Data Structures and Algorithms

Chapter 8

Graphs

Werner Nutt
Acknowledgments

• The course follows the book “Introduction to Algorithms”, by Cormen, Leiserson, Rivest and Stein, MIT Press [CLRST]. Many examples displayed in these slides are taken from their book.

• These slides are based on those developed by Michael Böhlen for this course.

  (See http://www.inf.unibz.it/dis/teaching/DSA/)

• The slides also include a number of additions made by Roberto Sebastiani and Kurt Ranalter when they taught later editions of this course.

  (See http://disi.unitn.it/~rseba/DIDATTICA/dsa2011_BZ//)
1. Graphs – Principles
2. Graph representations
3. Traversing Graphs
   • Breadth-First Search
   • Depth-First Search
4. DAGs and Topological Sorting
1. Graphs – Principles
2. Graph representations
3. Traversing Graphs
   • Breadth-First Search
   • Depth-First Search
4. DAGs and Topological Sorting
Graphs – Definition

A graph \( G = (V, E) \) is composed of
- a set of vertices \( V \)
- a set of edges \( E \subseteq V \times V \) connecting the vertices

An edge \( e = (u, v) \) is a pair of vertices

We assume directed graphs
- if a graph is undirected, we represent an edge between \( u \) and \( v \) by two pairs \( (u, v) \in E \) and \( (v, u) \in E \)

\[ V = \{A, B, C, D\} \]
\[ E = \{(A,B), (B,A), (A,C), (C,A), (C,D), (D,C), (B,C), (C,B)\} \]

![Graph example](image-url)
Chapter 8

Graphs

Applications

• Electronic circuits, pipeline networks
• Transportation and communication networks
• Modeling any sort of relationships
  (between components, people, processes, concepts)
Graph Terminology

A vertex $v$ is adjacent to vertex $u$ iff $(u,v) \in E$

The degree of a vertex: # of adjacent vertices

A path is a sequence of vertices $v_1, v_2, \ldots, v_k$ such that $v_{i+1}$ is adjacent to $v_i$ for $i = 1, 2, \ldots, k - 1$
Graph Terminology/2

**Simple path** – a path with no repeated vertices

**Cycle** – a simple path, except that the last vertex is the same as the first vertex

**Connected** graph – any two vertices are connected by some path
Graph Terminology/3

Subgraph – a subset of vertices and edges forming a graph

Connected component – maximal connected subgraph.
Example: the graph below has 3 connected components
Graph Terminology/4

**Tree** – connected graph without cycles

**Forest** – collection of trees
DSA, Chapter 8: Overview

1. Graphs – Principles
2. Graph Representations
3. Traversing Graphs
   • Breadth-First Search
   • Depth-First Search
4. DAGs and Topological Sorting
Adjacency Matrix

Matrix $M$ with entries for all pairs of vertices

- $M[i,j] = true$ – there is an edge $(i,j)$ in the graph
- $M[i,j] = false$ – there is no edge $(i,j)$ in the graph

Needs space $\Theta(|V|^2)$
Data Structures for Graphs

• The **adjacency list** of a vertex $v$:
  sequence of vertices adjacent to $v$

• A graph is represented by the adjacency lists of all its vertices

• Needs space $\Theta(|V|+|E|)$
Pseudocode Assumptions

Each node has some properties (fields of a record):
- \texttt{adj}: list of adjacent nodes
- \texttt{dist}: distance from start node in a traversal
- \texttt{pred}: predecessor in a traversal
- \texttt{color}: color of the node (is changed during traversal: white, gray, black)
- \texttt{starttime}: time when first visited during a traversal (depth first search)
- \texttt{endtime}: time when last visited during a traversal (depth first search)
1. Graphs – Principles
2. Graph Representations
3. Traversing Graphs
   • Breadth-First Search
   • Depth-First Search
4. DAGs and Topological Sorting
Graph Searching Algorithms

• Systematic search of every edge and vertex of the graph
• Graph $G = (V, E)$ is either directed or undirected
• Applications
  – Memory management
    (Cheney algorithm for garbage collection)
  – Graphics (ray tracing)
  – Maze-solving
  – Networks: routing, searching, clustering, etc.
1. Graphs – Principles
2. Graph Representations
3. Traversing Graphs
   • Breadth-First Search
   • Depth-First Search
4. DAGs and Topological Sorting
Breadth-First Search

- A Breadth-First Search (BFS) traverses a connected component of an (un)directed graph, and in doing so defines a spanning tree.

- BFS in an undirected graph G is like wandering in a labyrinth with a string and exploring the neighborhood first.

- The starting vertex s, it is assigned distance 0.

- In the first round the string is unrolled 1 unit. All edges that are 1 edge away from the anchor are visited (discovered) and assigned distance 1.
Breadth-First Search/2

• In the second round, all the new edges that can be reached by unrolling the string 2 edges are visited and assigned a distance of 2

• This continues until every vertex has been assigned a level

• The label of any vertex \( v \) corresponds to the length of the shortest path (in terms of edges) from \( s \) to \( v \)
BFS Algorithm

### BFS($G, s$)

01  for $u \in G.V$
02      $u.color := \text{white}$
03      $u.dist := \infty$
04      $u.pred := \text{NULL}$
05  $s.color := \text{gray}$
06  $s.dist := 0$
07  $Q := \text{new Queue} \quad // \text{FIFO queue}$
08  $Q.enqueue(s)$
09  while not $Q.isEmpty()$
10      $u := Q.dequeue()$
11          for $v \in u.adj$ do
12              if $v.color = \text{white}$ then
13                  $v.color := \text{gray}$
14                  $v.dist := u.dist + 1$
15                  $v.pred := u$
16                  $Q.enqueue(v)$
17      $u.color := \text{black}$

**Initialize all vertices**

**Initialize BFS with $s$**

**Handle all of $u$'s children before handling children of children**
Coloring of Vertices

• A vertex is **white** if it is undiscovered
• A vertex is **gray** if it has been discovered but not all of its edges have been explored
• A vertex is **black** after all of its adjacent vertices have been discovered (the adj. list was examined completely)

Let's do an example of BFS:
BFS Running Time

Given a graph $G = (V,E)$

- Vertices are enqueued if their color is white
- Assuming that en- and dequeuing takes $O(1)$ time, the total cost of this operation is $O(V)$
- Adjacency list of a vertex is scanned when the vertex is dequeued
- The sum of the lengths of all lists is $\Theta(E)$
  Thus, $\Theta(E)$ time is spent on scanning them
- Initializing the algorithm takes $\Theta(V)$

Total running time is $\Theta(V+E)$
(linear in the size of the adjacency list representation of $G$)
BFS Properties

Given a graph $G = (V,E)$.

Then BFS

- discovers all vertices reachable from a source vertex $s$,
- computes the shortest distance to all reachable vertices,
- computes a breadth-first tree that contains all such reachable vertices.

For any vertex $v$ reachable from $s$,

the path in the breadth first tree from $s$ to $v$

corresponds to a shortest path in $G$
1. Graphs – Principles
2. Graph representations
3. Traversing Graphs
   • Breadth-First Search
   • Depth-First Search
4. DAGs and Topological Sorting
Depth-First Search

A depth-first search (DFS) in an undirected graph $G$ is like wandering in a labyrinth with a string and following one path to the end

- We start at vertex $s$, tying the end of our string to the point and painting $s$ “visited (discovered)”; next we label $s$ as our current vertex called $u$
- Now, we travel along an arbitrary edge $(u,v)$
- If edge $(u,v)$ leads us to an already visited vertex $v$, we return to $u$
- If vertex $v$ is unvisited, we unroll our string, move to $v$, paint $v$ “visited”, set $v$ as our current vertex, and repeat the previous steps
Depth-First Search/2

- Eventually, we will get to a point where all edges from $u$ lead to visited vertices.
- We then backtrack by rolling up our string until we get back to a previously visited vertex $v$.
- $v$ becomes our current vertex and we repeat the previous steps.
**DFS Algorithm**

**DFS-All** \((G)\)

```
01 for u ∈ G.V
02   u.color := white
03   u.pred := NIL
04   time := 0
05 for u ∈ G.V
06   if u.color = white then DFS(u)
```

**DFS(u)**

```
01 u.color := gray
02 time := time + 1
03 u.starttime := time
04 for v ∈ u.adj
05   if v.color = white then
06     v.pred := u
07     DFS(v)
08 u.color := black
09 time := time + 1
10 u.endtime := time
```

- **Init all vertices**
- **Visit all vertices**
- **Visit all children recursively** (children of children are visited first)
DFS Algorithm/2

• Initialize – color all vertices white
• Visit each and every white vertex using DFS-All (even if there are disconnected trees)
• Each call to DFS(u) roots a new tree of the depth-first forest at vertex u
• When DFS returns, each vertex $u$ has assigned
  – a discovery time $d[u]$  
  – a finishing time $f[u]$
Example of DFS

• Start with s:

• Explores subgraph s first, t second
DFS Algorithm: Running Time

Running time

- the loops in DFS-All take time $\Theta(V)$ each, excluding the time to execute DFS
- DFS is called once for every vertex
  - it's only invoked on white vertices, and
  - paints the vertex gray immediately
- for each DFS a loop iterates over all $v.\text{adj}$

$$\sum_{v \in V} |v.\text{adj}| = \Theta(E)$$

- the total cost for DFS is $\Theta(E)$
- the running time of DFS-All is $\Theta(V+E)$
**DFS versus BFS**

- The BFS algorithms visits all vertices that are reachable from the start vertex. It returns one search tree.
- The DFS-All algorithm visits all vertices in the graph. It may return multiple search trees.
- The difference comes from the applications of BFS and DFS. This behavior of the algorithms depends on the policy according to which the next nodes to be processed are determined.
Generic Graph Search

```
GenericGraphSearch(G,s)
01 for each vertex u ∈ G.V { u.color := white; u.pred := NIL }
04 s.color := gray
05 init(GrayVertices)
06 addTo(GrayVertices,s)
07 while not isEmpty(GrayVertices)
08   u := extractFrom(GrayVertices)
09     for each v ∈ u.adj do
10       if v.color = white then
11         v.color := gray
12         v.pred := u
13         addTo(GrayVertices,v)
14     u.color := black
```

- **BFS** if GrayVertices is a **Queue** (FIFO)
- **DFS** if GrayVertices is a **Stack** (LIFO)
DFS Annotations

• A DFS can be used to annotate vertices and edges with additional information.
  – starttime (when was the vertex visited first)
  – endtime (when was the vertex visited last)
  – edge classification (tree, forward, back or cross edge)

• The annotations reveal useful information about the graph that is used by advanced algorithms
DFS Timestamping

• Vertex $u$ is
  – white before $u.starttime$
  – gray between $u.starttime$ and $u.endtime$, and
  – black after $u.endtime$

• Notice the structure throughout the algorithm
  – gray vertices form a linear chain
  – corresponds to a stack of vertices that have not been exhaustively explored (DFS started but not yet finished)
DFS Parenthesis Theorem

• Start and end times have parenthesis structure
  – represent \textit{starttime} of \textit{u} with \textit{left parenthesis} "(u"
  – represent \textit{endtime} of \textit{u} with \textit{right parenthesis} "u)"
  – history of start- and endtimes makes a well-formed expression (parentheses are properly nested)

• Intuition for proof:
  any two intervals are either disjoint or enclosed
  – Overlapping intervals would mean finishing ancestor, before finishing descendant or starting descendant without starting ancestor
DFS Parenthesis Theorem/2
DFS Edge Classification

- Tree edge (gray to white)
  - Edges in depth-first forest
- Back edge (gray to gray)
  - from descendant to ancestor in depth-first tree
  - Self-loops
DFS Edge Classification

• Forward edge (gray to black)
  – Nontree edge from ancestor to descendant in depth-first tree
• Cross edge (gray to black)
  – remainder – between trees or subtrees
DFS Edge Classification/3

• In a DFS the color of the next vertex decides the edge type (this makes it impossible to distinguish forward and cross edges)

• Tree and back edges are important

• Most algorithms do not distinguish between forward and cross edges
Suggested exercises

• Implement BFS and DFS, both iterative and recursive

• Using paper & pencil, simulate the behaviour of BFS and DFS (and All-DFS) on some graphs, drawing the evolution of the queue/stack
1. Graphs – Principles
2. Graph representations
3. Traversing Graphs
   • Breadth-First Search
   • Depth-First Search
4. DAGs and Topological Sorting
Directed Acyclic Graphs (DAGs)

• A DAG is a directed graph without cycles

• DAGs are used to indicate precedence among events (event x must happen before y)

• An example would be a parallel code execution

• We get a total order using Topological Sorting
DAG Theorem

A directed graph $G$ is acyclic if and only if a DFS of $G$ yields no back edges.

Proof: Suppose there is a back edge $(u,v)$: Then $v$ is an ancestor of $u$ in DFS forest. Thus, there is a path from $v$ to $u$ in $G$ and $(u,v)$ completes the cycle. Suppose there is a cycle $c$: Let $v$ be the first vertex in $c$ to be discovered and $u$ is the predecessor of $v$ in $c$.

- Upon discovering $v$ the whole cycle from $v$ to $u$ is white
- We visit all nodes reachable on this white path before DFS($v$) returns, i.e., vertex $u$ becomes a descendant of $v$
- Thus, $(u,v)$ is a back edge

Thus, we can verify whether $G$ is a DAG using DFS
Topological Sorting Example

- Precedence relations: an edge from $x$ to $y$ means one must be done with $x$ before one can do $y$
- Intuition: can schedule task only when all of its precondition subtasks have been scheduled
Topological Sorting/1

• Sorting of a directed acyclic graph (DAG)
• A topological sort of a DAG is a linear ordering of all its vertices such that for any edge \((u,v)\) in the DAG, \(u\) appears before \(v\) in the ordering
Topological Sorting/2

The following algorithm topologically sorts a DAG

The linked lists comprises a total ordering

```
TopologicalSort(G)
Call DSF(G) to compute
    v.endtime for each vertex v
As each vertex is finished,
insert it at the beginning of a linked list
Return the linked list of vertices
```
Topological Sorting Correctness

Claim: If $G$ is a DAG and $(u,v) \in E \Rightarrow u.\text{endtime} > v.\text{endtime}$

• When $(u,v)$ is explored, $u$ is gray. We can distinguish three cases:
  
  – $v.\text{color} = \text{gray} \Rightarrow (u,v)$ is a back edge (cycle, contradiction)
  
  – $v.\text{color} = \text{white} \Rightarrow v$ becomes descendant of $u$
    
    $v$ will be finished before $u$
    
    $v.\text{endtime} < u.\text{endtime}$

  – $v.\text{color} = \text{black} \Rightarrow v$ is already finished
    
    $v.\text{endtime} < u.\text{endtime}$

• The definition of topological sort is satisfied
Topological Sorting: Running Time

• Running time
  – depth-first search: $O(V+E)$ time
  – insert each of the $|V|$ vertices to the front of the linked list: $O(1)$ per insertion

• Thus the total running time is $O(V+E)$
Suggested Exercises

- Implement topological sorting, with a check for the DAG property

- Using paper & pencil, simulate the behaviour of topological sorting
Summary

• Graphs
  – $G = (V,E)$, vertex, edge, (un)directed graph, cycle, connected component, ...
• Graph representation: adjacency list/matrix
• Basic techniques to traverse/search graphs
  – Breadth-First Search (BFS)
  – Depth-First Search (DFS)
• Topological Sorting