CSE 326: Data Structures
Lecture #19
More Fun with Graphs

Henry Kautz

Winter Quarter 2002

Today

- How to Make Depth-First Search Find Optimal Paths
  - Why bother?
- Finding Connected Components
  - Application to machine vision
- Finding Minimum Spanning Trees
  - Yet another use for union/find

Is BFS the Hands Down Winner?

Consider finding a path from vertex S to G in an unweighted graph where you do *not* have a heuristic function *h(n)*.

- Depth-first search
  - Simple to implement (implicit or explict stack)
  - Does not always find shortest paths
  - Must be careful to "mark" visited vertices, or you could go into an infinite loop if there is a cycle
- Breadth-first search
  - Simple to implement (queue)
  - Always finds shortest paths
  - Marking visited nodes can improve efficiency, but even without doing so search is guaranteed to terminate

Space Requirements

Consider space required by the stack or queue…

- Suppose
  - G is *known* to be at distance d from S
  - Each vertex n has k out-edges
  - There are no (undirected or directed) cycles
- BFS queue will grow to size k^d
  - Will simultaneously contain all nodes that are at distance d (once last vertex at distance d-1 is expanded)
  - For k=10, d=15, size is 1,000,000,000,000,000

DFS Space Requirements

- Consider DFS, where we limit the depth of the search to d
  - Force a backtrack at d+1
  - When visiting a node n at depth d, stack will contain
    - (at most) k-1 siblings of n
    - parent of n
    - siblings of parent of n
    - grandparent of n
    - siblings of grandparent of n …
- DFS queue grows at most to size dk
  - For k=10, d=15, size is 150
  - *Compare with BFS 1,000,000,000,000,000*

Conclusion

- For very large graphs – ones that are generated "on the fly" rather than stored entirely in memory – DFS is hugely more memory efficient, *if* we know the distance to the goal vertex!
- But suppose we don't know d. What is the (obvious) strategy?

Iterative Deepening DFS

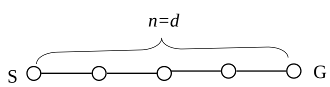
```
IterativeDeepeningDFS(vertex s, g){
  for (i=1;true;i++)
     if DFS(i, s, g) return;
}
// Also need to keep track of path found
bool DFS(int limit, vertex s, g){
  if (s==g) return true;
  if (limit-- <= 0) return false;
  for (n in children(s))
     if (DFS(limit, n, g)) return true;
  return false;
}
```

Analysis of Iterative Deepening

- Even without "marking" nodes as visited, iterative-deepening DFS never goes into an infinite loop
  - For very large graphs, memory cost of keeping track of visited vertices may make marking prohibitive
- Work performed with limit < actual distance to G is wasted – but the wasted work is usually small compared to amount of work done during the *last* iteration

Asymptotic Analysis

- There are "pathological" graphs for which iterative deepening is bad:



Iterative Deepening DFS =

$$1+(1+2)+(1+2+3+...)+... = \sum_{i=1}^{n}\sum_{j=1}^{i} j = O(n^2)$$

BFS = $O(n)$

A Better Case

Suppose each vertex n has k out-edges

- We don't worry about cycles – just search the vertices over again

- Exhaustive DFS to level i reaches k^i vertices – requires time ck^i for some constant c
- Iterative Deepening DFS(d) =

$$\sum_{i=1}^{d} k^i = O(k^d)$$ *ignore low order terms!*

BFS = $O(k^d)$

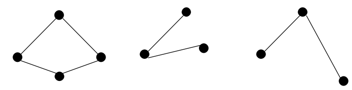
(More) Conclusions

- To find a shortest path between two nodes in a unweighted graph where no heuristic function is known, use either BFS or Iterated DFS
- If the graph is large, Iterated DFS typically uses much less memory
- If a good heuristic function is known, use A*
  - But what about memory requirements for A* for very large graphs??!!

(Final?) Conclusions & Questions

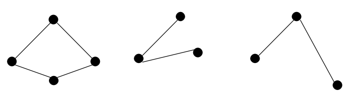
- In the worst case A* can also require a (priority) queue of size exponential in d, the distance to the goal vertex
- Question: Can one create an iterated, depth-first version of A* that (typically) uses less memory?
  - Yes, but you'll have to wait until you take CSE 473, Introduction to Artificial Intelligence to see it!
- Related Question: How can we adapt Iterated DFS for *weighted* graphs, in order to get an algorithm that is more memory efficient than Dijkstra's?

Counting Connected Components



Initialize the cost of each vertex to ∞
Num_cc = 0
While there are vertices of cost ∞ {
    Pick an arbitrary such vertex S, set its cost to 0
    Find paths from S
    Num_cc ++ }

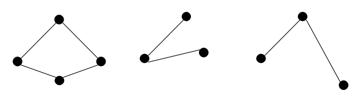
Using DFS



Set each vertex to “unvisited”
Num_cc = 0
While there are unvisited vertices {
    Pick an arbitrary such vertex S
    Perform DFS from S, marking vertices as visited
    Num_cc ++ }

Complexity = O(|V|+|E|)

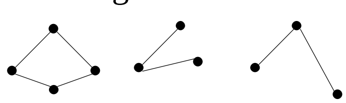
Using Union / Find



Put each node in its own equivalence class
Num_cc = 0
For each edge E = <x,y>
    Union(x,y)
Return number of equivalence classes

Complexity =

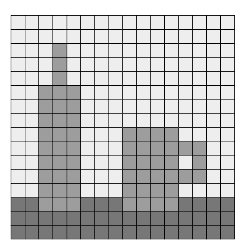
Using Union / Find



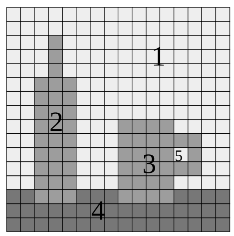
Put each node in its own equivalence class
Num_cc = 0
For each edge E = <x,y>
    Union(x,y)
Return number of equivalence classes

Complexity = O(|V|+|E| ack(|E|,|V|))

Machine Vision: Blob Finding



Machine Vision: Blob Finding



Blob Finding

- Matrix can be considered an efficient representation of a graph with a very regular structure
- Cell = vertex
- Adjacent cells of same color = edge between vertices
- Blob finding = finding connected components

Tradeoffs

- Both DFS and Union/Find approaches are (essentially) O(|E|+|V|) = O(|E|) for binary images
- For each component, DFS (“recursive labeling”) can move all over the image – entire image must be in main memory
- Better in practice: row-by-row processing
  - localizes accesses to memory
  - typically 1-2 orders of magnitude faster!

High-Level Blob-Labeling

- Scan through image left/right and top/bottom
- If a cell is same color as (connected to) cell to right or below, then union them
- Give the same blob number to cells in each equivalence class

Blob-Labeling Algorithm

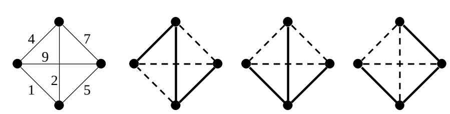
```
Put each cell <x,y> in it's own equivalence class
For each cell  <x,y>
        if color[x,y] == color[x+1,y] then
                Union( <x,y>,  <x+1,y> )
        if color[x,y] == color[x,y+1] then
                Union( <x,y>,  <x,y+1> )
label = 0
For each root <x,y>
        blobnum[x,y] = ++ label;
For each cell <x,y>
        blobnum[x,y] = blobnum( Find(<x,y>) )
```

Spanning Tree

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

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

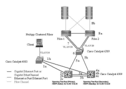
…forms a tree (is connected and contains no cycles)

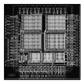


Minimum spanning tree: the spanning tree with the least total edge cost.

Applications of Minimal Spanning Trees

- Communication networks





- VLSI design
- Transportation systems

- Good approximation to some NP-hard problems (more later)

Kruskal's Algorithm for Minimum Spanning Trees

A greedy algorithm:

Initialize all vertices to unconnected

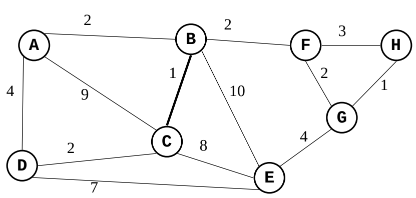
While there are still unmarked edges

Pick a lowest cost edge **`e = (u, v)`** and mark it

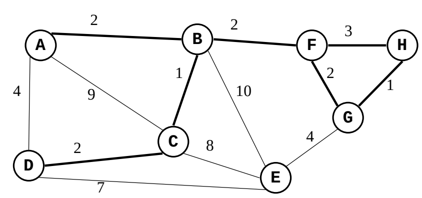
If **`u`** and **`v`** are not already connected, add **`e`** to the minimum spanning tree and connect **`u`** and **`v`**

Sound familiar?
(Think maze generation.)

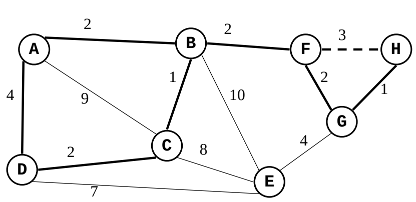
Kruskal's Algorithm in Action (1/5)



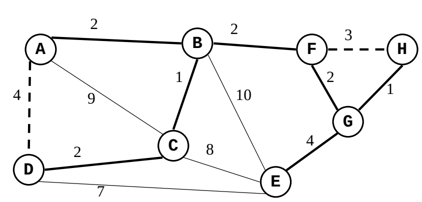
Kruskal's Algorithm in Action (2/5)



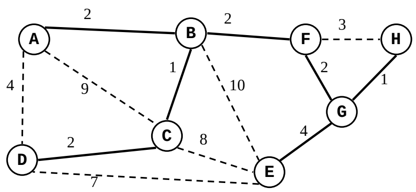
Kruskal's Algorithm in Action (3/5)



Kruskal's Algorithm in Action (4/5)



Kruskal's Algorithm Completed (5/5)



Why Greediness Works

Proof by contradiction that Kruskal's finds a minimum spanning tree:

- Assume another spanning tree has *lower cost* than Kruskal's.
- Pick an edge `e`$_1$ `= (u, v)` in that tree that's *not* in Kruskal's.
- Consider the point in Kruskal's algorithm where u's set and v's set were about to be connected. Kruskal selected some edge to connect them: call it `e`$_2$.
- But, `e`$_2$ must have at most the same cost as `e`$_1$ (otherwise Kruskal would have selected it instead).
- So, swap `e`$_2$ for `e`$_1$ (at worst keeping the cost the same)
- Repeat until the tree is identical to Kruskal's, where the cost is the same or lower than the original cost: **contradiction**!

Data Structures for Kruskal's Algorithm

|E| times:
Pick the lowest cost edge…

Once:
Initialize heap of edges…

buildHeap

findMin/deleteMin

|E| times:
If **u** and **v** are not already connected…
…connect **u** and **v**.

union

runtime: |E| + |E| log |E| + |E| ack(|E|,|V|)

Data Structures for Kruskal's Algorithm

|E| times:
Pick the lowest cost edge…

Once:
Initialize heap of edges…

buildHeap

findMin/deleteMin

|E| times:
If **u** and **v** are not already connected…
…connect **u** and **v**.

union

runtime: |E| + |E| log |E| + |E| ack(|E|,|V|) = O(|E|log|E|)

Prim's Algorithm

- Can also find Minimum Spanning Trees using a variation of Dijkstra's algorithm:

Pick a initial node

Until graph is connected:

Choose edge (u,v) which is of minimum cost among edges where u is in tree but v is not

Add (u,v) to the tree

- Same "greedy" proof, same asymptotic complexity

Does Greedy Always Work?

- Consider the following problem:

Given a graph G = (V,E) and a *designated subset* of vertices S, find a minimum cost tree that includes all of S

- Exactly the same as a minimum spanning tree, except that it does not have to include ALL the vertices – only the specified subset of vertices.
  - *Does Kruskal's or Prim's work?*

Nope!

- Greedy can fail to be optimal
  - because different solutions may contain different “non-designated” vertices, proof that you can covert one to the other doesn’t go through
- This Minimum Steiner Tree problem has *no* known solution of O(n^k) for any fixed k
  - NP-complete problems strike again!
  - Finding a spanning tree and then pruning it a pretty good approximation

Some other NP-Complete Problems

If you see one: approximate or search (maybe A*), and be prepared to wait…

- Traveling Salesman – given a *complete* weighted graph, find a minimum cost simple cycle of all the nodes.
- Graph Coloring – can each node in a graph be given a color from a set of k colors, such that no adjacent nodes receive the same color?

A Great Book You Should Own!

- *Computers and Intractability: A Guide to the Theory of NP-Completeness*, by Michael S. Garey and David S. Johnson