Design and Analysis of Algorithms | CS 323
Lecture #8 | 30 March 2016
Notes by: Anna Ip

Announcements #35 (Summary)

Exam #1 was returned to the students. 27 students took the exam. The high score was 79/70, the median was 50/70, and the mean was 52/70. The exam and solutions are available on-line. We briefly discussed question 8 in class as the question was not understood in two respects - the need for the comparison based model (aka decision tree) to establish a lower bound for comparison-based sorting, and that little oh and big -oh are different generally, including as a time complexity for sorting.

We then began our unit on graph algorithms, including elements of graph theory, representations of graphs, graph traversals (depth-first, breadth-first, and topological), and different implementations of Dijkstra's single-source shortest path algorithm. The relevant sections in our text are 11.1-11.4.

Next week we will present another implementation of Dijkstra's SSSP algorithm, followed by Prim's and Kruskal's minimum spanning tree algorithms, and Floyd's all-pair's shortest path (APSP) algorithm. This will entail some background on important topics in algorithms - the greedy strategy, amortized analysis and dynamic programming.

If you haven't done so already, please submit your topic for the term project. (By this point, all topics should have been chosen and work should be in progress.)

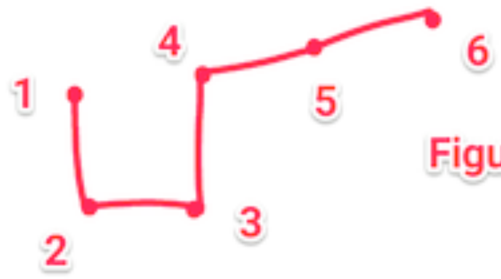


Figure 1

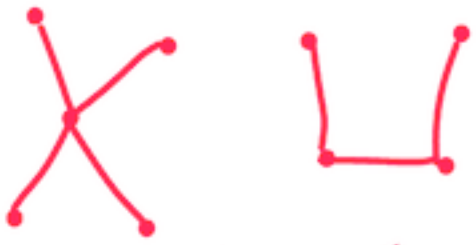
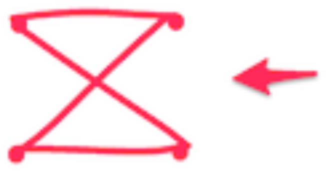


Figure 2

Trees



C[4]

Figure 3

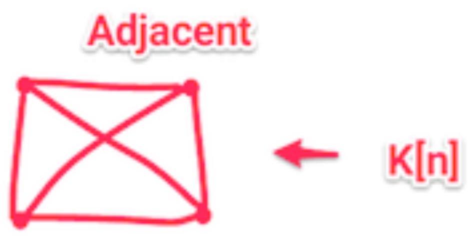


Figure 4

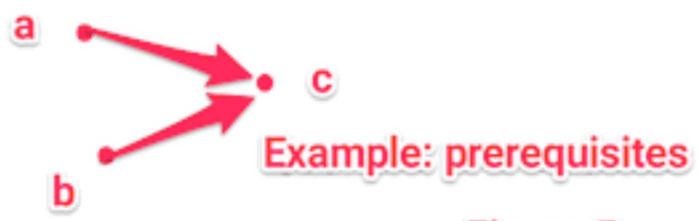


Figure 5

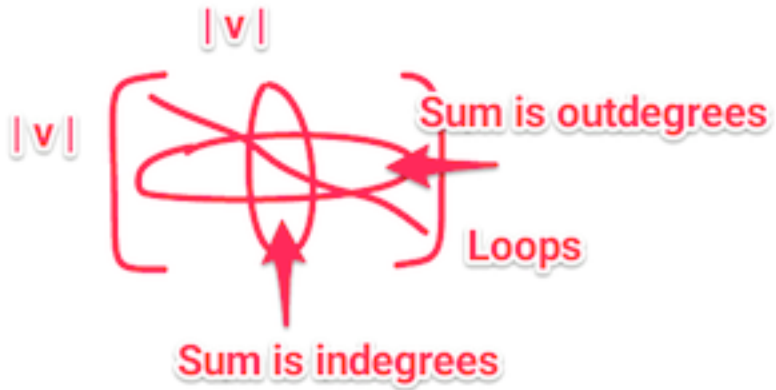


Figure 6

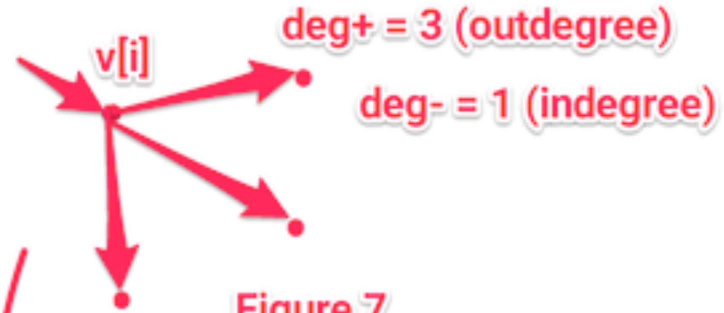


Figure 7

$$\begin{array}{c|ccc} & e1 & e2 & e3 \\ \hline v1 & 1 & 1 & 1 \\ v3 & & & \\ v4 & & & \\ v6 & & & \end{array}$$

Figure 8

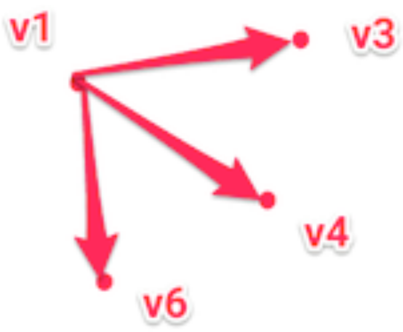


Figure 9

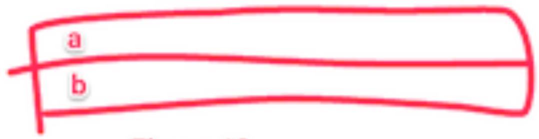


Figure 10

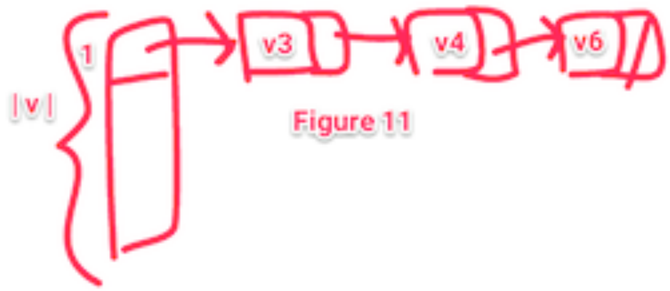


Figure 11

Figure 12

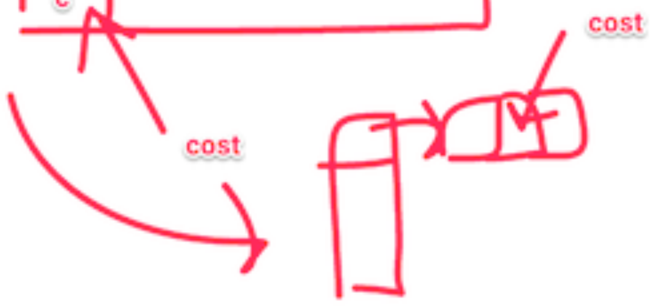
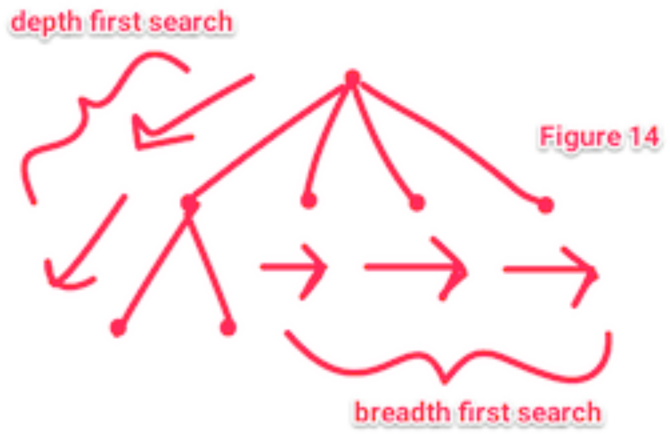


Figure 13


depth first search
Figure 14
breadth first search

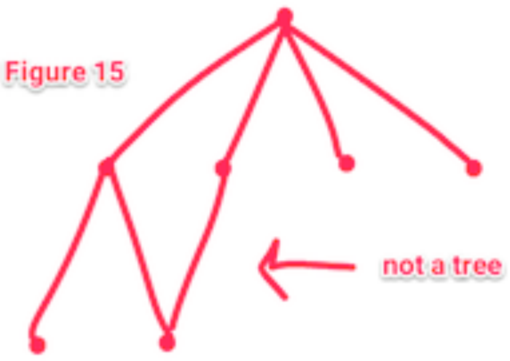
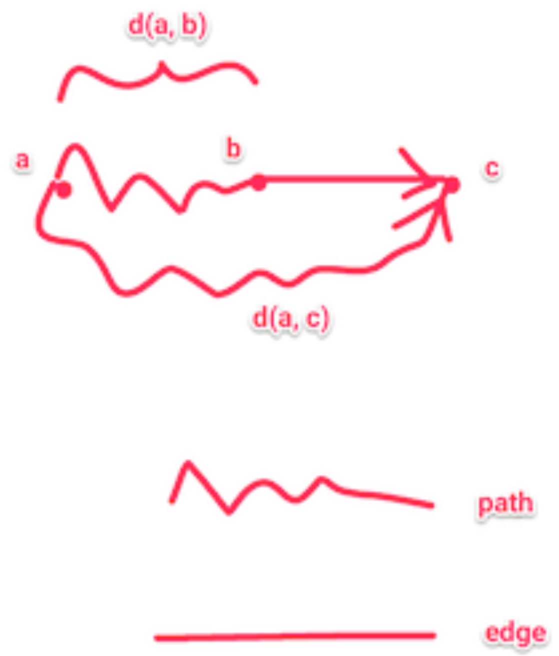


Figure 15
not a tree

Figure 16



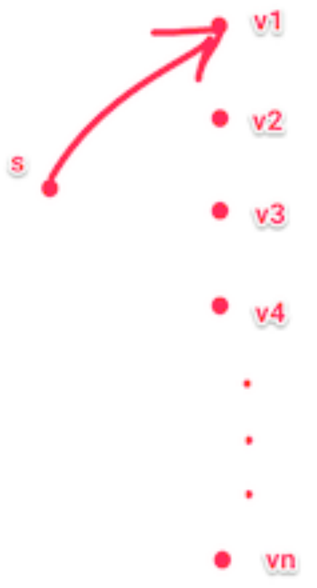


Figure 17

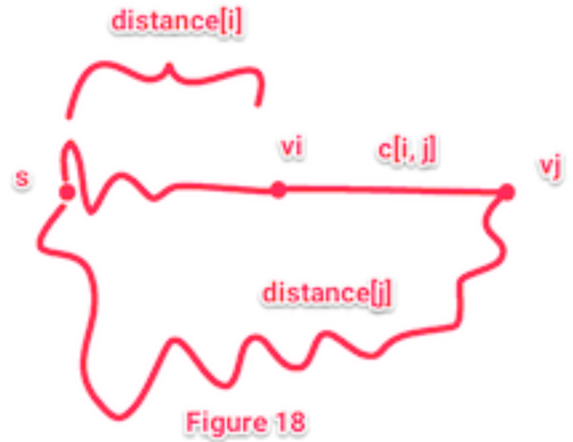


Figure 18

Graph Algorithms

G = (V, E)

V: vertices
E: edges

Figure 1

V = { 1, 2, 3, 4, 5, 6 }
E = { [1, 2], [2, 3], [3, 4], [4, 5], [5, 6] }

Cartesian Product

A x B = { [a, b] | a $\in$ A, b $\in$ B }

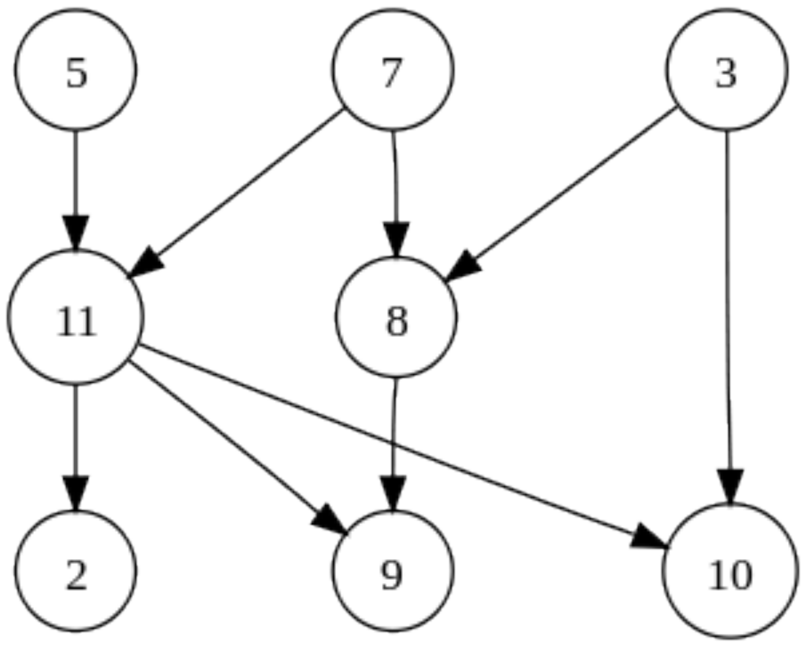
Relation (on A x B): subset of Cartesian Product

Example: A x A

[S1, S2] $\in$ R
[S3, S4] $\in$ R

Directed Graph

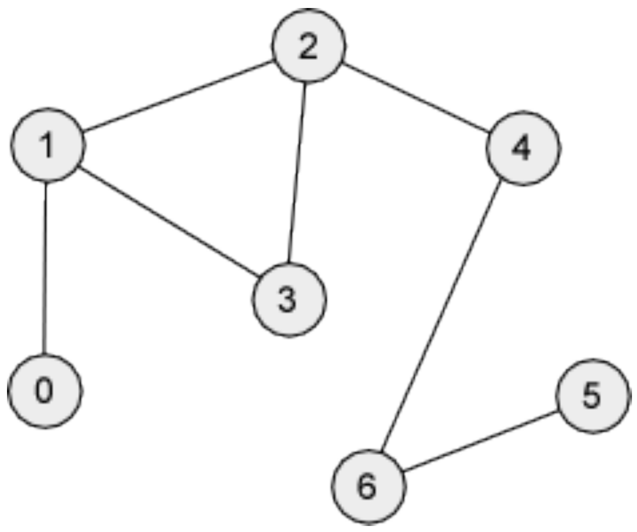
A directed graph is graph, i.e., a set of objects (called vertices or nodes) that are connected together, where all the edges are directed from one vertex to another. A directed graph is sometimes called a digraph or a directed network. (mathinsight.org/definition/directed_graph)



(https://upload.wikimedia.org/wikipedia/commons/thumb/0/03/Directed_acyclic_graph_2.svg/305px-Directed_acyclic_graph_2.svg.png)

<u>Undirected Graph</u>

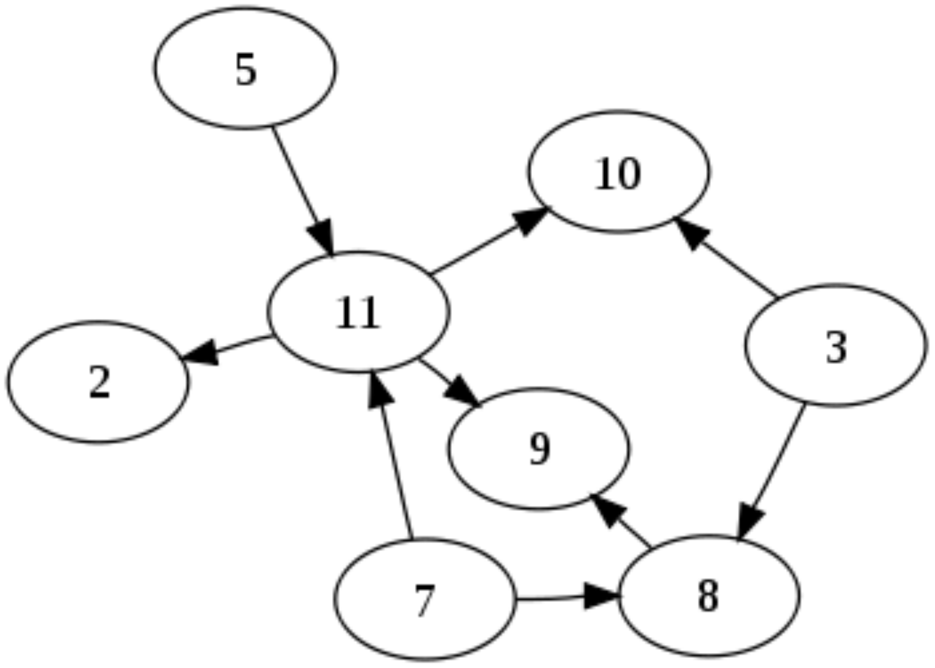
An undirected graph is graph, i.e., a set of objects (called vertices or nodes) that are connected together, where all the edges are bidirectional. An undirected graph is sometimes called an undirected network. In contrast, a graph where the edges point in a direction is called a directed graph. (http://mathinsight.org/definition/undirected_graph)



(http://i.stack.imgur.com/mPzx7.gif)

DAG: Directed Acyclic Graph

In mathematics and computer science, a directed acyclic graph is a directed graph with no directed cycles. That is, it is formed by a collection of vertices and directed edges, each edge connecting one vertex to another, such that there is no way to start at some vertex v and follow a sequence of edges that eventually loops back to v again. (https://en.wikipedia.org/wiki/Directed_acyclic_graph)



(https://upload.wikimedia.org/wikipedia/commons/3/39/Directed_acyclic_graph_3.svg)

All trees are graphs, but not all graphs are trees.

Figure 2

C[n] <-- Cycle Graph

Figure 3

n(n-1)/2 --> C(n, 2) = nC2: n choose 2

Figure 4

K[n] <-- Complete Graph = there is an edge between every vertex (but no edge from a vertex to itself --> no loop)

Figure 5

Figure 6 <-- Adjacency Matrix (A)

A[i, j] = { 1 if ∋ an edge between v[i] and v[j] (they are adjacent), 0 otherwise

Figure 7

A = A ^ T (Transpose)

The transpose of a matrix is a new matrix whose rows are the columns of the original. (This makes the columns of the new matrix the rows of the original). (https://chortle.ccsu.edu/VectorLessons/vmch13/vmch13_14.html)

$$\begin{pmatrix} 5 & 4 \\ 4 & 0 \\ 7 & 10 \\ -1 & 8 \end{pmatrix}^{T}_{4 \times 2} = \begin{pmatrix} 5 & 4 & 7 & -1 \\ 4 & 0 & 10 & 8 \end{pmatrix}_{2 \times 4}$$

(https://chortle.ccsu.edu/VectorLessons/vmch13/mtrxTrans2.gif)

ϴ(| V | ^ 2)

G = (a, b) ∈ E

Figure 10

Figure 9

Figure 11 <-- Adjacency Table (List)

ϴ(| V | + | E |)

time complexity to find outdegree: outdegree, indegree: proportionate to v

Figure 8 <-- Incidence Matrix

ϴ(| V | * | E |)

Weighted Graph

c[i, j] <-- weight or cost of edge (i, j)

Figure 12

Figure 13

adjacent to <-- there is a edge between the vertices
connected to <-- there is a path between the vertices

If there is no edge, make the cost ∞.

Figure 14

Figure 15

search a graph --> traversal

bool value isVisited

Depth First Search (DFS) <-- Θ(| V | + | E |)

Depth-first search (DFS) is an algorithm for traversing or searching tree or graph data structures. One starts at the root (selecting some arbitrary node as the root in the case of a graph) and explores as far as possible along each branch before backtracking. (https://en.wikipedia.org/wiki/Depth-first_search)

Breadth First Search (BFS) <-- Θ(| V | + | E |)

Breadth-first search (BFS) is an algorithm for traversing or searching tree or graph data structures. It starts at the tree root (or some arbitrary node of a graph, sometimes referred to as a 'search key') and explores the neighbor nodes first, before moving to the next level neighbors. (https://en.wikipedia.org/wiki/Breadth-first_search)

Topological Search <-- Θ(| V | + | E |)

In the field of computer science, a topological sort (sometimes abbreviated toposort) or topological ordering of a directed graph is a linear ordering of its vertices such that for every directed edge uv from vertex u to vertex v, u comes before v in the ordering. (https://en.wikipedia.org/wiki/Topological_sorting)

put all the children in the queue
dequeue

```
main
   for i = 1 to n
      if !visited dfs(v)
```

dfs(vertex)

Topological Search <-- can't have a cycle
   flattens to linear structure
   honors prerequisite structure

Shortest Path

In graph theory, the shortest path problem is the problem of finding a path between two vertices (or nodes) in a graph such that the sum of the weights of its constituent edges is minimized.
(https://en.wikipedia.org/wiki/Shortest_path_problem)

SSSP (single-source shortest path)

The single-source shortest path problem, in which we have to find shortest paths from a source vertex v to all other vertices in the graph.
(https://en.wikipedia.org/wiki/Shortest_path_problem)

APSP (all-pairs shortest path)

The all-pairs shortest path problem, in which we have to find shortest paths between every pair of vertices v, v' in the graph.
(https://en.wikipedia.org/wiki/Shortest_path_problem)

negative-weight edge

Figure 16

Dijkstra's SSSP

Dijkstra's algorithm solves the single-source shortest path problem.
(https://en.wikipedia.org/wiki/Shortest_path_problem#Single-source_shortest_paths)

Dijkstra's algorithm is an algorithm for finding the shortest paths between nodes in a graph, which may represent, for example, road networks. It was conceived by computer scientist Edsger W. Dijkstra in 1956 and published three years later. (https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm)

$\Theta(n^3)$ <-- n = | v |

Ɵ(n^2)
Ɵ(| E | log | V |)

```
init:
for i = 1 to | v |
    distance[i] = c[s, i]
    pred[i] = s

visit neighbors of i:
if (distance[j] = distance[i] + c[i, j]
    pred[j] = i
```

Figure 17

Ɵ(| V | ^ 2) <-- naive approach (array to store distances, iterate through array to find smallest, etc.)
Ɵ(| V | log | V | + | V | ^ 2 log | V |) <-- Ɵ(min heap + adjacency matrix)
Ɵ(| V | log | V | + | E | log | V |) <-- Ɵ(min heap + adjacency list)