Mat 3770
Week 11

Spring 2014

Homework

Due date	Tucker	Rosen
4/7	4.1	9.6, set 1
4/7	Dijkstra–I	worksheet
4/9		9.6, set 2
4/9	Dijkstra–II	worksheet
4/11	3.4	10.5
4/11	TSP	worksheet

Week 11 — More Student Responsibilities

- Reading, Tucker: 4.1, 3.4
- Reading, Rosen: 9.6, (653–655)
- Attendance **Spring-i-ly** Encouraged

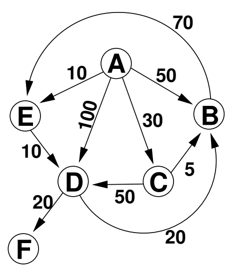
Single Source – Shortest Paths

- **Problem Statement**:

  Given a direct graph $G = (V, E)$, with edge weights and a vertex $v \in V$, find the weight of a shortest path from v to every other vertex in G.

- **Assumptions**:
  1. Weights are positive real numbers ($w : E_G \rightarrow \Re^+$)
  2. Path weights = sum of weights of all edges in the path
  3. Define *SHORT*(w) to be the weight of the shortest path from v to w.

An Example



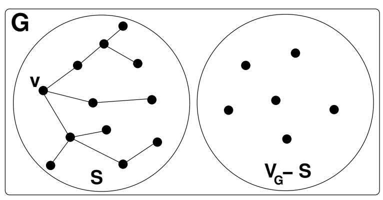
Graph G

Let v = A

SHORT (A) = 0
SHORT (B) = 35
SHORT (C) = 30
SHORT (D) = 20
SHORT (E) = 10
SHORT (F) = 40

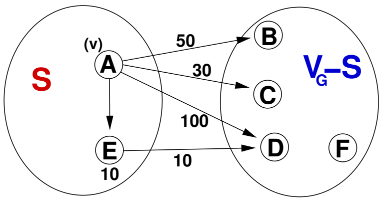
Idea Behind the Algorithm

Grow a *tree* of shortest paths
from v to other vertices in V_G.



Graph G

Let S $\subseteq$ V_G, v $\in$ S.
Suppose for each w $\in$ **S**, we know SHORT(w).



Graph G

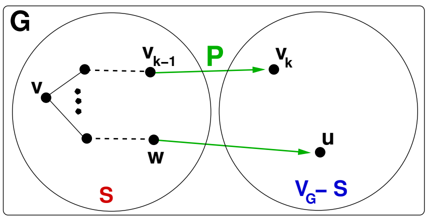
For each (w, u) $\in$ E_g, where **w** $\in$ **S**, **u** $\in$ **V**$_G$ - **S**, look at **SHORT(w)** + **weight(w,u)**.

Claim:
If (w, u) is the edge **minimizing**
SHORT(w) + weight(w, u)
— over *all* **w** $\in$ **S** and **u** $\in$ **V** - **S** —
then
SHORT(u) = **SHORT(w)** + **weight(w, u)**.

In other words, pick vertex from **V** - **S** to add to **S** which has the least weight path.

Thus, we add u to S and (w, u) to the tree.

Proof: Consider (w, u)



Graph G

Let **P** = $\{v = v_1, v_2, \ldots, v_k\}$ be the least weight path from **v** to some vertex v_k $\in$ **V** - **S**.

- **Observation**: $\{v = v_1, v_2, \ldots, v_{k-1}\}$ are all in **S** !
  (Else there is a shorter path, ending before v_k).
- Hence

$$\text{weight}(\mathbf{P}) \;=\; \mathbf{SHORT}(v_{k-1}) + \mathbf{weight}(v_{k-1}, v_k)$$
$$\geq\; \mathbf{SHORT(w)} + \mathbf{weight(w,\ u)}$$

- Thus, [**the shortest path from v to w**] + [**edge(w, u)**] gives a path from **v** to **u** which is as short as any path from **v** to any **x** $\in$ **V** - **S**.

			SHORT(i)					
Step	S	V-S	A	B	C	D	E	F
Init	A	BCDEF	0	∞	∞	∞	∞	∞
1	AE	BCDF	0	∞	∞	∞	10	∞
2	AED	BCF	0	∞	∞	20	10	∞
3	AEDC	BF	0	∞	30	20	10	∞
4	AEDCB	F	0	35	30	20	10	∞
5	AEDCBF		0	35	30	20	10	40

We want to minimize *SHORT(w)* + weight(w, u).

Algorithm — First Pass

```
Dijkstra(G, v) : SHORT[1..n]
    // Initializations
    S ← v
    SHORT[v] ← 0
    FOR each w ≠ v do
        SHORT[w] ← ∞
    // Build tree of shortest paths
    While |S| < |V| do
        Find edge e = (w, u) such that w ∈ S, u ∈ V-S, and
            SHORT(w) + weight(w, u) is minimal
        SHORT(u) ← SHORT(w) + weight(w, u)
        S ← S ∪ {u}
```

Implementation Analysis

- What are the slow steps — i.e., what will bound our time?

  Find edge e = (w, u) such that w $\in$ S, u $\in$ V-S, and SHORT(w) + weight(w, u) is minimal

- Suppose we had a TABLE, D[1..n], for u $\notin$ S, where:

  D[u] = $min_{w \in S}$ { SHORT[w] + weight(w, u) }

- Then we could
  1. search the list D for best u and add u to S
  2. next we would update D, since D[u'] may have changed for some elements
- So, how long to *update* D?
  1. Well, how many elements need changed?
  2. At most, outdeg(u) vertices — the neighbors of u
- Thus, overall time updating D is: $\sum_{u \in V} \; outdeg(u) \; = \; |E|$
- and, the algorithm takes time:

$$O(|V| \; * \; |V| \; + \; |E|) \; = \; O(|V|^2 \; + \; |E|)$$

or # iterations × Search D + Update D

Dijkstra(v) // v is the source node

variables:

D[ ] — the length of the shortest path from v to each node

P[ ] — the parent of each node

```
// Initialize cost to each node from source
//    and make source the parent
for all vertices u ∈ V
    if (v, u) ∈ E
        D[u] = weight(v, u)
    else
        D[u] = ∞
    P[u] = v
```



```
// Delete source from nodes not considered so far
//    and set distance to itself to 0
V' = V - { v }
D[v] = 0

// Determine shortest distances and parents
for i = 1 to n − 1
    Select u ∈ V' such that D[u] = min_{x∈V'} D[x]
    V' = V' − { u }
    for all w ∈ V'
        //Do we get shorter path to w through u?
        if D[w] > D[u] + weight(u, w)
            D[w] = D[u] + weight (u,w)
            P[w] = u
```


Obtaining Shortest Path Routes, source to sink

Idea: backtrack through parents until source is reached

```
for all nodes w ∈ V
    // print shortest path from w to v
    q = w
    print q
    while q ≠ v
        q = P[q]
        print q
    print q
```


Can We Do Better?

We always want to find the *minimum* in D

- If we implement D as a min–heap, Searching takes O(1) time
- **But**, how would we then update D?

  I.e., when u is added to S, if (u, u') $\in$ E, u' $\in$ V-S, we may need to change D[u']
- Solution: Add a **table**, Dindex[1..n] which stores the *position* of vertices in heap D

  Whenever heap elements are moved, update Dindex[].

- Now, updating D and Dindex takes

  outdeg(u) * $\log|V|$ steps

- So the algorithm takes

  $O((|V| + |E|) * \log|V|)$ time

- Is this better than $O(|V|^2 + |E|)$

- Yes, if $|E| \in O(\frac{|V|^2}{\log|V|})$

  For example, if G is planar or not "almost complete" (i.e., sparse).

The Traveling Salesperson Problem

- **Problem Statement**:

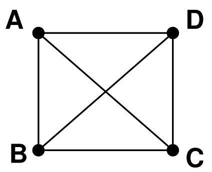
  Given a graph, find the least–cost circuit which includes every vertex — i.e., the cheapest Hamilton Circuit.

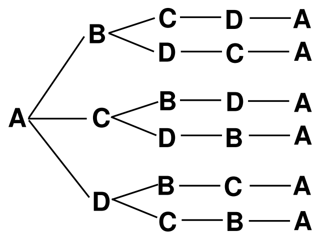
- **Multiple Uses**:
  1. Business and industry, ex: robotic motion planning for wiring or laser–drilled holes in circuit boards
  2. Efficient routing of telephone calls and Internet connections
  3. Archaeologists: help in determining the sequence of deposits
  4. Delivery routes

Several algorithms

1. Brute Force
2. Nearest Neighbor
3. Branch and Bound
4. Approximation Algorithm, Using Minimal Spanning Tree
5. And others. . .

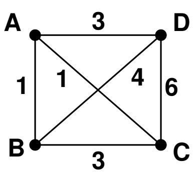
TSP: Brute Force





1. Select a starting point
2. Enumerate all possible Hamilton Circuits with that starting point
3. Find the total weight of each circuit
4. choose one which produces the minimum cost

Brute Force Example



Circuit	Total weight of the circuit
1. $A \to B \to C \to D \to A$	$3 + 6 + 3 + 1 = 13$
2. $A \to B \to D \to C \to A$	$3 + 4 + 3 + 1 = 11$
3. $A \to C \to B \to D \to A$	$1 + 6 + 4 + 1 = 12$
4. $A \to C \to D \to B \to A$ (opp 2)	11
5. $A \to D \to B \to C \to A$ (opp 3)	12
6. $A \to D \to C \to B \to A$ (opp 1)	13

Efficiency of Brute Force?

- The Brute Force method provides a way to find a minimum Hamilton circuit if one exists (optimal solution).

- Consider a complete, weighted graph with 10 vertices. How many circuits and total weights would we have to calculate for this graph? (10!, or 3,628,800 Hamilton circuits, but we'd only have to calculate the weights for half of them. . .)

- So, in worst case, if the graph had n vertices, how long would this algorithm take to execute? Is that feasible?

- If we used the Brute Force algorithm as the basis for a program for a super–computer, and had fed it a 100–vertex TSP when the universe was created, it would still be far from done. . . ($100! = 9.3326211544 \times 10^{157}$)

Approximation Algorithms

Algorithms which do not take too much computer time,
and that give **reasonably** good solutions
most of the time are called
approximation algorithms.

Nearest Neighbor Algorithm

Idea: choose as the next vertex, the **nearest** one to the current vertex. (Used on complete graphs.)

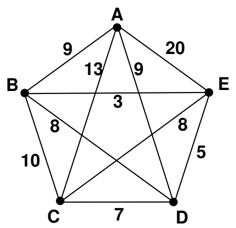
1. Choose a starting point for the circuit; say vertex `A`
2. Check all the edges incident to `A`, and choose one that has smallest weight; proceed along this edge to the next vertex
3. Repeat

   At each vertex you reach, check the edges from there to ***vertices not yet visited****. Choose one with smallest weight. Proceed along this edge to the next vertex.*

   until all vertices have been reached
4. Return to the starting vertex

Nearest Neighbor Example

A courier is based at the head office (A), and must deliver documents to four other offices (B, C, D, and E). The estimated time of travel in minutes between each of these offices is shown on the graph below.



Use the Nearest Neighbor algorithm to find an approximate solution to this problem. Calculate the total time required to cover the chosen route. Begin at A.

Answer:

Total Weight:

Is this the optimum answer?

Starting Vertex	Circuit Using Nearest Neighbor	Total Weight
`A`	$A \to D \to E \to B \to C \to A$	$9 + 5 + 3 + 10 + 13 = 40$
`B`	$B \to E \to D \to C \to A \to B$	$3 + 5 + 7 + 13 + 9 = 37$
`C`	$C \to D \to E \to B \to A \to C$	$7 + 5 + 3 + 9 + 13 = 37$
`D`	$D \to E \to B \to A \to C \to D$	$5 + 3 + 9 + 13 + 7 = 37$
`E`	$E \to B \to D \to C \to A \to E$	$3 + 8 + 7 + 13 + 20 = 51$

- Any circuit of minimum weight may be used.
- Since these are circuits, may begin at any office.
- Have we found a **minimum** Hamilton Circuit?

  Optimality is not guaranteed with Approximation Algorithms.

The Branch and Bound Algorithm

A general and usually inefficient method for solving optimization problems.

Goal

Find the **best** (i.e., optimal) solution to a problem.
Used for optimization problems and Artificial Intelligence

Branch and Bound Ideas

- Examine all feasible solutions in an orderly manner: the configurations of solutions (the **Search Space**) can be stored or represented in a tree structure.
- Search the tree using a depth–first approach: the traversal of one downward path in the tree produces a (possibly **infeasible**) solution.
- Eliminate as many of the feasible and infeasible solutions as possible by **pruning** branches from the search tree.
- The aim is to avoid traversing the entire tree by stopping searches at nodes (**pruning**) when it is ascertained an optimal solution cannot be represented by the current path.

Branch and Bound Ideas, Continued

- We **bound** our solutions if possible so we can **prune** some of the branches.
- We may be able to use both **upper** and **lower** bounds to aid in pruning.
- Generally, it is hard to claim anything about the running time of Branch and Bound algorithms.

Branch and Bound Overview

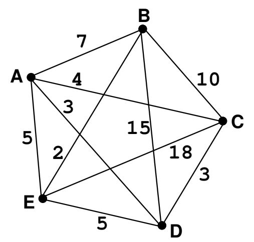
- Keep track of **active** nodes
- Choose the best value from among active nodes to determine the next node to consider
- If the value of an active node is too large to lead to an optimal solution, or too small to lead to a feasible solution, **prune** at that node

Common Assumptions

- Costs are symmetric: it costs the same to get from A to B as it does to get from B to A.
- Costs satisfy the triangle inequality: the cost to get from A to C is less than or equal to the cost if one detours through B while traveling from A to C.

Quick TSP Tour Construction

- Pick any node as a starting circuit consisting of one node
- While there exists a node not yet in the circuit
  - Find nodes *u* and *v* such that
    - *u* is not in the circuit
    - *v* is in the circuit
    - *cost*(*u*, *v*) is minimized
  - Insert *u* immediately in front of *v* in the circuit




```
A  B  C  D  E  A =

A  C  D  E  B  A =
```

Using Branch & Bound