Pathfinding

JJV

30 août 2024

DFS: depth-first search in Python

```python
def dfs(graph, u, visited):
    visited[u] = True
    for v in graph[u]:
        if not visited[u]:
            dfs(graph, v, visited)
```

In your C++ notebook

```cpp
vector<int> vs[N];
int visited[N];

void dfs(int u) {
        if (visited[u] == 1)
                /*...*/;
        else if (visited[u] == 2)
                /*...*/;
        else {
                visited[u] = 1;
                for (int v : vs[u])
                        dfs(v);
                visited[u] = 2;
        }
}
```

Pathfinding in graphs

```
todo = SomeDataStructure()
Put start in todo
While todo is not empty
    Get node from todo
    For each neighbor of node
        Add neighbor to todo if not visited yet
```

According to the data structure, the complexity and algorithm can be different

- Stack $\rightarrow$ what?
- Queue $\rightarrow$ what?
- Heap $\rightarrow$ what?
- ? $\rightarrow$ graph with edges 0 and 1

Actually, when we mark nodes can have an impact on the complexity too

DFS: depth-first search

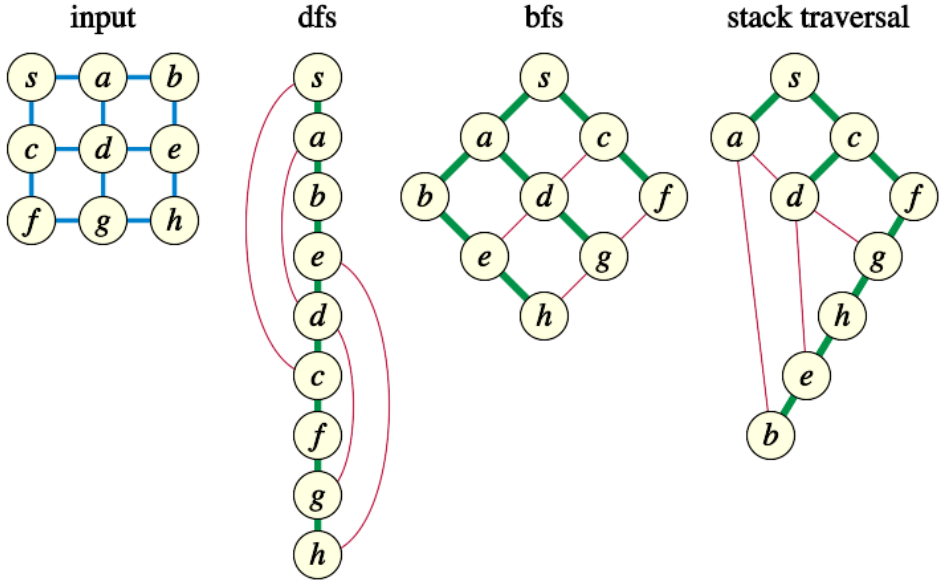
```
def dfs_recursive(graph, node, seen):
    seen[node] = True
    for neighbor in graph[node]:
        if not seen[neighbor]:
            dfs_recursive(graph, neighbor, seen)
```

DFS: depth-first search

```
def dfs_recursive(graph, node, seen):
    seen[node] = True
    for neighbor in graph[node]:
        if not seen[neighbor]:
            dfs_recursive(graph, neighbor, seen)

def dfs_iterative(graph, start, seen):
    seen[start] = True
    to_visit = [start]
    while to_visit:
        node = to_visit.pop()
        for neighbor in graph[node]:
            if not seen[neighbor]:
                seen[neighbor] = True
                to_visit.append(neighbor)
```

DFS

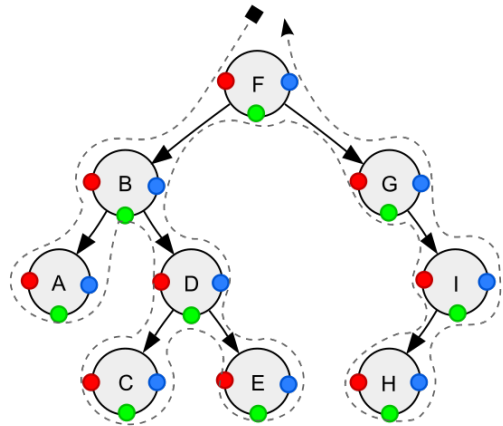

input
dfs
bfs
stack traversal
s a b
c d e
f g h
s a b e d c f g h
s a c b d f e g h
s a c d f g h e b

DFS

```
vector<int> vs[N];
int visited[N];

void dfs(int u) {
        if (visited[u] == 1)
                /*...*/;
        else if (visited[u] == 2)
                /*...*/;
        else {
                visited[u] = 1;
                for (int v : vs[u])
                        dfs(v);
                visited[u] = 2;
        }
}
```

DFS tree traversal



Pre-order: F B A D C E G I H

In-order: A B C D E F G H I (binary tree)

Post-order: A C E D B H I G F

(Wikipedia)

Floyd-Warshall: all source–destination pairs

```
void floydwarshall(vector<vector<int>>& d) {
        // d[u][v] = c(u, v) si (u, v) arc et +oo sinon
        int n = d.size();
        for (int w = 0; w < n; w++)
                for (int u = 0; u < n; u++)
                        for (int v = 0; v < n; v++)
                                d[u][v] = min(d[u][v], d[u][w] + d[w][v]);
}
```

$d_k[u][v]$ = shortest length between u and v using only nodes $< k$

$$d(u, v) = \min_w d(u, w) + d(w, v)$$

Distributivity in semirings (thanks Bellman)

Matrix multiplication	$\sum_k a_{ik} b_{kj}$	$(+, \times)$	(Binet, 1812)
Shortest path	$\min_k d_{ik} + d_{kj}$	$(\min, +)$	(Bellman, 1958)
Most probable path	$\max_k p_{ik} p_{kj}$	$(\max, \times)$	(Viterbi, 1967)

Bellman-Ford

Repeat at most $|V|$ times
  For each edge $(u, v) \in E$
    $d(v) = \min(d(v), d(u) + w_{uv})$

```
const int oo = 1e9;
int n, m;
int dist[N];
vector<tuple<int, int, int>> edge;

bool bellmanford (int u0) {
        fill_n (dist, n, +oo);
        dist[u0] = 0;
        bool stable = false;
        for (int t = 0; t < n && !stable; t++) {
                stable = true;
                for (auto[u, v, c] : edge) if (dist[u] < +oo && dist[u] + c < dist[v]) {
                        dist[v] = dist[u] + c;
                        stable = false;
                }
        }
        return stable;
}
```

$d_k[v]$ = shortest length from source to v using at most k edges

Dijkstra's algorithm

```
const int oo = 1e9;
int n, m;
vector<pair<int, int>> vs[N];
int dist[N];
fill_n(dist, N, oo);

void dijkstra(int u0) {
        priority_queue<pair<int, int>> q;
        q.emplace(-0, u0);
        while (!q.empty()) {
                auto [d, u] = q.top();
                q.pop();
                if (dist[u] == +oo) {
                        dist[u] = -d;
                        for (auto [v, c] : vs[u])
                                q.emplace(d - c, v);
                }
        }
}
```

Summary: shortest paths

problem	complexity	algorithm	implementation
unweighted graph	$O(\|E\|)$	breadth-first search	bfs
grid	$O(\|E\|)$	breadth-first search adapted to the grid graph	dist_grid
{0,1} weighted graph	$O(\|E\|)$	Dijkstra with a deque	graph01
non negative weighted graph	$O(\|E\| \log \|V\|)$	Dijkstra	dijkstra
with lower bound on distance	$O(\|E\| \log \|V\|)$	A*	a_star
arbitrary weighted graph	$O(\|E\| \cdot \|V\|)$	Bellman-Ford	bellman_ford
all source destination pairs	$O(\|V\|^3)$	Floyd-Warshall	floyd_warshall