CS200: Graphs

Graph Traversal

Connected Components

An undirected graph is called connected if there is a path between every pair of vertices of the graph.

A connected component of a graph G is a connected subgraph of G that is not a proper subgraph of another connected subgraph of G.

\[ G = \{a, b, c, d, e, f, g\}, E \]

\[ G_1 = \{a, b, c\}, E_1 \]

\[ G_2 = \{d, e, f, g\}, E_2 \]

Graph Traversal

Depth First Search (DFS)

\[ \text{dfs} \text{ (in } v : \text{Vertex}) \]

\[ \text{visit } v \]

\[ \text{for (each unvisited vertex } u \text{ adjacent to } v) \]

\[ \text{dfs}(u) \]

- Need to track visited nodes
- Order of visiting nodes is not completely specified
- Is there a difference between directed undirected graphs?

Graph Terminology

A subgraph of a graph \( G = (V,E) \) is a graph \( (V',E') \) such that \( V' \) is a subset of \( V \) and \( E' \) is a subset of \( E \)

Iterative DFS

\[ \text{dfs} \text{ (in } v : \text{Vertex}) \]

\[ s \text{ – stack for keeping track of active vertices} \]

\[ s \text{.push}(v) \]

\[ \text{mark } v \text{ as visited} \]

\[ \text{while}(s \text{.isEmpty}) \}

\[ \text{if (no unvisited vertices adjacent to the vertex on top of the stack)} \}

\[ s \text{.pop()} \]

\[ \text{no backtracks} \]

\[ \text{else} \}

\[ \text{select unvisited vertex } u \text{ adjacent to vertex on top of the stack} \]

\[ s \text{.push}(u) \]

\[ \text{mark } u \text{ as visited} \]

\}
DFS Example

A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P

Graph Traversal –
Breadth First Search (BFS)

BFS

Similar to level order tree traversal
Whereas DFS is a last visited first explored strategy, BFS is a first visited first explored strategy

BFS

bfs(in v:Vertex)
q – queue of nodes to be processed
q.enqueue(v)
mark v as visited
while(!q.isEmpty()) {
    w = q.dequeue()
    for (each unvisited vertex u adjacent to w) {
        mark u as visited
        q.enqueue(u)
    }
}
Not easy to design a recursive version!

Graph Traversal

Properties of BFS and DFS:
- Visit all vertices that are reachable from a given vertex
- Therefore DFS(v) and BFS(v) visit a connected component
- Computation time for DFS, BFS for a connected graph: $O(|V| + |E|)$
- How to find all connected components?

Reachability

- Reachability
  - $v$ is reachable from $u$ if there is a directed path from $u$ to $v$
  - solve using BFS or DFS
- Transitive Closure (G*)
  - $G^*$ has edge from $u$ to $v$ if $v$ is reachable from $u$. 
Graphs Describing Precedence

- Examples:
  - prerequisites for a set of courses
  - dependencies between programs
- Edge from \( a \) to \( b \) indicates \( a \) should come before \( b \)
- Want an ordering of the vertices of the graph that respects the precedence relation
- The graph does not contain cycles. Why?

Topological Sorting of DAGs

- DAG: Directed Acyclic Graph
- Topological sort: listing of nodes such that if \((a, b)\) is an edge, \( a \) appears before \( b \) in the list
- Is a topological sort unique?

Topological Sort – Algorithm 1

IDEA: nodes with no successors can be added to the back of the list

```
A, D, E, B, G, C, F, H, I
```

Topological Sort - Algorithm 1

topSort1(in \( G: \text{Graph} \))

\( n = \text{number of vertices in } G \)

for (step = 1 through \( n \))

select a vertex \( v \) that has no successors

\( \text{aList}.add(1, v) \)

Delete from \( G \) vertex \( v \) and its edges

return \( \text{aList} \)

Algorithm relies on the fact that in a DAG there is always a vertex that has no successors

Topological Sort - Algorithm 2

- Modification of DFS: Traverse tree using DFS starting from all nodes that have no predecessor.
- Add a node to the list when ready to backtrack.
- Show on previous example.

Topological Sort - Algorithm 2

topSort2(in theGraph: Graph): List

\( \text{s.createStack()} \)

for (all vertices \( v \) in the graph \( \text{theGraph} \))

if (\( v \) has no predecessors)

\( \text{s.push}(v) \)

Mark \( v \) as visited

while (\( \text{s.isEmpty}() \))

if (all vertices adjacent to the vertex on top of the stack have been visited)

\( v = \text{s.pop()} \)

\( \text{aList}.add(1, v) \)

else

Select an unvisited vertex \( u \) adjacent to vertex on top of the stack

\( \text{s.push}(u) \)

Mark \( u \) as visited

return \( \text{aList} \)