Graph Theory *lite*

What am I looking at?

Daiwei Chen

April 11, 2019

Contents

1 Graph Theory *lite*

When you would like to model problems with "complex" relationships.

Flights Catching flights and changing planes.

Internet Routing The internet is a giant graph. Figuring out the number of hops and which hops to take in order to get to your final destination.

Circuit Boards Making sure connections are efficient.

Basically, things that are connected to each other. A Graph Consists of a set of **vertices** and **nodes** that are *connected* by edges.
Tools: Python - Networkx, c/c++ - nauty. Assumptions:

- No Self Loops
- Edges are Unique

1.1 Paths

Path List of vertices l such that $(l[i], l[i+1]) \epsilon E$.
On each small step of a path, the 2 nodes much have an edge between them.

Simple Path Path without repeated vertices.

Cycle A (mostly) simple path but the 1^{st} and last vertices are the same. Only the beginning and end are repeated.

Path Length The number of edges in a path.

1.2 Graphs

Acyclic Graph A graph without **ANY** cycles.

Connected Graph A graph in which for some a there is an a path from every vertex to every vertex.

Tree Connected acyclic graph.

- $|v| - 1$ edges.

1.3 Subgraphs

Subgraph Only contains verticies and edges of the original graph. Edges must have 2 end-points.

Spanning Tree A subgraph of G that contains all nodes in G and is a tree.

2 Keeping Track of Graphs

These 2 are not the only two styles of graph representations. There are many, many more and more specialized representations for the correct use case.

2.1 Adjacency List

Just a list, for example:

A	->	B, C, D
B	->	A, C, E
C	->	A, B
D	->	A, E
E	->	B, D

2.2 Adjacency Matrix

Shows if one node is connected to another.

	A	B	C	D	E
A	0	1	1	1	0
B	1	0	1	0	1
C	1	1	0	0	0
D	1	0	0	0	1
E	0	1	0	1	0

3 Weighted Graphs

Just a graph, but the edges has been assigned a value. For example, on a switch, different physical connections could have a weight of how much bandwidth each one gets?

4 Minimal Spanning Tree

You want to find the spanning tree where the total weight is the smallest.

5 Prim's Algorithm

A minimal spanning tree algorithm.

```
def prims(V:set, E:set):
    tree_vertex_set = {V[random(len(V))]} # The vertex starts with ANY vert we want to start
    tree_edge_set = {}

    for i=1..len(V)-1:
        # This very much depends on the way you're storing your edges
        find the minimal edge $e_m$ = ($u_m$, $V_m$) such that $u_m \epsilon$ tree_vertex_se
        tree_edge_set.add($e_m)
        tree_vertex_set.add($V_m)

    return tree_edge_set
```

The way that this algorithm works, is that on each run of every vertex, you always find the minimum edge from the starting point, and move on to the next one. Then you do the same until you run out of verticies.

Let's look at the complexity for prim's algorithm. It's fairly hard to say what the possible complexity for prim's algorithm is looking like because the finding the minimal edge depending on your algorithm for that and how the data is represented.
Some examples:

Prim's (Adjacency Matrix) $O(n^2)$

Prim's (List + Binary Heap) $O(e \log n)$

Prim's (List + Fibonacci Heap) $O(e + n \log n)$

6 Kruskal's Algorithm

It is going to put each vertex into its own set. Then, sort the edges on increasing weight. You start from the lowest weight, and if two vertecies are in different sets, then you combine the sets, and accept that edge. Keep combining as long as your edges are not in the same set.

```
def kruskals(V: set, E: set):
    construct |V| sets, each with a unique vertex
    sort edges by increasing weight
    tree_edges = {}
```

```
    for e=(u,v) in sorted edges:
        if u & v are in different sets:
            combine the sets
            tree_edges.add(e)
        if len(tree_edges) == V-1:
            return tree_edges
```

What is the big O on this alogrithm? It is $O(e \log e)$.

7 Shortest Path

Given a weighted directed graph G. Find the shortest path from a source vertex s to a sink vertex t. Notes:

- Negative weights edges are allowed
- Cycles with negative weights are NOT
- We will not have cycles with positive weights

```
def initSSSP(G, s):
    for v in G:
        v.pi = None # the prediciser
        v.d = Inf # best guess on distance

def relax(u, v):
    if v.d > u.d + weight(u,v):
        v.d = u.d + weight(u,v)
        v.pi = u

def bellman_ford(G, s):
    initSSSP(G, s)
    for i in range(x): # x runs ____ times Usually, x = len(V)
        for e = (u,v) in G: # for each edge, relax the edge.
            relax(u,v)
    # Detect a negative cycle
    for e = (u,v) in G:
        if v.d > u.d + weight(u,v): # If you can still relax after relaxing, then panic, You
            return false
    return true
```

u	pi	d
S	null	0
A	S	3
B	A	-1
C	D	-4
D	F	-1
E	S	2
F	E	5
T	B	6

Each time you go through the loop, you'll go at the very least, by one vertex. Note: You go through the table multiple times as to keep every single value within the table updated. In later iterations of the loop, there could be changes in paths without a change in where one vertex comes from vs another. Depending on the graph, it could take up to len(V). Which is why you must check for negative cycles. However, if you've gone through the graph and there're no changes in the table, then you're done relaxing. The order that you check the edges doesn't matter too much. Complexity-wise, this is slower than Dyxstra's algorithm, however, dyxtra does not always work on graphs with negatively weighted graphs.

8 Graph Traversal / Search

- Depth First Search (DFS)
- Breadth First Search (BFS)

8.1 Depth First Search

Go down, then move to the side. Stack used.

```
def dfs(g: Graph, s: vertex):
    visited = set()
    toVisit = stack()
    toVisit.push(s)
    while toVisit is not empty:
        v = toVisit.pop()
        if v not in visited:
            visited.add(v)
            for w in v.neighbors:
                toVisit.push(w)
            print(v)

def rdfs(g, s):
    if s not in visited:
        print(s)
```

```
        visited.add(s)
        for w in s.neighbors:
            rdfs(g, w)
```

8.2 Breadth First Search

Search down, layer by layer. Horizontally. Queue used.

9 Sepuku

How many times can you stab yourself in the stomach to die instantly?

10 Tree Searching using Backtracking

10.1 N-Queens Problem

Place queens on a N x N board so that no queens can be threatened. How many queens can you fill up? For example, you have a 4x4 board, is it possible to place 4 queens? In this case, it is $\binom{16}{4}$, which comes out to be $10 * 13 * 14$.

Now, construct a graph where you place one queen at a time on each level. Do this until you have N queens. Within the graph you've constructed, the solution will never be on level 1. Meaning you're trying to reach the bottom of the graph as fast as computery possible.

Upon each iteration of traversing the graph with depth first, you check if it's possible to place up to 4 queens, if not, you end that branch of the tree and move on to the next depth first search.

```
# The board is what the board looks like, the col is the depth of our board.
def nQueens(board: [][], col:int):
    if col >= len(board):
        return True
    for row in 0..numRows - 1:
        if board[row][col] is not threatened:
            board[row][col] = queen;
            if nQueens(board, col+1):
                return true
            board[row][col] = empty

    return false
```

11 Graph Sorting

Creating topilogical graphs to form the right tree of "prerequisits" (in the graduation problem).

11.1 Recursive Depth First Search

```
def rdfs(g: Graph, s: Vertex):
    if s not in visited:
        visit(s)
        for v in s.neighbors:
            rdfs(g, v)
        put s @ beg of topological sorted list
```

12 Graph Search Heuristics

You don't really need to know the shortest path of every single path. You only need to know the shortest path from point A to point B. Therefore, let's look at the "Best First" Search.

12.1 Best First Search

- Define a function to tell us how "desirable" a node is.
- At each step, expand the node with the best "desirability" until you reach the destination.

12.1.1 Greedy

- Define h(n) – A guess/estimate of the cost from n to the goal. A heuristic function.

h(n) For example, the straight line distance. You just look at the closest towards your destination.

- Make sure you don't visit nodes so you don't get caught in loops.
- This is very quick.

12.1.2 A* (A-star)

The basic plan Figuring out how expensive it is to get to the goal. Don't go down ways that are expensive.

The overall heuristic equation: f(n) = g(n) + h(n)

g(n) Cost from the starting point to the location you're checking.

h(n) Same heuristic of the greedy, checking how far away you are from the destination.

A* avoids going in loops because it counts all the costs, and loops induce more costs

You can accomplish this with a min-heap. By adding in the nodes, and the lowest cost one will rise to the top.

12.2 Admissible Heuristic

- h(n) is admissible if it never overestimates teh cost to reach the goal.
- Straight line distance is admissible; we will never move less distance than the SLD to travel between two points.
  - We don't have teleportation yet.
- Theorem: A* search is optimal if h(n) is an admissible heuristic!

13 8-Puzzle

You have a board of 8 numbers on a 3x3 grid and an opening. How can you move each number until you have all the numbers lined up in the right place.

13.1 Possible Heuristics

- Number of misplaced tiles (h_1)
- Total Manhatten Distance, total minimal distance (h_2)

13.2 Some Statistics

- Given an 8-Puzzle with a solution at depth 14 (fewest number of required moves)
- Number of expanded nodes
  - Iterative Deepening: 3 Million
  - A* w/ h_1: 513
  - A* w/ h_2: 116
- Given an 8-Puzzle with a solution at depth 24 (fewest number of required moves)
- Number of expanded nodes
  - Iterative Deepening: 54 Billion
  - A* w/ h_1: 39,135
  - A* w/ h_2: 1,641

14 Text Compression

- Fixed Lengt Encoding: Like Ascii
- Variable Length Encoding: Like unicode

14.1 Prefix Code

- One character's encoding is not a prefix of another character's

14.2 Huffman Code: Optimal Prefix Code

- Draw a tree with letter frequencies, and the less frequent the letter is, the longer the prefix code is going to be.

15 Exam Stuff

Text Searching and BMH Boyer Mores