G
(3) **while** there is a vertex x $\in$ Q with in-order 0
(4) insert x into S
(5) remove x and edges with x from Q
(6) **if** Q is empty
(7) **return** "G is not acylic"
(8) **return** S

Time complexity is O(|V| + |E|). Memory complexity can be decreased to O(|V|) if Q doesn't store an actual copy of the graph but instead only the orders of the vertices.

5.2.2 Postorder DFS on Reversed Graph

It is easy to see that a postorder DFS traversal of the edge-reversed DAG-trees will result in a topological sorting.

Algorithm 7: TopSortDFS
Input: A directed acylic graph G = (V, E).
Output: Topological sorted list of vertices.

```
TopSortDFS(G)
(1)    S ← empty list
(2)    foreach vertex v in S
(3)        visit(v, S, G)
(4)    return S
```

```
visit(v, S, G)
(1)    if v hasn't been visited
(2)        mark v as visited
(3)        foreach vertex x with an edge from x to v in G
(4)            visit(x, S, G)
(5)            insert v to S
```

Time complexity can easily be made to O(|V| + |E|).

This DFS-algorithm is even useful for problems where there won't be any guarantees that the graph is directed and/or acylic. For example, while the sorted order is then not necessary a "proper" topological sorted order, such a "best-effort" order can still aid one in determining which exact parts of the graph are involves in cycles.

5.3 Strongly Connected Component

A strongly connected component is a subset of the vertices in a graph where each vertex still can reach every other vertex in the subset even if an arbitrary edge is removed.

For those who are interested, there is the simpler to remember Kosaraju's algorithm and the slightly more complex and efficient Tarjan's algorithm.

5.4 Maximum Flow / Minimal Cut

This problem type was only mentioned as a good tool to solve many different problems, but was not looked at more closely since the students should already have experience in working with this kind of problems.