Graphs

2nd Semester

Ivan Canet · Feb 19, 2018 (Mar 10, 2018)

TABLE OF CONTENT

I. MAIN CONCEPTS

1. Undirected	2. Directed
A **Graph** (G) is a set of vertices: $G(V, I)$ • $V=\{a, b, c, d\}$ • $E=\{\{a, b\}, \{b, c\}, ...\}$ We call $n=\|V\|$ the **number of vertices**, and $m=\|E\|$ the **number of edges**.	E is a set of directed edges/arcs: $E=\{(a, b), (b, c), ...\}$
The **degree** of a vertex is the number of incident edges; it's written $d(v)$ and can be described as such: $d(v)=\|\{e \in E / v \in e\}\|$.	We differentiate between the **in-going degree** and the **out-going degree**: In: $d^-(v)=\|\{(u, w) \in E / w=v\}\|$ Out: $d^+(v)=\|\{(u, w) \in E / u=v\}\|$
A **loop** is an edge which links the same vertex twice. **Multiple edges** is a case where two edges link the same two vertices.	**Multiple edges** is a case where two edges *going the same way* link the same two vertices.
Two vertices which share an edge are called **neighbors**.	If an edge goes from u to v, we say that: • u is a **predecessor** of v • v is a **successor** of u
A **path** is a sequence of vertices ($P=u_1 u_2 ... u_k$) such that any vertices are linked pairwise ($\forall u_n, u_{n+1} \in P$ are neighbors). The length of P is $k-1$.	A **directed path** is a path where $\forall u_n, u_{n+1} \in P$, u_{n+1} is a successor of u_n. An **undirected path** is a path where $\forall u_n, u_{n+1} \in P$, u_{n+1} is either a successor or a predecessor of u_n.
A graph is **connected** if a path exists linking any two vertices.	A graph is **strongly connected** if a directed path exists between them. A graph is **weakly connected** if an undirected path exists between them.
An **elementary path** is a path where no vertex appears more than once. An **elementary cycle** is an elementary path that begins and ends with the same vertex.	
A **Eulerian path** is a path where every vertex appears exactly once. For one to exists, every vertex' degree must be even, except for 2 vertices. A **Eulerian cycle** is a cycle where every vertex appears exactly once. For one to exists, every vertex' degree must be even.	

II. Graph classes

1. Regular graph

A regular graph is a graph such that all degrees are the same: $\forall u, v \in V, d(u) = d(v)$.

2. Simple graph

A simple graph has neither loops nor multiple edges.

3. Complete graph

A complete graph is a graph such that every vertex shares an edge with any other.

4. Cycle

A cycle is a graph where the whole graph is a cycle.

5. Tournament

A tournament is a directed graph that is complete in only one direction.

6. Tree

A tree is a graph that has no cycles.

The maximum number of edges for a tree is $n-1$.

7. Bipartite graph

A bipartite graph can be split in two sets of vertices in which no neighbors exist.

8. Complete bipartite graph

A bipartite graph that is complete.

9. Planar

A planar graph can be drawn without crossing edges.

10. Subsets of graphs

In this section, we'll assume a graph G.

10.1. Subgraph

A subgraph G' of G is a graph such that any vertex of G' exists within G. A subgraph may not be connected.

10.2. Clique

A clique G' of G is a subgraph that is complete.

The clique number of G ($\omega(G)$) is the maximum size of a clique of G (the size of the biggest complete subgraph).

10.3. Stable / Independent set

A stable G' of G is a subgraph of G such that no two vertices in G' are neighbors.

The stability number of G ($\alpha(G)$) is the maximum size of a stable of G.

11. Proper coloration

11.1. Definition

A proper coloration is a graph where every vertex has a color (represented as an integer) such that no neighbors have the same color.

If the graph is a planar graph, 4 colors are sufficient. In any other graph, the number of colors needed is in worst case the maximum degree plus one.

The chromatic number of a graph ($\chi(G)$) is the minimum number of colors needed to give a proper coloration of G.

11.2. How to color a graph?

Two main algorithm exist today:

First-fit

Take any non-colored vertex; give it the first available color.

```
Function FirstFit(G:Graph)
      :Map<Vertex,Integer>
  Var m: Map<Vertex,Integer>,
      color: Integer
  Begin
    m ← new Map<Vertex,Integer>()
    For each v in G.getVertices()
      For each n in G.getNeighbors(v)
        color ← 0
        While m.get(n)=color
          color ← color+1
        m.put(v, color)
    Return m
  End
```

Welsh-Powell

Same as First-fit, in decreasing order of degree. This algorithm is faster most of the time.

12. Spanning tree

A spanning tree is a subgraph that has no cycle such that every vertex is interconnected.

12.1. Depth-First tree

Such a tree can be generated easily using the Depth-First algorithm:

```
Mark the current vertex as explored
For every neighbor:
      Recursively do the same
```

Or, in ExAlgo:

```
Action DFS(G:Graph,v:Vertex,
      IO explored:Map<Vertex,Bool>)
  Begin
    explored.put(v,true)
    For each n in G.getNeighbors(v)
      If !explored.get(n)
        DFS(G,n,explored)
  End
```

12.2. Breadth-First tree

A Breadth-First tree is also a spanning tree, but it also has the particularity of containing the shortest path from the first vertex to any other.

```
Mark the current vertex as explored
Add the neighbors to a queue
Loop with the bottom of the queue
```

Or, in ExAlgo:

```
Action BFS(G:Graph,v:Vertex)
  Var explored:Map<Vertex,Bool>
      waiting:Queue<Vertex>
  Begin
    Do
      explored.put(v)
      For each n in G.getNeighbors(v)
        If !explored.get(n)
          waiting.add(n)
      v ← waiting.remove()
    While v != null
  End
```

12.3. Minimum Spanning Tree (MST)

A MST is a spanning tree that has a minimal sum of costs *(see Shortest path, page 5)*.

There are two main algorithms to find the MST of a graph:

Kruskal

```
Sort the vertices
For each vertex
  If it does not create a cycle
    Add that vertex to the MST
```

A possible optimization is to stop when $n-1$ edges have been added *(see Tree, page 3)*.

This algorithm is very efficient, as it's complexity is much lesser than the sort's complexity.

Prim

```
Pick any vertex, add it to the MST
Among the edges that have one
endpoint in the MST
Add the one that has minimal cost
Repeat until n-1 edges
```

III. Shortest path

Weighted graph

A weighted graph is a graph in which every edge has a cost assigned.

1. The shortest path problem

A weighted graph is a graph in which each edge has a cost assigned to it. There exists multiple algorithms to find the shortest path in a weighted graph.

1.1. Bellman-Ford

This algorithm is easy to compute, but inefficient. The aim is to compute the best cost from a vertex u to all other vertices by searching for shortcuts.

```
Initialization:
  cost[all] ← ∞
  cost[u] ← 0
  pred[all] ← itself
Begin:
  For every edge
    If it's better to come to this vertex using this edge
      The cost of this vertex is updated
      The predecessor is updated
Repeat as long as at least 1 shortcut was found
```

Or, in ExAlgo:

```
Function Bellman-Ford(G:Graph,u:Vertex):Map<Vertex,Vertex>
  Var cost:Map<Vertex,int>
      pred:Map<Vertex,Vertex>
      w:Vertex
      done:Boolean

  Begin
    // Initialization
    For each v in G.getVertices()
      cost.put(v, ∞)
      pred.put(v, v)
    cost.put(u, 0)

    // Searching
    Do
      done ← true
      For each (v,w) in G.getEdges()
        If cost[w] + G.cost(v,w) < cost[v]
          done ← false
          cost[v] ← cost[w] + G.cost(v,w)
          pred[v] ← w
    While !done
    Return pred
  End
```

1.2. Dijkstra

This algorithm is very similar in intent; then main difference is that every vertex only has to be explored once. To achieve this, we only look at the vertex with the smallest score – it is then terminated.

```
Initialization:
  cost[all] ← ∞
  cost[u] ← 0
  pred[all] ← itself
Begin:
  For every vertex, in increasing order of cost
    For each un-terminated neighbor
      If it's better to come to the neighbor through the current vertex
        Update the neighbor's cost and predecessor
```

IV. APPENDICES

1. Lexical index