Chapter 6 : Breadth-First Search and Depth-First Search

Omar Saadi
omar.saadi@um6p.ma

UM6P - College of Computing

1st year of the engineering degree of UM6P-CS
Academic year 2024-2025

Outline

Outline

Introduction

Given a graph G, there are several problems one wants to solve related to the connectivity of the graph.

Introduction

Given a graph G, there are several problems one wants to solve related to the connectivity of the graph. Those include the following :

- Given a pair of vertices (u, v), is there a path in G from u to v ?

Introduction

Given a graph G, there are several problems one wants to solve related to the connectivity of the graph. Those include the following :

- Given a pair of vertices (u, v), is there a path in G from u to v ?
- Given a pair of vertices (u, v), what is the distance $d(u, v)$ and a shortest path from u to v ?

Introduction

Given a graph G, there are several problems one wants to solve related to the connectivity of the graph. Those include the following :

- Given a pair of vertices (u, v), is there a path in G from u to v ?
- Given a pair of vertices (u, v), what is the distance $d(u, v)$ and a shortest path from u to v ?
- Given a vertex s, find $d(s, v)$ for all $v \in V(G)$ and a **shortest path tree*** containing a shortest path from s to every $v \in V$.

* : A shortest path tree is a tree of G where the unique path from s to any vertex v inside this tree is a shortest path from s to v in G.

Shortest Paths Tree

Question

How to return a shortest path from the source vertex s to every vertex v in the graph ?

Shortest Paths Tree

Question

How to return a shortest path from the source vertex s to every vertex v in the graph ?

Problem

Many paths could have length $\Omega(|V|)$, so returning every path could require $\Omega(|V|^2)$ time.

Shortest Paths Tree

Question

How to return a shortest path from the source vertex s to every vertex v in the graph ?

Problem

Many paths could have length $\Omega(|V|)$, so returning every path could require $\Omega(|V|^2)$ time.

Idea

- For every $v \in V$, store its parent $P(v)$ in a shortest path from s to v.
- This set of parents has an $\Omega(|V|)$ size and comprises a shortest paths tree.
- It provides a reversed shortest paths back to s from every vertex v reachable from s

Breadth-First Search (BFS)

We want to : Store $d(s, v)$ and $P(v)$ in Set data structures mapping vertices v to distance and parent respectively.
(If no path from s to v, do not store v in P and set $d(s, v)$ to ∞).

Breadth-First Search (BFS)

We want to : Store $d(s, v)$ and $P(v)$ in Set data structures mapping vertices v to distance and parent respectively.
(If no path from s to v, do not store v in P and set $d(s, v)$ to ∞).

Idea : Explore graph nodes in increasing order of distance from s.

Goal : Compute level sets $L_i = \{v \mid v \in V \text{ and } d(s, v) = i\}$ (i.e., all vertices at distance i from s).

Breadth-First Search (BFS)

We want to : Store $d(s, v)$ and $P(v)$ in Set data structures mapping vertices v to distance and parent respectively.
(If no path from s to v, do not store v in P and set $d(s, v)$ to ∞).

Idea : Explore graph nodes in increasing order of distance from s.

Goal : Compute level sets $L_i = \{v \mid v \in V \text{ and } d(s, v) = i\}$ (i.e., all vertices at distance i from s).

Claims :

- Every vertex $v \in L_i$ must be adjacent to some vertex $u \in L_{i-1}$ (i.e., $v \in \text{Adj}(u)$).
- No vertex that is in L_j for some $j < i$, appears in L_i.

Breadth-First Search (BFS)

We want to : Store $d(s, v)$ and $P(v)$ in Set data structures mapping vertices v to distance and parent respectively.
(If no path from s to v, do not store v in P and set $d(s, v)$ to ∞).

Idea : Explore graph nodes in increasing order of distance from s.

Goal : Compute level sets $L_i = \{v \mid v \in V \text{ and } d(s, v) = i\}$ (i.e., all vertices at distance i from s).

Claims :

- Every vertex $v \in L_i$ must be adjacent to some vertex $u \in L_{i-1}$ (i.e., $v \in \text{Adj}(u)$).
- No vertex that is in L_j for some $j < i$, appears in L_i.

Invariant : $d(s, v)$ and $P(v)$ have been computed correctly for all v in any L_j for $j < i$.

BFS Algorithm

- Base case (i = 0) : $L_0 = \{s\}$, $d(s, s) = 0$, $P(s) = \text{None}$
- Inductive Step : To compute L_i :
  For every vertex $u \in L_{i-1}$:
  For every vertex $v \in \text{Adj}(u)$ that does not appear in any L_j for $j < i$, do :
  Add v to L_i, set $d(s, v) = i$, and set $P(v) = u$.
- Repeatedly compute L_i from L_j for $j < i$ for increasing i until L_i is the empty set.
- Set $d(s, v) = \infty$ for any $v \in V$ for which $d(s, v)$ was not set.

Remarks

- The principle of the above Breadth-First Search algorithm is :
  - Start with the source vertex s
  - Visit his neighbours
  - Then visit the neighbours of its neighbours (that were not yet explored) and so on $\cdots$ until visiting all reachable vertices from s.

Remarks

- The principle of the above Breadth-First Search algorithm is :
  - Start with the source vertex s
  - Visit his neighbours
  - Then visit the neighbours of its neighbours (that were not yet explored) and so on $\cdots$ until visiting all reachable vertices from s.
- Recall that the largest distance from a fixed vertex u to another vertex is the eccentricity $\epsilon(u)$. Hence BFS started at u allows to compute this eccentricity.
- So we can compute the diameter of a graph by running Breadth-First Search from each vertex.

Remarks

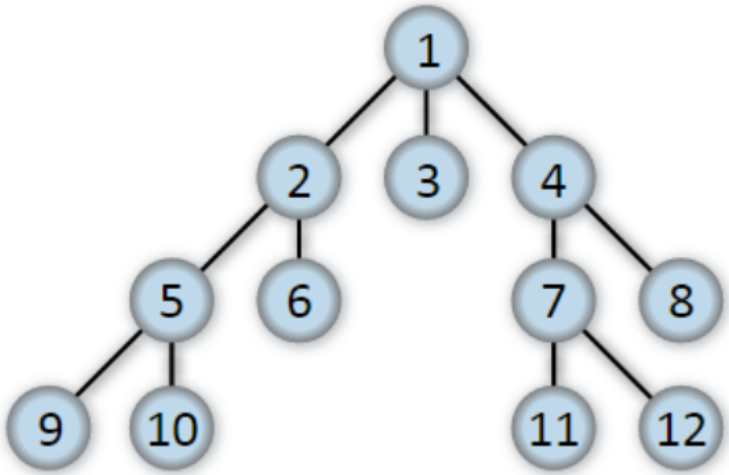
- The principle of the above Breadth-First Search algorithm is :
  - Start with the source vertex s
  - Visit his neighbours
  - Then visit the neighbours of its neighbours (that were not yet explored) and so on $\cdots$ until visiting all reachable vertices from s.
- Recall that the largest distance from a fixed vertex u to another vertex is the eccentricity $\epsilon(u)$. Hence BFS started at u allows to compute this eccentricity.
- So we can compute the diameter of a graph by running Breadth-First Search from each vertex.
- BFS can be applied in its same above format for **directed graphs**. In this case, the neighbors of u are taken in the directed sens.

Remarks

- The principle of the above Breadth-First Search algorithm is :
  - Start with the source vertex s
  - Visit his neighbours
  - Then visit the neighbours of its neighbours (that were not yet explored) and so on $\cdots$ until visiting all reachable vertices from s.
- Recall that the largest distance from a fixed vertex u to another vertex is the eccentricity $\epsilon(u)$. Hence BFS started at u allows to compute this eccentricity.
- So we can compute the diameter of a graph by running Breadth-First Search from each vertex.
- BFS can be applied in its same above format for **directed graphs**. In this case, the neighbors of u are taken in the directed sens.
- BFS can also be applied to the particular case of trees to do a tree traversal.

Example

Performing BFS search on the following tree, starting from the vertex 1, leads to the next traversal of the tree :



Proposition

Breadth-first search correctly computes all $d(s, v)$ and $P(v)$ for every $v \in V$.

Proof : Straightforward by an induction over i.

Proposition

Breadth-first search correctly computes all $d(s, v)$ and $P(v)$ for every $v \in V$.

Proof : Straightforward by an induction over i.

Proposition

Breadth-first search runs in $O(|V| + |E|)$ time (i.e. linear time).

Proof : (Next slide)

Proof

- Algorithm adds each vertex u to at most one level L_i and spends $O(1)$ time for each $v \in \text{Adj}(u)$.

Proof

- Algorithm adds each vertex u to at most one level L_i and spends $O(1)$ time for each $v \in \text{Adj}(u)$.
- Time complexity upper bounded by $O(1) \times \sum_{u \in V} \deg(u) = O(|E|)$
- Spend $O(|V|)$ at end to assign $d(s, v)$ for vertices $v \in V$ not reachable from s

  $\Rightarrow$ Breadth-first search runs in linear time $O(|V| + |E|)$

Application : checking strongly connectivity

Problem

Given a directed graph, check if it is strongly connected or not.

Application : checking strongly connectivity

Problem

Given a directed graph, check if it is strongly connected or not.

Solution 1 (a naïve one)

Perform BFS starting from every vertex in the graph. If each BFS call visits every other vertex in the graph, then the graph is strongly connected.
Complexity : We are performing BFS $|V|$ times, and each call of BFS need $O(|V| + |E|)$, so the total complexity is $O(|V|(|V| + |E|))$.

Solution 2

- Choose a vertex u, and perform BFS starting from u.
- Reverse the direction of all edges in the digraph G, and perform again BFS starting from the same vertex u.

Solution 2

- Choose a vertex u, and perform BFS starting from u.
- Reverse the direction of all edges in the digraph G, and perform again BFS starting from the same vertex u.

Proposition

The graph is strongly connected if and only if both calls of BFS visit every vertex in the graph.

Proof : There is a path from u to every other vertex v, and also a path from every other vertex v to u.

Solution 2

- Choose a vertex u, and perform BFS starting from u.
- Reverse the direction of all edges in the digraph G, and perform again BFS starting from the same vertex u.

Proposition

The graph is strongly connected if and only if both calls of BFS visit every vertex in the graph.

Proof : There is a path from u to every other vertex v, and also a path from every other vertex v to u.

Complexity : We are performing BFS twice, so the total complexity is $O(|V| + |E|)$.

Remark : this problem can be solved also by DFS (described below) instead of BFS.

Outline

Depth-First Search (DFS)

Goal :

- Search a graph from a vertex s, like BFS.
- Solve Single Source Reachability : i.e. Given a vertex s (source), identify all the vertices u reachable from s and provide a path from s to u (not necessarily a shortest path).
- Return parent tree of parent pointers back to s (like BFS, but not necessarily shortest paths).

Depth-First Search (DFS)

Goal :

- Search a graph from a vertex s, like BFS.
- Solve Single Source Reachability : i.e. Given a vertex s (source), identify all the vertices u reachable from s and provide a path from s to u (not necessarily a shortest path).
- Return parent tree of parent pointers back to s (like BFS, but not necessarily shortest paths).

Idea :

Visit outgoing adjacencies recursively, but never revisit a vertex.
Meaning that : follow any path until you get stuck, backtrack until finding an unexplored path to explore.

DFS algorithm :

- $P(s) = \text{NONE}$, then run visit(s), where the function "visit" is defined recursively as follows :

  visit(u) :

  - for every $v \in \text{Adj}(u)$ that does not appear in P : Set $P(v) = u$ and recursively call visit(v).
  - DFS finishes visiting vertex u (for use later !).

DFS algorithm :

- $P(s) = \text{NONE}$, then run visit(s), where the function "visit" is defined recursively as follows :

  visit(u) :

  - for every $v \in \text{Adj}(u)$ that does not appear in P : Set $P(v) = u$ and recursively call visit(v).
  - DFS finishes visiting vertex u (for use later !).

Remarks :

- The principle of the above Depth-First Search algorithm is :
  - Start with the source vertex s
  - Visit one of his neighbours
  - Then visit one neighbour of this neighbour (that was not yet explored) and so on $\cdots$ until visiting all reachable vertices from s (when no more neighbors exist backtrack until finding an unexplored path to explore).

DFS algorithm :

- $P(s) = \text{NONE}$, then run visit(s), where the function "visit" is defined recursively as follows :

  visit(u) :

  - for every $v \in \text{Adj}(u)$ that does not appear in P : Set $P(v) = u$ and recursively call visit(v).
  - DFS finishes visiting vertex u (for use later !).

Remarks :

- The principle of the above Depth-First Search algorithm is :
  - Start with the source vertex s
  - Visit one of his neighbours
  - Then visit one neighbour of this neighbour (that was not yet explored) and so on $\cdots$ until visiting all reachable vertices from s (when no more neighbors exist backtrack until finding an unexplored path to explore).
- DFS can also be applied to **directed graphs**. Here the neighbors of u are taken in the directed sens.
- DFS can also be applied to the particular case of trees to do a tree traversal.

Example

Performing DFS search on the same tree as the example above, starting from the vertex 1, leads to the next traversal of the tree (left), to compare to the previous BFS traversal (right) :

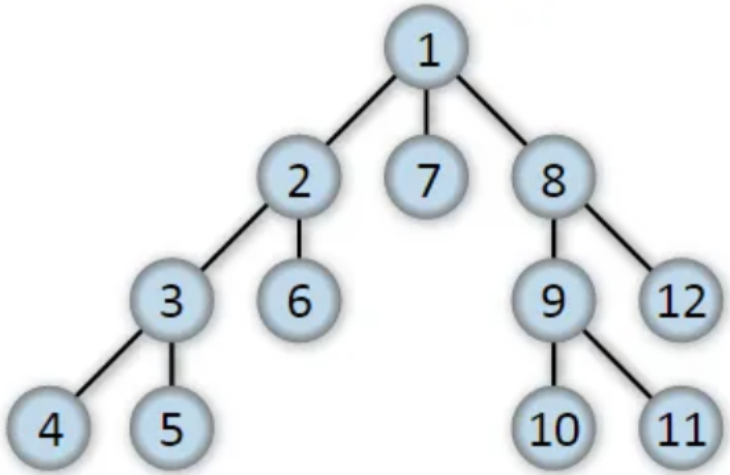


Figure – DFS traversal

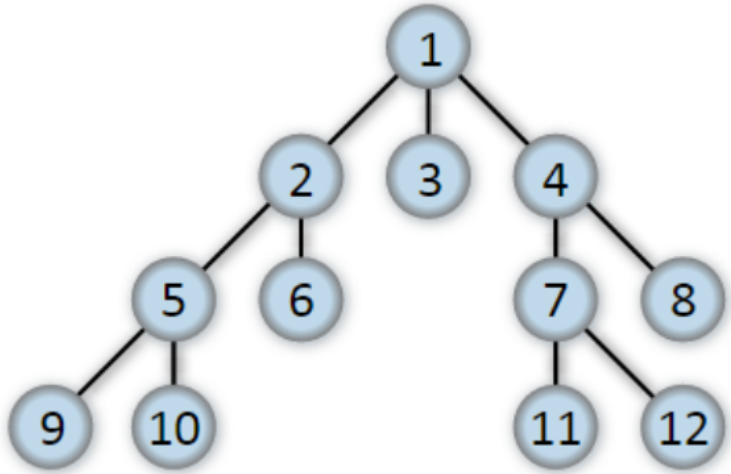


Figure – BFS traversal

Proposition

DFS visits v and correctly sets $P(v)$ for every vertex v reachable from s.

Proof

- Induction on k, for the result on only vertices within distance k from the source vertex s.

Proposition

DFS visits v and correctly sets $P(v)$ for every vertex v reachable from s.

Proof

- Induction on k, for the result on only vertices within distance k from the source vertex s.
- Base case $(k = 0)$: $P(s)$ is set correctly for s and s is visited.

Proposition

DFS visits v and correctly sets $P(v)$ for every vertex v reachable from s.

Proof

- Induction on k, for the result on only vertices within distance k from the source vertex s.
- Base case $(k = 0)$: $P(s)$ is set correctly for s and s is visited.
- Inductive step : Consider vertex v with $d(s, v) = k + 1$.

  Consider vertex u, the second to last vertex on some shortest path from s to v.

Proposition

DFS visits v and correctly sets $P(v)$ for every vertex v reachable from s.

Proof

- Induction on k, for the result on only vertices within distance k from the source vertex s.
- Base case $(k = 0)$: $P(s)$ is set correctly for s and s is visited.
- Inductive step : Consider vertex v with $d(s, v) = k + 1$.

  Consider vertex u, the second to last vertex on some shortest path from s to v.

  By induction, since $d(s, u) = k$, DFS visits u and sets $P(u)$ correctly.

  While visiting u, DFS considers $v \in \text{Adj}(u)$.

Proposition

DFS visits v and correctly sets $P(v)$ for every vertex v reachable from s.

Proof

- Induction on k, for the result on only vertices within distance k from the source vertex s.
- Base case $(k = 0)$: $P(s)$ is set correctly for s and s is visited.
- Inductive step : Consider vertex v with $d(s, v) = k + 1$.

  Consider vertex u, the second to last vertex on some shortest path from s to v.

  By induction, since $d(s, u) = k$, DFS visits u and sets $P(u)$ correctly.

  While visiting u, DFS considers $v \in \text{Adj}(u)$.

  Either v is in P, so has already been visited, or v will be visited while visiting u.

  In either case, v will be visited by DFS and added correctly to P.

Running time of DFS

- Algorithm visits each vertex u at most once and spends $O(1)$ time for each $v \in \text{Adj}(u)$.

Running time of DFS

- Algorithm visits each vertex u at most once and spends $O(1)$ time for each $v \in \text{Adj}(u)$.
- Total complexity upper bounded by $O(1) \times \sum_u \deg(u) = O(|E|)$.

Running time of DFS

- Algorithm visits each vertex u at most once and spends $O(1)$ time for each $v \in \text{Adj}(u)$.
- Total complexity upper bounded by $O(1) \times \sum_u \deg(u) = O(|E|)$.
- Unlike BFS, we don't return a distance for each vertex, so DFS runs in $O(|E|)$ time.

Running time of DFS

- Algorithm visits each vertex u at most once and spends $O(1)$ time for each $v \in \text{Adj}(u)$.
- Total complexity upper bounded by $O(1) \times \sum_u \deg(u) = O(|E|)$.
- Unlike BFS, we don't return a distance for each vertex, so DFS runs in $O(|E|)$ time.

Remark : If we want to identify also the vertices that are not reachable from s, i.e. that does not appear in P, the complexity becomes $O(|V| + |E|)$, like BFS.

Full-BFS and Full-DFS

- Suppose we want to explore entire graph, not just vertices reachable from one vertex : A simple BFS or DFS from one vertex is not sufficient especially in the case of non connected graph (non strongly connected digraph).

Full-BFS and Full-DFS

- Suppose we want to explore entire graph, not just vertices reachable from one vertex : A simple BFS or DFS from one vertex is not sufficient especially in the case of non connected graph (non strongly connected digraph).
- **Idea :** Repeat the graph search algorithm (BFS or DFS) on any unvisited vertex. This unvisited vertex will allow to explore another connected component (or strongly connected component in a digraph).
- We call this algorithm a Full-BFS or a Full-DFS. Notice that they are providing parent forests instead of parent trees.

Full-BFS and Full-DFS

- Suppose we want to explore entire graph, not just vertices reachable from one vertex : A simple BFS or DFS from one vertex is not sufficient especially in the case of non connected graph (non strongly connected digraph).
- **Idea :** Repeat the graph search algorithm (BFS or DFS) on any unvisited vertex. This unvisited vertex will allow to explore another connected component (or strongly connected component in a digraph).
- We call this algorithm a Full-BFS or a Full-DFS. Notice that they are providing parent forests instead of parent trees.

Remark : A Full-BFS (or a Full-DFS) allows to identify the connected components (or the strongly connected components in a digraph).

Complexity : Since we are going through all the vertices, the time complexity of both algorithms Full-BFS and Full-DFS is $O(|V| + |E|)$.

Application : Topological Sort

Definition

A **Directed Acyclic Graph** (DAG) is a directed graph that contains no directed cycles.
A **Topological Order** of a graph $G = (V, E)$ is an ordering f on the vertices such that : every edge $(u, v) \in E$ satisfies $f(u) < f(v)$.

Recall (ES4 - ex7) : A directed graph admits a topological ordering if and only if it is a DAG.

Application : Topological Sort

Definition

A **Directed Acyclic Graph** (DAG) is a directed graph that contains no directed cycles.
A **Topological Order** of a graph $G = (V, E)$ is an ordering f on the vertices such that : every edge $(u, v) \in E$ satisfies $f(u) < f(v)$.

Recall (ES4 - ex7) : A directed graph admits a topological ordering if and only if it is a DAG.

Question : How to find a topological order ?

Application : Topological Sort

Definition

A **Directed Acyclic Graph** (DAG) is a directed graph that contains no directed cycles.
A **Topological Order** of a graph $G = (V, E)$ is an ordering f on the vertices such that : every edge $(u, v) \in E$ satisfies $f(u) < f(v)$.

Recall (ES4 - ex7) : A directed graph admits a topological ordering if and only if it is a DAG.

Question : How to find a topological order ?

Definition

A **Finishing Order** is the order in which a Full-DFS finishes visiting each vertex in G.

Application : Topological Sort

Proposition

A digraph $G = (V, E)$ is a DAG if and only if the reverse of a finishing order of a Full-DFS is a topological order.

Proof

$\Leftarrow$) This implication is trivial.

$\Rightarrow$) Need to prove, for every edge $(u, v) \in E$ that u is ordered before v, i.e. that the visit to v finishes before the visit to u finishes. Two cases :

- If u is visited before v :
  - Before visit to u finishes, will visit v (via (u, v) or otherwise).
  - Thus the visit to v finishes before finishing the visit to u.
- If v is visited before u :
  - u can't be reached from v since graph is acyclic.
  - Thus the visit to v finishes before visiting u.

Application : Cycle detection

- Full-DFS will find a topological order iff the graph $G = (V, E)$ is acyclic.
- If G contains a cycle then reverse finishing order for Full-DFS (called f) is not a topological order. Meaning that there exists $(u, v) \in E$ such that Full-DFS finishes visiting u before it finishes visiting v (i.e. $f(v) < f(u)$).

Proposition

Let G be a diagraph that contains a cycle, and $(u, v) \in E$ such that Full-DFS finishes visiting u before it finishes visiting v, then this Full-DFS contains a path from v to u constructed by considering the reversed path back to s from vertex v. Adding the edge (u, v) to this path we obtain a cycle.

Application : Cycle detection

Proof (of Prop.)

Necessarily v is visited before u (because otherwise using the edge (u, v), v will be visited and finished before u). Since Full-DFS finishes visiting u before v this means that during the visit of v, Full-DFS starts and finishes the visit of u, and this means that v is an ancestor of u, giving a path from v to u.

Complexity : We need to run Full-DFS needing $O(|V| + |E|)$. Then, we need to find an edge (u, v) such that $f(v) < f(u)$ and this will take $O(|E|)$ to go through all the edges. Finally we need to go through the reverse path from v to u, and this will need at most $O(|V|)$. Finally, the total complexity is

$$O(|V| + |E|).$$