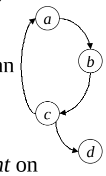
CSE 373: Graphs

Pete Morcos
University of Washington
5/3/00

http://www.cs.washington.edu/education/courses/cse373/00sp

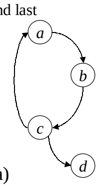
Definition

- A graph is a set V of *vertices* and E of *edges*
  - Vertices (singular vertex) also known as nodes
  - Edges in example are (*a*,*b*), (*b*,*c*), (*c*,*a*), (*c*,*d*)
  - Edges can be *directed* as in example, or *undirected*
- Vertices *a* and *b* are *adjacent* if there is an edge (*a*,*b*) or (*b*,*a*) in E
  - (*a*,*b*) and (*b*,*a*) are the same if the graph is undirected
- Edges travelling into a vertex are *incident* on that vertex, those leaving are *outgoing*



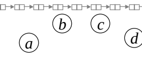
Paths and Cycles

- A *path* is a sequence of vertices connected by edges, e.g. *bcd* (but not *dcb*)
  - Formally, a sequence $v_1 \ldots v_n$, where $(v_i, v_{i+1}) \in E$
  - A *simple path* has no repeated vertices, except the first and last can be the same
  - Path length is number of edges, i.e. n-1
- A *cycle* is a path that begins and ends at the same vertex (e.g. *abca*)
  - Graphs without cycles are *acyclic*
  - Directed acyclic graphs common, abbreviated DAG
- Sometimes we allow *loop* edges in graphs, e.g. (a,a)
  - Loops aren't cycles (i.e. cycles have length > 1)

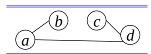


Connectivity

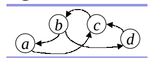
- A graph can have no edges at all...



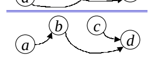
- If any pair of nodes in an undirected graph have a path between them, the graph is *connected*



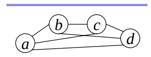
  - In a directed graph, this is called *strongly connected*



  - A directed graph that would be connected if the edges were undirected is called *weakly connected*

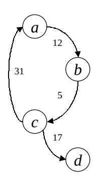


- Graph is *complete* if there is an edge between every pair of nodes



Weighted Graphs

- Very common to associate a numeric *weight* or *cost* to each edge in a graph
  - e.g. cost of path *abcd* is 12+5+17
- Path length is different from path cost
  - *abcd* has length 3, cost 34
- Could also associate costs with vertices

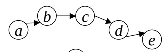


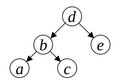
Why Graphs?

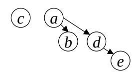
- Useful whenever we want to express relationships between things
  - Road distances between cities on a map
  - Airline flight costs between airports
  - Wires connecting electrical pins on a chip
  - Soap opera relationships (Cliff loves Jade, Jade loves Rod, Buffy loves Angel, ...)
  - Call structures in programs (main calls qsort, main calls printf, qsort calls partition, ...)
- Very, very common in computer science

Graphs vs. Other Data Structures

- Graphs can be seen as a generalized version of many structures we've seen
  - Lists: directed, N-1 edges, all nodes except ends have exactly one incident and one outgoing edge
  - Trees: directed, N-1 edges, all nodes except root have one incident and up to *k* outgoing edges
  - Heaps, binomial queues, disjoint sets are also graphs





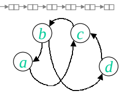


Useful Edge Properties to Know

- *(Assume undirected graphs for this slide)*
- A graph can have as few as 0 edges
- Let N = | V |, i.e. the number of vertices in V
- A complete, undirected graph has N(N-1)/2 edges
  - Would be $N^2/2$ if we allowed loops
  - Directed graph can have twice as many edges
- A connected graph has at least N-1 edges
  - If there are exactly N-1 edges then there are no cycles

Implementing Graphs with Arrays

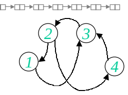
- Adjacency Matrix – use a 2D array
  - A[i][j] = 1 if (i,j) ∈ E, 0 otherwise
  - For weighted graphs, use cost instead of 1
  - Time to access edge is O(1)
  - But space cost is O($|V|^2$)
    - even if |E| << |V|, known as a *sparse* graph
    - e.g. 10,000 students, each with up to 10 friends; need 100M cells to hold 100,000 friendships
  - For undirected graphs, note that half of array is redundant since A[i][j] == A[j][i]

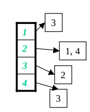


	a	b	c	d
a	0	0	1	0
b	1	0	0	1
c	0	1	0	0
d	0	0	1	0

Implementing Graphs with Lists

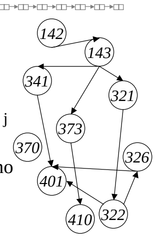
- Adjacency List – use array of lists
  - For each vertex, a list of adjacent vertices
  - Only requires O(|E| + |V|) space
  - But takes longer to check for an edge
  - How do we access array if nodes are not named with numbers?
  - For undirected graphs, usually still use redundant space by storing info on both sides, for convenience
- Either representation OK, so a good reason to use ADT operations and hide the implementation





Topological Sort

- A simple "starter" problem to get used to working with graphs
- Turns a DAG into a list $v_1 v_2 ... v_n$ as follows:
  - If there is a path from v_i to v_j in the graph, then i < j
  - (There can't also be a path from v_j to v_i—why?)
- Example: assign numbers to CSE classes so no class has a higher-numbered prerequisite.
- Many orderings possible—most graphs only specify a partial order on the nodes
  - e.g. 373 and 326 can appear in any order

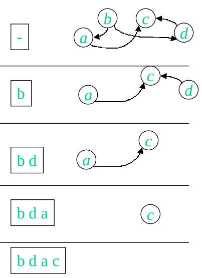


Topological Sort Algorithm

- Repeat:
  - Pick any node *n* with zero incident edges
    - Let's call this an "in-degree" of zero
    - There must be at least one such node if there are no cycles
  - Append *n* to list (list is initially empty)
  - Remove *n* and all outgoing edges from graph
- Naive impl.: first step is O(|V|), repeated |V| times = O($|V|^2$)
- Make scan more efficient by maintaining stack or queue of zero-incident nodes, and tracking in-degree for all nodes.
  - Each time a node is removed, decrement in-degrees of adjacent nodes.
  - Any node that drops to in-degree==0 is added to stack.
- If adjacency lists used, time is now O(|V| + |E|)

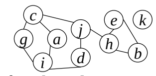
Topological Sort Example

- Indegrees: a=1,b=0,c=2,d=1
  - Choose b
- Indegrees: a=0, c=2, d=0
  - Choose d (could choose a)
- Indegrees: a=0, c=1
  - Choose a
- Indegrees: c=0
  - Choose c, done



Breadth-first search (BFS)

- Suppose we want to find the shortest path from node *a* to node *b*, if there is one
- Starting at node *a*:
  - Step along each of its edges
  - If *b* not found, step along each edge of each node you saw
    - Skip already-visited nodes
  - Repeat, and fail when no new nodes are reachable
- If we do this over the whole graph, the visited edges form a *spanning tree* for the graph
  - Actually, just for the part of graph reachable from *a*
- O(|V| + |E|) time if using adjacency lists



BFS Shortest Path Pseudocode

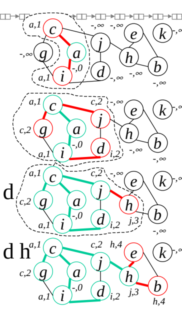
- Use a queue to track which nodes need to be expanded
- For each node, store distance and previous node in the path

```
curdist = dist[a] = 0
enqueue(a)
while (queue not empty)
   x = dequeue
   for each node y adjacent to x and not visited
        prev[y] = x
        dist[y] = dist[x] + 1
        enqueue y
```

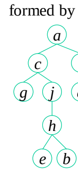
- We can now use 'prev' to recover the shortest path
- Items underlined change for weighted graphs

BFS Shortest Path Example

- Visited: a
  - Adjacent: c i
- Visited: a c i
  - Adjacent: g j d
- Visited: a c i g j d
  - Adjacent: h
- Visited: a c i g j d h
  - Adjacent: e b



Spanning tree formed by BFS:



Traversal order of this tree sometimes called "level order"

About BFS

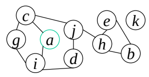
- BFS refers to a particular traversal order
  - "Expanding circles", each one hop farther than last
  - Cycles not a problem, since we track visited nodes
- Because of this traversal order, BFS is a suitable algorithm for finding the shortest path
  - We know that any nodes not yet visited must be more hops from start node than all nodes seen so far
- But, BFS also used for other purposes
- BFS works fine on directed graphs as well

Depth-first search (DFS)

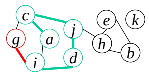
- DFS searches down one path as far as it can go
  - When no new nodes available, it *backtracks*
  - Backtracking reveals side paths that weren't taken
- Naturally recursive, whereas BFS is naturally iterative
- You usually use DFS to scan a graph unless you specifically need the BFS traversal order
  - DFS requires less bookkeeping
    - BFS uses queue because its traversal doesn't follow the graph edges
  - Possible to find target faster, if good branching choices made

DFS Example

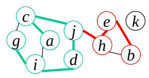
- Start at *a*



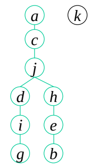
- Visit *c, j, d, i, g* at *g*, must backtrack
    - *i* and *d* have no new neighbors



- Backtrack to *j*
    - try *h*
    - then continue with *e, b*



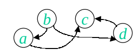
Spanning tree formed by DFS:



This tree was traversed in preorder

DFS Forests

- On an unconnected graph, or a weakly connected directed graph, DFS may not visit all nodes
- Solution is to repeatedly pick an unvisited node and run DFS again with that node as root
- The result is a DFS forest
- Note that in a directed graph, there may be graph edges that would connect the trees if the DFS had chosen different starting roots
    - e.g. start with b, or start with d



DFS Pseudocode

- Call DFS on each vertex *v* in V

```
DFS(v)
  if v is unvisited
     mark v as visited
     for each edge (v,w)
          DFS(w)
```

- We can use DFS to do topological sort
    - At end of DFS function, prepend *v* to the result list

Various Graph Problems

- You've seen a couple of graph algorithms
    - Topological sort
    - Unweighted shortest path (using BFS)
- Now that you've seen BFS and DFS, we can discuss several more well-known graph problems
    - Dijkstra's algorithm (weighted shortest path)
    - Minimum spanning trees
        - Prim's algorithm, Kruskal's algorithm
    - Hamiltonian circuit

Dijkstra's Algorithm

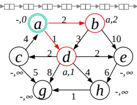
- BFS shortest path doesn't work on weighted graphs
    - Shortest weighted path may have more hops
- Recall how BFS shortest path worked
    - Each node has a path length associated
    - Nodes to be expanded have shortest length seen so far
- Dijkstra's algorithm is similar; we always expand the best node seen so far
- Unlike BFS, we may update nodes we've already visited, if we find a better path to that node

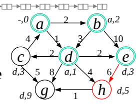
Weighted Shortest Path Applications

- Cheapest multiple-hop airline schedule
    - BFS would give minimum number of hops
- Email routing
    - Vertices are computers, edges are network links
    - Find routing path with smallest total link delay
- Shipping costs via multiple carriers
- etc etc etc

Dijkstra Example (paths from *a*)

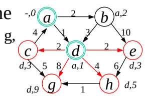
- a is done
  - update b, d

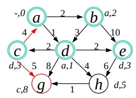




- e is next (tie with c)
  - no update

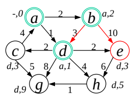
- d has shortest path; it's done
  - update c, e, g, h

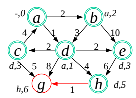




- c is next
  - update g

- b has shortest path; it's done
  - no updates





- h is next
  - update g
- finally, g
  - no update

Dijkstra pseudocode

- Only works if no edge has a negative cost

```
repeat until no undone nodes
  v = undone node with shortest path
  v.done = TRUE
  for each node w adjacent to v
     if v.dist + (v,w).cost < w.dist
          update w.dist
          w.prev = v
```

- At this point, for any target node *x*, we can follow the 'prev' chain to compute the path from *a*

Dijkstra Analysis

- Outer loop executes |V| times; body has two parts:
  - Find smallest remaining—naive would be O(|V|) scan
  - Update neighbors—O(|E|) over whole algorithm
- So, $O(|E| + |V|^2)$, which is ≈ O(|E|) for dense graphs
- For sparse graphs, however, this is too slow
  - Use priority queue to find smallest: DeleteMin is O(1)
  - Update neighbor is a DecreaseKey: O(log |V|)
  - O(|E| log |V| + |V| log |V|) ≈ O(|V| log |V|) for sparse graphs
- Only works if no edges have negative costs
  - Algorithm to handle that mentioned in book, cost is O(|E| * |V|)!

Greedy Algorithms

- Dijkstra's algorithm is an example of a *greedy* algorithm
- Greedy algorithms always take the step that currently seems best
  - No consideration of long-term or global issues
  - Not always optimal or even correct
- Be happy if a greedy technique is applicable, because the shortsighted approach often results in a near-linear cost

Minimum Spanning Trees (MST)

- We saw that BFS and DFS will create a spanning tree for a (connected) graph
- However, there are many possible spanning trees
- Given an undirected, weighted graph, we want to construct the spanning tree with the minimum total edge cost
- Examples:
  - Oil pipelines between various cities
  - Cables run between outlets in a house

MST: Greed is Good

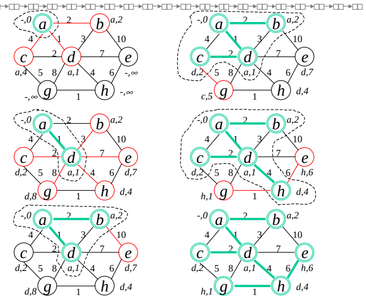
- An MST satisfies an important property:
  - Adding one more edge will create one cycle
    - Removing any other edge from that cycle makes it a tree again
- This suggests a greedy technique
  - Add an edge to the MST if you can remove a higher-cost edge from any cycle you create
- Two greedy algorithms are available to construct MSTs: Prim's and Kruskal's

MST: Prim's Algorithm

- Tree starts with any single node
- Add nodes to tree as follows:
  - Find lowest-cost edge (u,v) where u is in the tree and v is not
- Don't want to scan all edges
  - For each vertex not in tree, remember the cost and name of the nearest node in the tree, if any
  - After adding a node to tree, update all adjacent nodes that aren't in the tree
- Almost exactly like Dijkstra's algorithm

MST: Prim example

- Start with a
  - update b c d
- d closest (to a)
  - update c e g h
- b closest (to a)
  - no updates
- c closest (to d)
  - update g
- h closest (to d)
  - update e g
- g closest (to h)
- e last

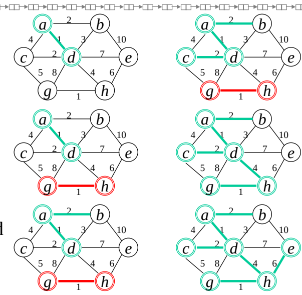


MST: Kruskal's Algorithm

- Prim added cheapest edge that was attached to the current tree
- Instead, just add cheapest edge anywhere, as long as it doesn't create a cycle
  - During construction, we'll have a forest, not a single tree
- How to tell if adding (u,v) creates a cycle?
  - Only if u and v are both in the same tree
- Note: no data tracked per node, unlike Prim/Dijkstra

MST: Kruskal example

- Start with a
- (a,d)==1
- (g,h)==1
- (a,b)==2
- (c,d)==2
- (b,d),(a,c) rejected
- (d,h)==4
- (c,g) rejected
- (e,h)==6

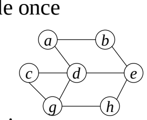


Implementing MST

- Prim is just like Dijkstra, so use a priority queue if the graph is not dense
  - O(|E| log |V|) for sparse, O(|V|^2) for dense
- Kruskal needs to detect whether two nodes are part of the same tree
  - This is a job for the union/find structure!
    - Each time edge (u,v) is tested, reject if Find(u)==Find(v)
    - Otherwise do a Union(Find(u),Find(v))
  - Still need a priority queue to get smallest edge
  - May have to try every edge, so O(|E| log |E|) = O(|E| log |V|)
    - In practice, not so bad

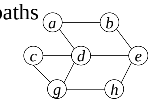
Hamiltonian Circuit

- A Hamiltonian circuit of a graph is a simple cycle (i.e. no repeats) that visits every node once
- Is there a Hamiltonian circuit?
  - In example, yes, d a b e h g c d
- There is no known algorithm to solve this problem in polynomial time
  - Not even a large polynomial, like N^{1000}



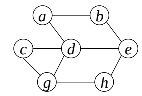
Solving Hamiltonian Circuit

- One way to do this is to try every possible path
- Recall that DFS marks nodes so they won't be visited more than once
  - e.g. once we visit adgc, we won't touch g again
  - so if we try that path, we won't find a Hamiltonian
- Modify DFS to unmark nodes when done with them, so they can be visited via other paths
  - e.g. we could do adgc, then abekg



Exhaustive Search

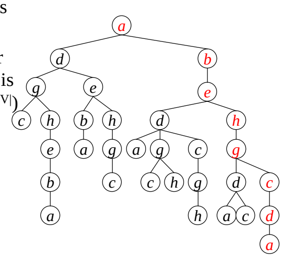
```
Exhaustive(v)
  if v is unvisited
     mark v as visited
     for each edge (v,w)
          Exhaustive(w)
  unmark v
```



- Sequence of paths tried might be:
  - adgc, adgheba, adeba, adehgc, abeda, abedgc, abedcgh, abehgda, abedgh, abehgcda, abehgdc
- How many possible paths are there?

Analyzing Exhaustive Search

- Can write all possible paths as a *search tree*
- If average branching factor (i.e. size of adjacency list) is B, number of paths is $O(B^{|V|})$
- Exponential time!



Exponential Time is Bad

log log N	log N	N	N log N	N^2	2^N
-	0	**1**	-	1	2
0	1	**2**	2	4	4
1	2	**4**	8	16	16
	3	**10**	30	100	1024
2	7	**100**	700	10,000	1,000,000,000,000,000,000,000,000,000,000
3	10	**1,000**	10,000	1,000,000	you've got to be kidding
4	20	**1,000,000**	20,000,000	1,000,000,000,000	300,000 zeroes!!
5	30	**1,000,000,000**	30,000,000,000	1,000,000,000,000,000,000	

P and NP

- Exponential is asymptotically worse than *any* polynomial function
- In CS theory, the set P contains all problems which can be solved in polynomial worst case time
- The set NP contains all problems for which a candidate solution can be verified in polynomial time
  - e.g. given a path, test whether it's a Hamiltonian circuit
  - includes all of P

P ≠ NP ??????

- It is believed that there are problems in NP which are not in P, i.e. that don't have a polynomial-time solution
- But no one is sure! This is one of the biggest, oldest unsolved problems in computer science
  - Fame awaits you if you can figure it out
- Faced with this dilemma, a lot of theory has grown around NP to at least give us some information

NP-completeness

- A subset of problems in NP (including the hardest ones) are known as NP-complete (NPC)
- Any problem in NP can be converted to an NP-complete problem in polynomial time
- So, if anyone ever finds a polynomial solution to just one NP-complete problem, they're all solved!
  - No one has, yet
- To show that a problem Q is NPC, prove that a known NPC problem can be converted to Q in polynomial time

NP-complete problems

- Many interesting problems are NP-complete
  - Hamiltonian circuit
  - Traveling Salesman: shortest Hamiltonian circuit
  - Boolean Satisfiability
  - Longest path
  - Integer Partition: are there 2 subsets with same sum?
  - Graph coloring: how many colors needed so no adjacent nodes have same color?
  - Clique: find largest subset of vertices which are completely connected to each other
- Plenty of others occur in all branches of CS, engineering, math, science

Living with NP-completeness

- Many techniques have been used to get around NPC
  - Dynamic programming
    - Avoid repeatedly solving the same subproblems
  - Very probable average case that is polynomial
    - Worst case still exponential, but make it very unlikely
  - Approximate solutions
    - Get an answer within some tolerance of optimum
  - Wimpy exponentials
    - 1.00001^N is tolerable up to N=1,000,000 or so

Who Cares?

- Although we won't do much in this class, it's important to know about NP-completeness
- You may find, as you design a program, that you have to solve a complex subproblem
  - If you are familiar with the NP-complete problems, you can make a good guess whether yours is NPC also
  - If it is, perhaps you should try another solution ☺