Graphs

2nd Semester

Ivan Canet · Feb 19, 2018 (Feb 19, 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 $n=\vert V\vert$ the **number of vertices**, and $m=\vert E\vert$ 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: $d(v)=\vert\{e\in E / v\in e\}\vert$.	We differentiate between the **in-going degree** and the **out-going degree**: In: $d^{-}(v)=\vert\{(u,w)\in E / w=v\}\vert$ Out: $d^{+}(v)=\vert\{(u,w)\in E / u=v\}\vert$
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.

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.

III. APPENDICES

1. Lexical index