Graph Traversal Algorithm (in python)

In computer science, **graph traversal** (also known as **graph search**) refers to the process of visiting (checking and/or updating) each vertex in a graph. Such traversals are classified by the order in which the vertices are visited. Tree traversal is a special case of graph traversal.

Redundancy

Unlike tree traversal, graph traversal may require that some vertices be visited more than once, since it is not necessarily known before transitioning to a vertex that it has already been explored. As graphs become more dense, this redundancy becomes more prevalent, causing computation time to increase; as graphs become more sparse, the opposite holds true.

Thus, it is usually necessary to remember which vertices have already been explored by the algorithm, so that vertices are revisited as infrequently as possible (or in the worst case, to prevent the traversal from continuing indefinitely). This may be accomplished by associating each vertex of the graph with a "color" or "visitation" state during the traversal, which is then checked and updated as the algorithm visits each vertex. If the vertex has already been visited, it is ignored and the path is pursued no further; otherwise, the algorithm checks/updates the vertex and continues down its current path.

Several special cases of graphs imply the visitation of other vertices in their structure, and thus do not require that visitation be explicitly recorded during the traversal. An important example of this is a tree: during a traversal it may be assumed that all "ancestor" vertices of the current vertex (and others depending on the algorithm) have already been visited. Both the depth-first and breadth-first graph searches are adaptations of tree-based algorithms, distinguished primarily by the lack of a structurally determined "root" vertex and the addition of a data structure to record the traversal's visitation state.

Graph traversal algorithms

Note. — If each vertex in a graph is to be traversed by a tree-based algorithm (such as DFS or BFS), then the algorithm must be called at least once for each connected component of the graph. This is easily accomplished by iterating through all the vertices of the graph, performing the algorithm on each vertex that is still unvisited when examined.

Depth-first search

A depth-first search (DFS) is an algorithm for traversing a finite graph. DFS visits the child vertices before visiting the sibling vertices; that is, it traverses the depth of any particular path before exploring its breadth. A stack (often the program's call stack via recursion) is generally used when implementing the algorithm.

The algorithm begins with a chosen "root" vertex; it then iteratively transitions from the current vertex to an adjacent, unvisited vertex, until it can no longer find an unexplored vertex to transition to from its current location. The algorithm then backtracks along previously visited vertices, until it finds a vertex connected to yet more uncharted territory. It will then proceed down the new path as it had before, backtracking as it encounters dead-ends, and ending only when the algorithm has backtracked past the original "root" vertex from the very first step.

DFS is the basis for many graph-related algorithms, including topological sorts and planarity testing.

Pseudocode

Input: A graph G and a vertex v of G.

Output: A labeling of the edges in the connected component of v as discovery edges and back edges.

DFS Implementation in Python

Applications:

Depth-First Search (DFS): This algorithm explores as far as possible along each branch before backtracking. It uses a stack to keep track of nodes to be visited. DFS is often implemented using recursion.

Depth-Limited Search (DLS): Similar to DFS, DLS imposes a depth limit to avoid going too deep into the graph. This is useful to prevent infinite loops in cyclic graphs.

Iterative Deepening Depth-First Search (IDDFS): IDDFS is a combination of DFS and DLS. It repeatedly performs DFS with increasing depth limits until the goal is found.

Backtracking: Backtracking is a general algorithmic technique that can be used for graph traversal. It explores all possible paths recursively and abandons a path as soon as it determines that the path cannot lead to the desired solution.

Breadth-first search

A breadth-first search (BFS) is another technique for traversing a finite graph. BFS visits the sibling vertices before visiting the child vertices, and a queue is used in the search process. This algorithm is often used to find the shortest path from one vertex to another.

Pseudocode

Input: A graph G and a vertex v of G.

Output: The closest vertex to v satisfying some conditions, or null if no such vertex exists.

BFS Implementation in python

Applications:

Breadth-First Search (BFS): BFS explores neighbors of a node before moving on to their children. It uses a queue to manage the order of exploration.

Bidirectional BFS: This technique runs two BFS searches simultaneously, one from the start node and the other from the goal node. The two searches meet in the middle, significantly reducing the search space.

Uniform-Cost Search: This algorithm expands nodes with the least accumulated cost first. It's often used in weighted graphs, where edge weights represent costs.

Dijkstra's Algorithm: Specifically designed for finding the shortest paths in weighted graphs, Dijkstra's algorithm maintains a priority queue to explore nodes with the lowest tentative distance from the source.

A Search:* A* combines both heuristic information (estimated cost to goal) and actual cost (from start) to guide the search process efficiently. It uses a priority queue and is widely used in pathfinding and artificial intelligence.

Greedy Best-First Search: This algorithm uses a heuristic to decide which node to expand next. It focuses on reaching the goal quickly based on the heuristic estimate, without necessarily considering the path cost.

Breadth-first search also can be used to solve many problems in graph theory, for example:

finding all vertices within one connected component;

Cheney's algorithm;

finding the shortest path between two vertices;

testing a graph for bipartiteness;

Cuthill–McKee algorithm mesh numbering;

Ford–Fulkerson algorithm for computing the maximum flow in a flow network;

serialization/deserialization of a binary tree vs serialization in sorted order (allows the tree to be re-constructed in an efficient manner);

maze generation algorithms;

flood fill algorithm for marking contiguous regions of a two dimensional image or n-dimensional array;

analysis of networks and relationships.

Graph exploration

The problem of graph exploration can be seen as a variant of graph traversal. It is an online problem, meaning that the information about the graph is only revealed during the runtime of the algorithm. A common model is as follows: given a connected graph G = (V, E) with non-negative edge weights. The algorithm starts at some vertex, and knows all incident outgoing edges and the vertices at the end of these edges—but not more. When a new vertex is visited, then again all incident outgoing edges and the vertices at the end are known. The goal is to visit all n vertices and return to the starting vertex, but the sum of the weights of the tour should be as small as possible. The problem can also be understood as a specific version of the travelling salesman problem, where the salesman has to discover the graph on the go.

For general graphs, the best known algorithms for both undirected and directed graphs is a simple greedy algorithm:

In the undirected case, the greedy tour is at most O(ln n)-times longer than an optimal tour.[1] The best lower bound known for any deterministic online algorithm is 10/3.[2]

Unit weight undirected graphs can be explored with a competitive ration of $2 - \varepsilon$,[3] which is already a tight bound on Tadpole graphs.[4]

In the directed case, the greedy tour is at most $(n - 1)$-times longer than an optimal tour. This matches the lower bound of $n - 1$.[5] An analogous competitive lower bound of $\Omega(n)$ also holds for randomized algorithms that know the coordinates of each node in a geometric embedding. If instead of visiting all nodes just a single "treasure" node has to be found, the competitive bounds are $\Theta(n2)$ on unit weight directed graphs, for both deterministic and randomized algorithms.

Universal traversal sequences

A universal traversal sequence is a sequence of instructions comprising a graph traversal for any regular graph with a set number of vertices and for any starting vertex. A probabilistic proof was used by Aleliunas et al. to show that there exists a universal traversal sequence with number of instructions proportional to O(n5) for any regular graph with n vertices.[6] The steps specified in the sequence are relative to the current node, not absolute. For example, if the current node is vj, and vj has d neighbors, then the traversal sequence will specify the next node to visit, vj+1, as the ith neighbor of vj,

where $1 \leq i \leq d$.