CPSC 320: Intermediate Algorithm Design & Analysis

Greedy Algorithms and Graphs

Steve Wolfman

Problem-Solving Approaches

Many problems can be solved by the same broad style of approach. We'll run into several of these styles:

- Input consuming (like insertion sort)
- Output producing (like selection sort)
- Divide-and-Conquer (like merge sort)
- ***Greedy*** (like change-making)
- Dynamic Programming

Optimization Problems

In an optimization problem, we want to report a* *best* answer according to some metric (i.e., a function that maps correct solutions to their value, where lower values are better).

Example: coin changing. Given an amount of change to make, give that amount of change *with the fewest coins*.

* Why do we say "a best answer" and not "the best answer"?

Greedy (and not) Coin Changing

- Coin changing with Cdn coins:
  penny, nickel, dime, quarter
  - Repeatedly add the largest coin that "fits" in the change remaining to be made.
- Coin changing with Cdn coins but no nickels:
  penny, dime, quarter
  - Does the algorithm above work? (Try $0.30.)
- Coin changing with Cdn coins plus the bauxel:
  penny, nickel, dime, bauxel ($0.15), quarter
  - Is there only ***one*** best solution? (Try $0.30.)

Greedy Algorithms

- Repeatedly make the "locally best choice" until the choices form a complete solution.

More Greedy (or not) Problems

- Activity Selection
- Minimum Spanning Tree
- Shortest Path

The latter two are graph problems; so, some graph review is in order...

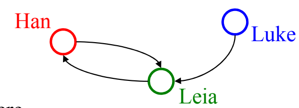
Interesting Properties for Greedy Algorithms

- Optimal substructure: An optimal solution to the problem is composed of pieces which are themselves optimal solutions to subproblems.
- Greedy-choice property: locally optimal (greedy) choices can be extended to a globally optimal solution.

Graph ADT

Graphs are a formalism useful for representing relationships between things

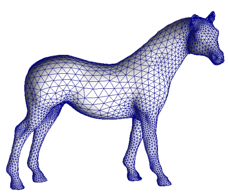
- a graph `G` is represented as `G = (V, E)`
  - `V` is a set of vertices: $\{v_1, v_2, \ldots, v_n\}$
  - `E` is a set of edges: $\{e_1, e_2, \ldots, e_m\}$ where each e_i connects two vertices (v_{i1}, v_{i2})
- operations might include:
  - creation (with a certain number of vertices)
  - inserting/removing edges
  - iterating over vertices adjacent to a specific vertex
  - asking whether an edge exists connecting two vertices



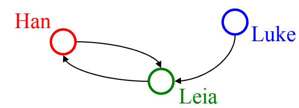

```
V = {Han, Leia, Luke}
E = {(Luke, Leia),
     (Han, Leia),
     (Leia, Han)}
```


Graph Applications

- Storing things that are graphs by nature
  - distance between cities
  - airline flights, travel options
  - relationships between people, things
  - distances between rooms in Clue
- Compilers
  - *callgraph* - which functions call which others
  - *dependence graphs* - which variables are defined and used at which statements
- Others: mazes, circuits, class hierarchies, *horses*, networks of computers or highways or…



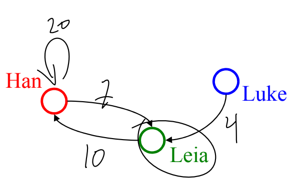
Graph Representations



- 2-D matrix of vertices (marking edges in the cells)
  "adjacency matrix"
- List of vertices each with a list of adjacent vertices
  "adjacency list"

Adjacency Matrix

A `|V| x |V|` array in which an element `(u, v)` is true if and only if there is an edge from `u` to `v`



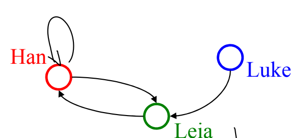
	Han	Luke	Leia
Han	20	0	7
Luke	0	0	4
Leia	10	0	0

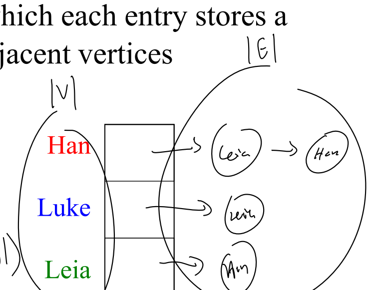
bad one: list vert adj $O(|V|)$

runtime for various operations? space requirements: $O(|V|^2)$

Adjacency List

A `|V|`-ary list (array) in which each entry stores a list (linked list) of all adjacent vertices





good one: adj vert $O(|V|)$ or $O(|\deg(v)|)$

bad one: does edge exist? $O(|V|)$ or $O(|\deg(v)|)$

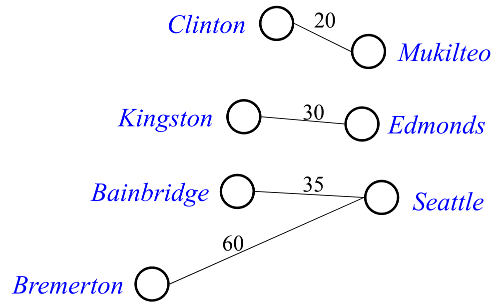
runtime for various operations? space requirements: $O(|V|+|E|)$

Directed vs. Undirected Graphs

- Adjacency lists and matrices both work fine to represent *directed* graphs.
- To represent *undirected* graphs, either ensure that both orderings of every edge are included in the representation or ensure that the order doesn't matter (e.g., always use a "canonical" order), which works poorly in adjacency lists.

Weighted Graphs

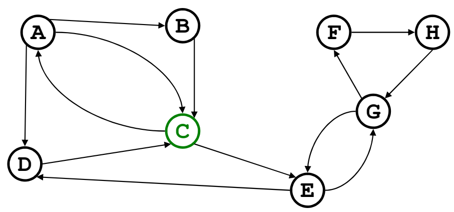
Each edge has an associated weight or cost.



As far as we're concerned, weight is a function from the set of nodes to the reals.

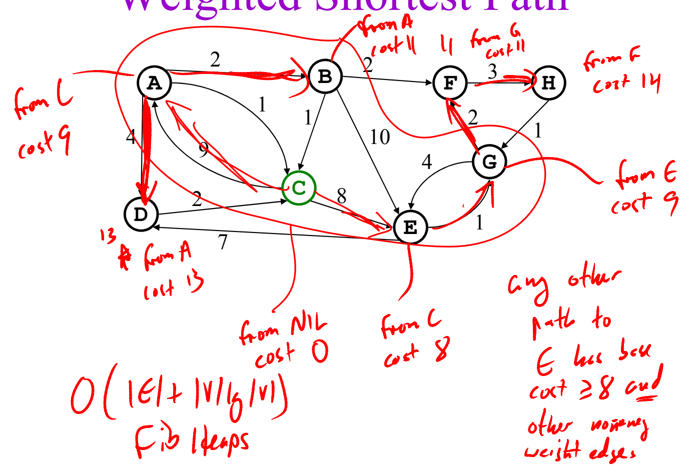
Unweighted Shortest Path Problem

Assume source vertex is **C**…



Distance to: **A B C D E F G H**

Weighted Shortest Path

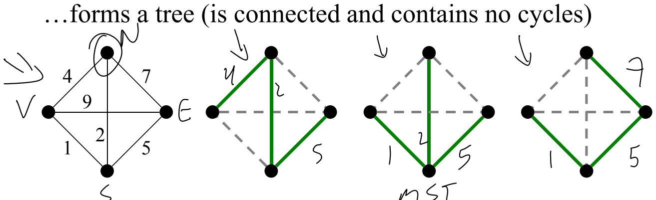


Spanning Tree

Spanning tree: a subset of the edges from a connected graph that…

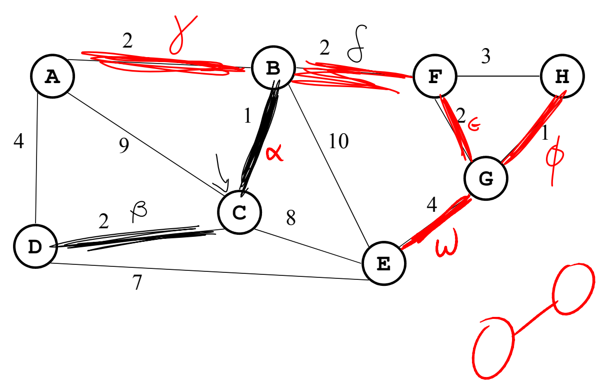
…touches all vertices in the graph (*spans* the graph)

…forms a tree (is connected and contains no cycles)

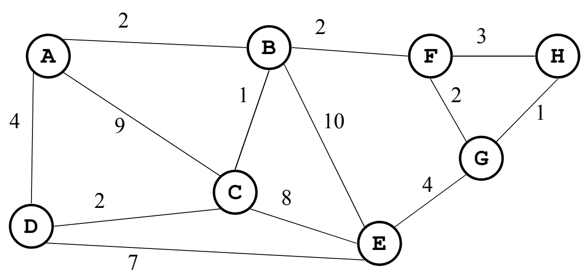


Minimum spanning tree: the spanning tree with the least total edge cost.

Prim's Algorithm Sample Graph



Kruskal's Algorithm Sample Graph



What's Next?

- Dynamic Programming: CLRS Chapter 15

On Your Own

- Review graphs
- Practice designing and analyzing greedy algorithms for optimality/performance.
- Play around with problems to see when small changes can keep a greedy algorithm from working (like dropping nickels from coin changing).