CSE 589 Part II

When you start on a long journey, trees are trees, water is water, and mountains are mountains. After you have gone some distance, trees are no longer trees, water no longer water, mountains no longer mountains. But after you have traveled a great distance, trees are once again trees, water is once again water, mountains are once again mountains.

-- Zen teaching

Readings

Dynamic programming
- Skiena, chapter 3
- CLR, chapter 16 (especially 16.1)

Graph algorithms
- Skiena, chapter 4
- CLR, chapters 23-26

Two basic paradigms

Break problem down into smaller, more easily solved pieces.

Divide and conquer

typically, split problem in half, solve each half, combine results to get full solution

Dynamic programming
- **characterize structure of an optimal solution**
- **recursively define value of an optimal solution**
- **compute the value of an optimal solution in bottom-up fashion**
- **construct an optimal solution from computed information**

Example DP Problem: Matrix Chain Multiplication

Given a sequence $A_1, A_2, \ldots, A_n$, where matrix A_i has dimension $p_{i-1} \times p_i$, fully parenthesize the product in a way that minimizes the number of scalar operations.

```
Matrix-Multiply(A,B)
  for i = 1 to rows [A]
    for j = 1 to columns [B]
      C[i,j] = 0;
      for k = 1 to columns [A]
        C[i,j] = C[i,j] + A[i,k]*B[k,j]
```

A pxq B qxr

result: C p xr

running time: pqr

It really does make a difference

A x B x C where
- A is 10 x 100
- B is 100 x 5
- C is 5 x 50

((A x B) x C)
- 10x100x5 + 10x5x50 = 7500

(A x (B x C))
- 10x100x50 + 100x5x50 = 75,000

Optimal parenthesization

How would you figure out the optimal parenthesization?

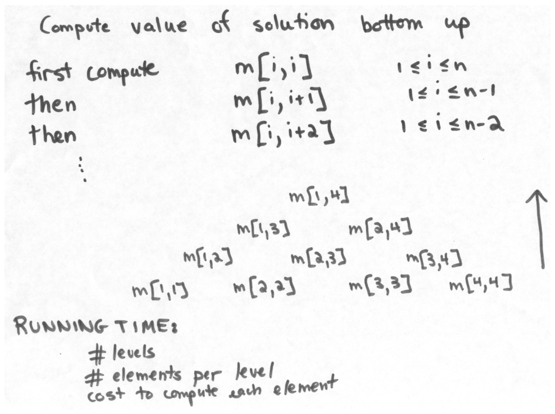
Dynamic Program for Matrix Chain Multiplication

m[i,j] = minimum number of scalar multiplications needed to compute $A_{i..j}$

Solution = m[1,n]

How to compute m[i,j]:

$\min_{i \leq k < j}\{ m[i,k] + m[k+1,j] + p_{i-1}p_kp_j\}$ $i < j$ (*)



Compute value of solution bottom up

first compute	m[i,i]	$1 \leq i \leq n$
then	m[i,i+1]	$1 \leq i \leq n-1$
then	m[i,i+2]	$1 \leq i \leq n-2$
⋮		

m[1,4]

m[1,3]   m[2,4]

m[1,2]   m[2,3]   m[3,4]

m[1,1]   m[2,2]   m[3,3]   m[4,4]

RUNNING TIME:

# levels
# elements per level
cost to compute each element

DP for Matrix Chain Multiplication

```
For i:= 1 to n do
  m[i,i] = 0;
For j := 1 to n
  for i := 1 to n
     compute m[i,i+j] using recurrence (*)
```

Remarks

- Optimal substructure within optimal solution is hallmark of applicability of dynamic programming
- property of overlapping subproblems is another hallmark of applicability of DP
- need to be careful about order in which subproblems are computed

Example: Approximate String Matching

P -- a pattern string

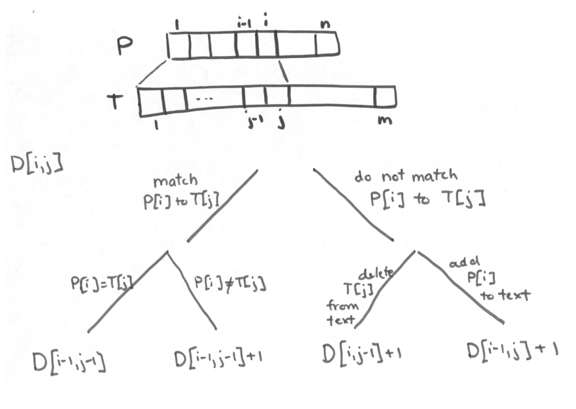
T -- a text string

Edit distance between P and T is smallest number of changes to transform T into P, where changes are:

- Substitution ....kat.... ----> ....cat....
- Insertion ....ct.... ----> ....cat....
- Deletion ....caat.... ----> ....cat....

Dynamic Program for AST

D[i,j] = edit distance between P[1..i] and segment of T ending at j.



DP

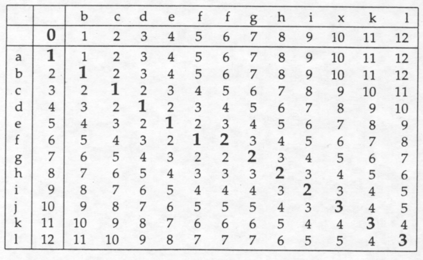
```
For i:= 0 to n do  D[i,0] = i
for j:= 0 to m do  D[0,j] = j;
For i := 1 to n
  for j := 1 to m
     D[i,j] = min( D[i-1,j-1] + matchcost(P[i],T[j]),
           D[i-1,j] + 1,  D[i,j-1] + 1)
```

T o recover actual alignment

Walk backwards through D[i,j] table starting at D[n,m].

At each cell, look at costs of three neighbors, to reconstruct choice made to get to goal cell.

Direction of backwards step determines if it was an insertion/deletion or match/substitution.



		b	c	d	e	f	f	g	h	i	x	k	l
	0	1	2	3	4	5	6	7	8	9	10	11	12
a	**1**	1	2	3	4	5	6	7	8	9	10	11	12
b	2	**1**	2	3	4	5	6	7	8	9	10	11	12
c	3	2	**1**	2	3	4	5	6	7	8	9	10	11
d	4	3	2	**1**	2	3	4	5	6	7	8	9	10
e	5	4	3	2	**1**	2	3	4	5	6	7	8	9
f	6	5	4	3	2	**1**	**2**	3	4	5	6	7	8
g	7	6	5	4	3	2	2	**2**	3	4	5	6	7
h	8	7	6	5	4	3	3	3	**2**	3	4	5	6
i	9	8	7	6	5	4	4	4	3	**2**	3	4	5
j	10	9	8	7	6	5	5	5	4	3	**3**	4	5
k	11	10	9	8	7	6	6	6	5	4	4	**3**	4
l	12	11	10	9	8	7	7	7	6	5	5	4	**3**

Figure 3–4. Example dynamic programming matrix for edit distance computation, with the optimal alignment path highlighted in bold

Dynamic Programming S ummary

1. Formulate answer as recurrence relation or recursive algorithm
2. Show that number of values of recurrence bounded by poly
3. Specify order of evaluation for recurrence so have partial results when you need them.

DP works particularly well for optimization problems with inherent left to right ordering among elements.

Resulting global optimum often much better than solution found with heuristics.

Once you understand it, usually easier to work out from scratch a DP solution than to look it up.

Dynamic Programming Problem S urface R econs truction

P= $\{p_0, p_1,\dots, p_m\}$;   Q=$\{q_0, q_1,\dots, q_n\}$
planar, parallel polygons in 3D

A tiling of P and Q is a set of triangles (tiles) of the form $\{p_i, p_{i+1}, q_j\}$ or $\{q_j, q_{j+1}, p_i\}$.

Cross edge: tile edge with vertex from each polygon

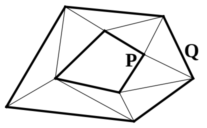
Tiling must satisfy 2 properties:

- every edge of each polygon is side of exactly one tile
- every cross edge is a side of 2 tiles

✓

Dynamic Programming Problem Surface Reconstruction

Problem: given P,Q, construct tiling of minimum surface area (the sum of the areas of all the triangles in the tiling)



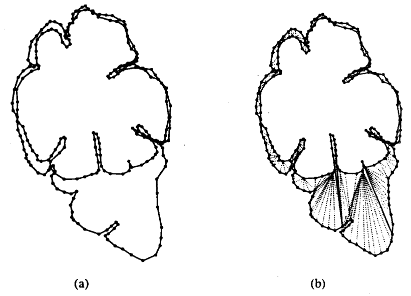


FIGURE 9.3 (a) A pair of contours obtained from the cerebral cortex of the human brain. The two con-

Graphs

A set of points ("vertices") and a set of lines ("edges") connecting pairs of points.

Examples:

- airline flight map
- communication networks
- precedence constraints on the scheduling of jobs
- flow networks

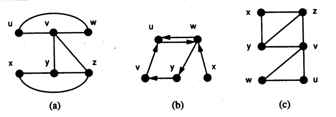


Figure 12.1 Three graphs. (a) and (c) are undirected graphs, (b) is a directed graph. The placement of the vertices on the paper is immaterial when we draw graphs; for example, (a) and (c) are in fact depictions of the same graph.

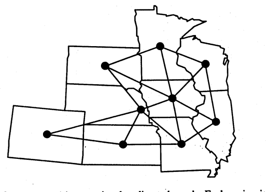


Figure 12.2 A map and its associated undirected graph. Each region is represented by a vertex, and an edge joins each pair of vertices that correspond to bordering regions.

Definitions

An **undirected graph** is a pair (V,E), where V is a finite set, and E is a set of unordered pairs (u,v), where u different from v and both are in V.

Terminology: If (u,v) is an edge (i.e., in E)

- u and v are **adjacent**; v is a **neighbor** of u

A **directed** graph is a pair (V,E), where V is a finite set, and E is a set of ordered pairs (u,v) (u different from v, both in V).

Representing graphs for algorithms

Adjacency matrix A[0..n-1, 0..n-1], where V={0,1,..., n-1}.

A[i,j] = 1 if (i,j) in E
0 otherwise

Adjacency lists:

L[0..n-1] array of lists where L[i] = {v | (i,v) in E}, the set of neighbors of i.

✔

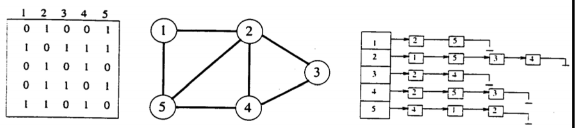


Figure 4–4. The adjacency matrix and adjacency list of a given graph

More Definitions:
Assume n = |V|, e = |E|

The **size** of the graph is n+e

- An algorithm running in time O(n+e) has just time to inspect each vertex and edge.

A **path** in *G* is a sequence (v_0,v_1,…,v_k) of vertices such that (v_i, v_{i+1}) in E, for all 0 <= i < k. Its **length** is k and it is a path from v_0 to v_k.

A **cycle** is a path such that $v_0 = v_k$.

An undirected graph is **connected** if and only if there is a path between every pair of vertices.

Tree (undirected)

An undirected graph which is:

- acyclic, i.e., has no cycles
- for every pair of vertices u,v, there is a unique, simple path from u to v
- deleting any edge yields a disconnected graph

Often orient the edges of the tree, so each vertex (except the root) has unique parent

More definitions

A **subgraph** of a graph *G*=(V,E) is a graph *G'*=(V',E') such that V' is contained in V and E' is contained in E.

A **connected component** of an undirected graph *G* is a maximal connected subgraph of *G*.

Graph Searching

Goal: want to traverse the edges of the graph and visit each vertex reachable from the starting vertex.

Two important traversals

- breadth-first search
- depth-first search

Can you determine if there is a path from v to w in linear time?

Breadth First Search (BFS)

```
BreadthFirstSearch (graph G, vertex v):
   Queue Q;  (initialized empty)

   for each vertex w do Encountered(w) = false;
   Encountered(v) = true;
   Q.Enqueue(v);
   while  !Q.IsEmptyQueue() do
      w := Q.Dequeue()
      Visit(w)
      for each neighbor w' of w do
          if !Encountered(w') then
              Encountered(w') := true;
              Q.Enqueue w')
```

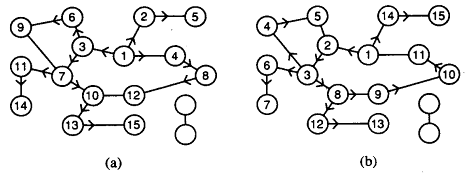


Figure 12.5 An undirected graph searched by (a) breadth-first and (b) depth-first search. In each case the search starts at the vertex labelled 1; the labels in the vertices show the order in which they are first encountered. (For breadth-first search this is also the order in which they are *Visit*ed. For depth-first search it is the order in which they are *PreVisit*ed; the *PostVisit* order is 5, 4, 7, 6, 11, 10, 9, 13, 12, 8, 3, 2, 15, 14, 1.) The arrows, which are not part of the graph, illustrate edges that were followed to previously unencountered vertices. Only the connected component containing the starting vertex is searched.

BFS Analysis

Running time:

- every vertex w is enqueued only once (because Encountered(w) is true thereafter) => only dequeued and visited once.
- Every edge is looked at tiwce.
- => total running time is

BFS Analysis

Running time:

- every vertex w is enqueued only once (because Encountered(w) is true thereafter) => only dequeued and visited once.
- Every edge is looked at twice.
- => total running time is O(n+e)

Consequences of BFS

Can determine if there is a path from v to w in O(n+e). How?

Consequences of BFS

Can determine if there is a path from v to w in O(n+e). How?

Run BFS(v).
If Encountered(w) is true, then there is a path. Otherwise, there isn't.

Consequences of BFS

Definition: The **distance** between two vertices v and w is the minimum length of any path from v to w, or infinite if there is no such path.

Can determine the distance from v to w in O(n+e). How?

Breadth First Search (BFS)

```
BreadthFirstSearch (graph G, vertex v):
   Queue Q;  (initialized empty)

   for each vertex w do
      Encountered(w) = false;  d(w) = infinity;
   Encountered(v) = true;  d(v) = 0;
   Q.Enqueue(v);
   while  !Q.IsEmptyQueue() do
      w := Q.Dequeue()
      Visit(w)
      for each neighbor w' of w do
          if !Encountered(w') then
              d(w') = d(w) + 1;
              Encountered(w') := true;
              Q.Enqueue w')
```

Spanning Trees

Definition: Let *G*=(V,E) be a connected, undirected graph. A **spanning tree** of *G* is a subgraph *G'* of *G* containing all the vertices of *G*, such that *G'* is a tree.

Can find a spanning tree in O(n+e). How?

Breadth First Search (BFS)

```
BreadthFirstSearch (graph G, vertex v):
   Queue Q;  (initialized empty)

   for each vertex w do Encountered(w) = false;
   Encountered(v) = true;
   Q.Enqueue(v);
   while  !Q.IsEmptyQueue() do
      w := Q.Dequeue()
      Visit(w)
      for each neighbor w' of w do
          if !Encountered(w') then
              Encountered(w') := true;
              Parent(w') = w;
              Q.Enqueue w')
```

Depth-First Search uses stack instead of queue

```
DepthFirstSearch (graph G, vertex v):
   for each vertex w in G do
        Encountered(w) = false;
   RecursiveDFS(v);

procedure RecursiveDFS(vertex v):
   Encountered(v) = true;
   PreVisit(v);
   for each neighbor w of v do
      if !Encountered(w) then
          Parent(w) = v;
          RecursiveDFS(w);
   PostVisit(v);
```

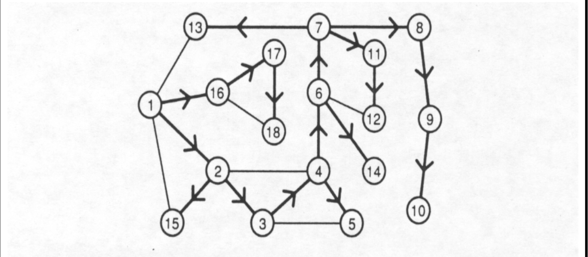


Figure 12.7 Depth-first search in an undirected graph. The search begins at the vertex labelled 1, and the vertices are labelled in the order they are encountered during the search. Followed edges are drawn with heavy lines and skipped edges with light lines; the arrows on the followed edges indicate the direction in which the edge was followed. When the skipped edges are deleted, the result is a tree; if vertex 1 is taken as the root of the tree then skipped edges join only ancestrally related vertices.

DFS Analysis

Running time: *O*(n+e)

- call Recursive DFS on each node exactly once (afterwards Encountered is true)
- each edge is examined twice (once from each endpoint)

assuming adjacency list representation of graph

✗

Consequences of DFS

Can determine if there is a path from v to w in O(n+e).
Can find a spanning tree of a connected graph in O(n+e).
Can be used to detect if graph is a tree or if it contains a cycle.
Can determine if the graph is connected.
Can find the connected components of the graph.

Most important property of DFS on undirected graphs

Every edge is either a tree edge or an edge between an ancestor and a descendent in the tree.

Consequences of DFS (cont.)

Definition: a **topological ordering (sort)** of the vertices of a directed graph is an order $v_1,\ldots, v_n$ such that there is no edge (v_i,v_j) of G with $j < i$.
Application: tasks to be performed sequentially, with precedence constraints: if the pair (T_i, T_j) is a constraint then task i must be performed before task j.
Goal: find ordering (schedule) s.t. all constraints obeyed.

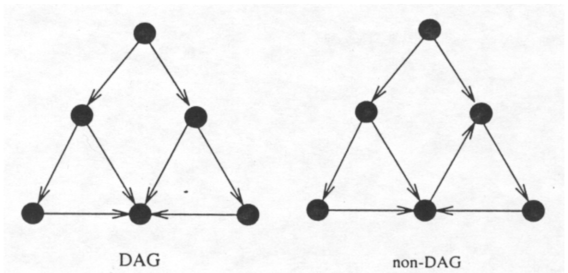


Figure 4–9. Directed acyclic and cyclic graphs

Using DFS to get topological sort:

modify main procedure to perform DFS from every vertex of G.

During PostVisit assign a number to each vertex (starting at n, and decrementing each time.)

Modify DFS to solve topological sort problem

```
DFSTopoSort (graph G):
   for each vertex v in G do
        Enc(v) = false;
   for each vertex v do
        if Enc(v) == false then RecursiveDFS(v);
   nextnumber := n;

procedure RecursiveDFS(vertex v):
   Encountered(v) = true;
   for each neighbor w of v do
      if !Encountered(w) then RecursiveDFS(w);
   Number(v) := nextnumber;
   nextnumber --;
```

✘

Consequences of DFS (cont.)

Lemma : **Suppose run DFSTopoSort on graph G which is a DAG. If** Number(v) < Number(w), then (w,v) not in E.

Proof:

Number(v) < Number(w) => RDFS(w) completes first.

At that time, either:

Enc(v) = false ----> no edge (w,v) ; search of w would have followed it.

Enc(v) == true ----> exploration of v in progress => path from v to w; edge (w,v) would imply cycle.

Other Applications of DFS

Finding articulation points in a graph.

Finding the strongly connected components of a directed graph.

Planarity testing.

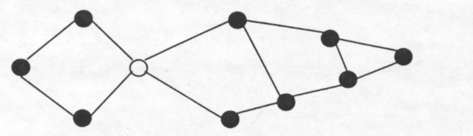


Figure 4–10. An articulation vertex is the weakest point in the graph

Single-Source Shortest Paths (Dijkstra's algorithm)

Using BFS, solved problem of finding shortest path from s to t.

What if edges have associated costs or distances? (BFS problem assumes edge costs are all 1.)

Assume each edge (u,v) has nonnegative cost c(u,v).

Define the **cost of a path** = total costs of all edges on path.

Problem: Find, for each vertex v, the least cost path from s to v.

Idea of Dijkstra's Algorithm:

Maintain:

- Dist [0..n-1] where Dist (v) is the cost of best path from s to v found so far, and
- U, set of vertices v for which Dist (v) is not yet known to be optimal.

Initially:

- Dist (s) = 0; Dist (v) = infinity for all v other than s.
- U = V.

In each step, remove that v in U with minimum Dist(v)

update those w in U s.t. (v,w) in E and

Dist(w) > Dist(w) + c(v,w).

Dijkstra's Algorithm

Assumption: c(u,v) = infinity if (u,v) not in E.

```
DijkstraShortestPaths (directed graph G, vertex s):
   Set U  (initialized to be empty)
   for each vertex v in G except s do
       Dist(v) := infinity
       U.insert(v);
   Dist(s) := 0;  U.insert(s);
   repeat |V| times
       v := any member of U with minimum Distance
       U.Delete(v);
       for each neighbor w of v do
           if U.contains(w) then
              Dist(w) := min(Dist(w), Dist(v) + c(v,w));
```

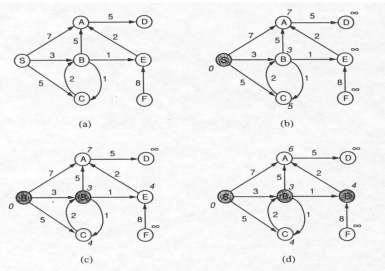


Figure 12.9 An example of Dijkstra's algorithm. (a) A directed graph, with a cost on each edge. We wish to find the least-cost path from S to each other vertex. (b) The distance to each vertex (except S) is tentatively set to ∞. (Tentative distances are shown in italics.) The vertex with least tentative distance is S, so it is removed from U, here depicted by shading the vertex. The tentative distance to each neighbor of S is updated. (c) Now vertex B has least tentative distance, and is removed from U. The tentative distance to C is updated from 5 to 4 and the tentative distance to E is updated from ∞ to 4. The distance via B to A is 8 which is greater than the tentative distance of A, so no update is necessary. (d) E is now removed from U (C could have been selected as well) and the distance to A is updated. The next vertex to be removed from U will be C (Problem 32).

Why is this algorithm correct?

Theorem: At the termination of the algorithm, Dist(v) is the length of the shortest path from s to v for each vertex v of G.

Proof: by induction on |V-U|.

Inductive hypothesis: Let |V-U|=k.

- for every v in V-U, Dist(v) is length of shortest path from s to v
- the vertices in V-U are the k closest vertices to s.
- for every v in U, Dist(v) is the length of shortest path from s to v that only goes through vertices in V-U.

At all times, for all v, Dist(v) is the length of shortest path from s to v that only goes through vertices in V-U

Base Case: U = V-{s}

Induction Step: Suppose true for first k steps. The SP to the (k+1)st closest vertex, say w, can go through only vertices in V-U, otherwise, there would be a closer vertex. Therefore, selecting the min => add the k+1st.

Say w is added.

New Dist value for a vertex x is min of old Dist value and Dist(w) + c(w,x)

Run Time Analysis

Underlying data structure: priority queue

Stores set S such that there is a linear order on key values.

Supports operations:

- Insert(x) -- insert element with key value x into set.
- FindMin() -- return value of smallest element in set
- DeleteMin() -- delete smallest element in set.

and usually:

- Lookup(x)
- Delete(x)

Most priority queue implementations

Can implement all these operations in O(log n) time for sets of size n.

=> Running time of Dijkstra's algorithm:

insertions:

deleteMins:

lookups:

modifying Dist values:

Running time of Dijkstra's algorithm:

n insertions:	O(n log n) time (actually O(n))
n deleteMins:	O(n log n) time
e lookups:	O(e log n) time (actually O(e))
e Dist value mods:	O(e log n) time

Running time: O((n + e) log n))

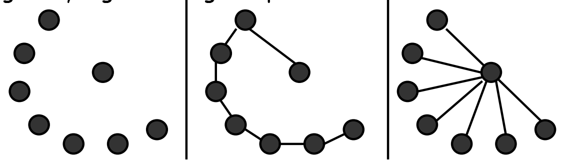
Can also do O(n^2). Better for dense graphs

Minimum Spanning Trees

Spanning tree of G -- subgraph of G that has same set of vertices as G and is a tree.

MST of weighted graph G -- spanning tree of G whose edges sum to minimum weight

greedy algorithms give optimal solution



Kruskal's MST algorithm

Assume G is connected.

Idea:

- Sort edges in increasing order of cost
- Initialize tree T to be empty
- For each edge e in sorted order
  - If T U {e} does not contain a cycle, add e to T;
- output T;

Correctness

Theorem: Kruskal's algorithm outputs an MST

Proof by induction

that at all times T is a subset of an MST F.

Let e be the next edge chosen by Kruskal.

Claim that T U{e} is a subset of an MST.

If e not in F, let e' in F connect two components of T.

Then F - {e'} + {e} is an MST.

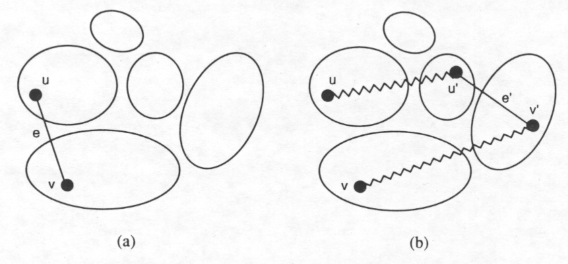


Figure 12.8 Illustration for the correctness proof of Kruskal's algorithm. (a) The graph $\langle V, E'\rangle$ and the new edge e. Each oval represents a connected component of $\langle V, E'\rangle$. (b) The path in F from u to v, which must contain an edge e' between distinct connected components of $\langle V, E'\rangle$. The set F' contains all of F except for e', and since it also contains e there is a path P' in F' from u' to v'.

To implement, use Disjoint Set ADT

Problem: maintain information about a collection of sets $S_1,\ldots,S_k$ that is dynamically changing through merges (unions)

Operations

- MakeSet (x): create new singleton set consisting of x
- Union (S,T): replace sets S, T by S U T
- Find(x): return "name" of set S such that x is in S.

Kruskal's Algorithm with Disjoint Set ADT

```
KruskalMST (undirected graph G):  set of edges in MST
   Set  T;  (initialized empty)  // set of edges in tree
   components := n; // number of connected comps of (G,E')
   Sort edges
   for each vertex v of G do MakeSet(v);
   while  components > 1 do  // process next edge in order
       (u,w) := next edge in sorted order;
        U := Find (u);
        W := Find (w);
        if (U different from W) then
            Union(U,W);
            Insert((u,w), T);
            components := components - 1;
   return T;
```

Best result

Theorem (Tarjan, 1975): There is an implementation of the Disjoint Set ADT such that the worst case time to perform any sequence of n-1 weighted Unions and m >= n Finds is Theta(m a(m,n))

$\alpha(m,n)$ is the inverse Ackermann function

$\alpha(m,n) <= 4$ for all $n < 2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}$ (17 times)

Run-time of Kruskal’s Algorithm

Sorting:
Find:
Unions:

Run-time of Kruskal’s Algorithm

Sorting: O(e log e)
# finds: e
# unions: n-1

Grand total: $O(e \log e + e\,\alpha(e,n))$

Modeling the Problem

An art, but key to applying algorithm design techniques to real-world problems.

Most algorithms designed to work on rigorously defined abstract structure.

Your turn!!!!

To try your hand at formulating problems as graph problems, and, if possible, using one of the algorithms we have discussed to solve them.

Problems

1. Speech recognition problem -- distinguishing between words that sound alike (e.g. to, two, too). You’re given a list of words and the likelihood of transitions between them. For each word spoken into the microphone, your program determines a set of words that sound like it. How might you figure out what sentence was spoken?
2. Given a set of processes that need to be scheduled, and dependencies that tell you which processes need to be completed in order for a given process to execute, how would you schedule the processes so as not to violate any of the dependencies?
3. How would you find your way out of a maze?

Problems

4. Network reliability -- how would you determine if there is a single point of failure in your network?
5. In an optical character recognition system, a method for separating lines of text is needed. Although there is some white space between the lines, problems like noise and the tilt of the page make the separation difficult to find. How would you go about doing line segmentation?
6. DNA sequences -- given experimental data consisting of small fragments. For each fragment f, know that certain other fragments are forced to lie to the left of f, certain fragments forced to lie to right, and rest either way. How do you find a consistent ordering of the fragments from left to right that satisfies all the constraints?

Problems

7. Multicast -- How would you decide what routes to use to send a message to a subset of the processors in your network? You can assume that you have information about the bandwidth and congestion on the various links in the network.
8. How would you design natural routes for a video-game character to follow through an obstacle-filled room?
9. How might you test if a certain set of conditions in your program can cause deadlock? For every pair of processes, A and B, it is possible to determine if A might ever wait for B.
10. Circuit simulation -- Given a circuit layout and input/output information for each gate in the circuit, (where its outputs go and where its inputs come from) how would you schedule the simulation of the gates in the circuit?