Introduction to graphs and their data structures

bfs dfs dijkstra floyd warshall graphs

Original Article:

- Introduction to Graphs and Their Data Structures: Section 1 (by gladius)
- Introduction to Graphs and Their Data Structures: Section 2 (by gladius)
- Introduction to Graphs and Their Data Structures: Section 3 (by gladius)
- Using Hash Function In C++ For User-Defined Classes

Longest Path Problem

The longest path problem is a theoretical CS problem to find a **simple** path of maximum length in a given graph *G*. The longest path problem for a general graph is NP-hard.

This can be shown as the Hamiltonian Path problem reduces to the decision version of this problem and that is known to be NP-complete. To check if a graph has a hamiltonian path, we just need to check if a graph has a longest path of length n - 1 where n is the number of vertices in the graph.

For a directed acyclic graph (DAG) on the other hand this can be solved in linear time. Since then the graph edges can be negated and this reduces to the shortest path problem (since its a DAG there can be no negative cycles). This is also called as the critical path problem.

Algorithm:

```
- For graph (V,E) find the topological ordering using DFS
- For every vertex v starting from the first in the topo sort, calculate the length of the maximum path ending at that point
- return the max of all such paths
```

Breadth First Search

If all edges in a graph are unit length then the shortest distance from a give source to a destination can also be found using BFS. The first time BFS (started from the source) hits the destination, you get the shortest distance.

Also make sure you **don't apply DP** in such questions, as they generally have cycles in them hence DP won't work. Consider an undirected graph `1 <-> 2 <-> 3 <-> 4 <-> 5 <-> 1`. We can see it has a cycle of length 5. Consider a simple DP, that returns the minimum distance to vertex 5.

```
DP(1) calls DP(2), DP(5)
  DP(2) calls DP(3) not DP(1) as we have already visted that
    DP(3) calls DP(4)
      DP(4) calls DP(5)
        DP(5) = 0
      DP(4) = 1
    DP(3) = 2
  DP(2) = 3
DP(1) = min(DP(2), DP(5)) = 0
```

Although the value of DP(1) is correct the value of DP(2) is incorrect.

Remember whenever the sub problems of a DP don't get smaller it doesn't have optimal substructure

Don't use DP in such cases. A child of DP call should never return back to it.

Floyd Warshall

Misconception: That Floyd-Warshall algorithm can return the shortest simple paths in a graph having negative length cycles. But, this is incorrect. Consider we are at the kth iteration and suppose we have found two short paths from `a -> k -> k + 1` and `k + 1 -> k -> b`. Now suppose in the (k+1)th iteration `dist[a][b] > dist[a][k+1] + dist[k+1][b]`. Therefore, we have created a path from a to b with a cycle, `a -> k -> k + 1 -> k -> b` and this possible because we have negative length cycles. This why FW also fails in such circumstances.

BFS

Consider a problem in which we have to find the shortest path from from point A in the grid to point B. And there are some *harmful* regions that cost 1 life each time you step in one and some *deadly* regions that you cannot step on. Rest regions are *normal* and don't cost anything. One idea is to use direct BFS and keep a visited value at each grid point to store the best path to that node uptill now. If we visit that node again with a shorter path we replace the visited value and traverse again. This idea is sub-optimal because it could incur ballooning the complexity.

The method to solve this problem would be **Dijkstra's algo** or **0/1 BFS**.

In 0/1 BFS, for each neighbour with 0 length edge we add it to the front of the queue. And each neighbour with 1 length edge is added to the back of the queue. Essentially the queue here is a deque. This doesn't affect our base assumption that when we visit an vertex all vertex with paths less than the current path length have already been visited. But this method is only applicable in cases where the edge lengths are only 0 and 1, this **doesn't work** for the general case.

Dijkstra's Algorithm

Dijkstra's is a single source shortest path algorithm with a runtime of O(ElogV). To achieve this runtime in practical scenarios, we have to implement the `decreaseKey()` method for the min heap. We do this by maintaining an auxillary array that contains the position of each element in the heap and if we decrease a key in the heap. We just go that position and rebalance the heap by traversing up the heap from that point.

Although the function `decreaseKey()` is called for very few cases so we can just reinsert the element with the new cost in the heap and maintain a distance array. So whenever we get a top element with cost more than that in the distance array we just discard that element.

There are variety of methods to implement the min heap in C++, because by default the priority_queue in C++ is a max heap.

Min heap using custom comparators:

```cpp
struct node{
  int idx;
  int cost;
};

struct compare{
  bool operator()(const node& a, const node& b){
    return a.cost > b.cost;
  }
};
...
...
// then initialise the PQ
priority_queue<node, vector<node>, compare> pq;
// for pair<int,int> we can also use the in-built greater comparator
priority_queue<ii, vector<ii>, greater<ii> > pq;
```

Using sets:

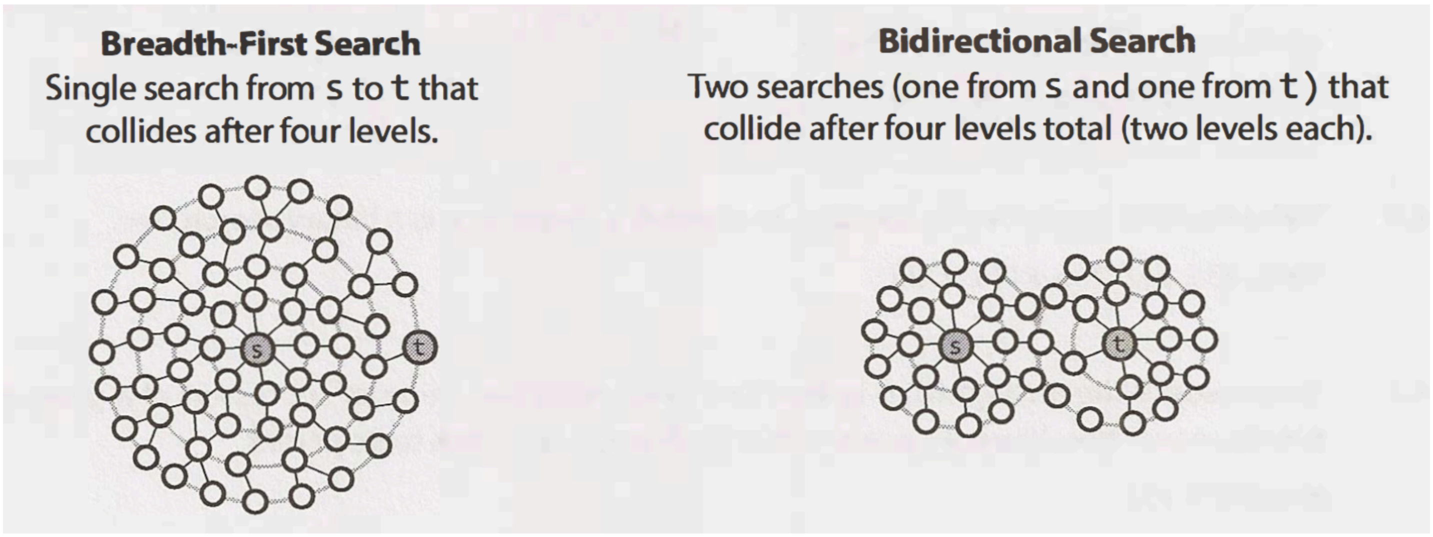
```cpp
struct node{
    int x;
    int y;
    int angle;
    int cost;
};

bool operator<(const node &a, const node &b){
    if(a.cost != b.cost)
        return a.cost < b.cost;
    if(a.x != b.x)
        return a.x < b.x;
    if(a.y != b.y)
        return a.y < b.y;
    return a.angle < b.angle;
}

set<node> pq;
pq.insert(st);
pq.erase(pq.begin()); // make sure to do this (preferably before inserting anything else)
```

while using sets make sure to use all the fields in the structure `node` because in C++ the sets check equality between `a` and `b` by checking if `!(a < b) && !(b < a)`. If all fields are not used it can lead to multiple values being clubbed together based only on the cost and can cause catastrophic errors.

Side tip: `unordered_map` doesn't work when the key that is being indexed is `pair<int,int>` better to use `map` in such cases.
For `unordered_map` one needs to override the in-built hash function for your own structure and also implement the operator `==` for your structure in case of collisions. Otherwise your data would be overriden. The same would be true for `unordered_set` too.

Bidirectional Search

This works by starting simultaneous BFS from both the source and destination. One might think that this doesn't matter but, in practice this is way faster.


Breadth-First Search
Single search from s to t that collides after four levels.
Bidirectional Search
Two searches (one from s and one from t) that collide after four levels total (two levels each).
s
t
s
t

Consider a graph with k neighbors at each vertex. Conventional BFS will find the path from s to t in O(k^d), given d is the distance between s and t. Bidirectional path will get the same path in O(k^(d/2)). This is k^(d/2) (significantly) times faster.

Graph as a matrix

The count of walks of length k from u to v is the [u][v]'th entry in (graph[V][V])^k. We can calculate power of by doing O(Logk) multiplication by using the divide and conquer technique to calculate power. A multiplication between two matrices of size V x V takes O(V^3) time. Therefore overall time complexity of this method is O(V^3Logk).

This is self evident as consider `C = A x B`.
`C[i][j] = sum of all A[i][k] * B[k][j]` which is essentially multiply all 1 length paths from `i -> k` and all 1 length paths from `k -> j`. Hence, this is correct.

Strongly Connected Components (SCC)

Kosaraju's Two Pass Algorithm

Base fact: Start a DFS from any point in a graph and you'll get a union of multiple SCCs. So if you start the DFS in the correct order you'll always get 1 SCC.

Algorithm:

```
1. Let Grev = G with all edges reversed
2. Call a DFS on Grev and store the finishing times of each vertex.
3. Now call a second DFS in the original graph G in reverse order of the finishing time.

Each SCC can be identified by its leader.
```