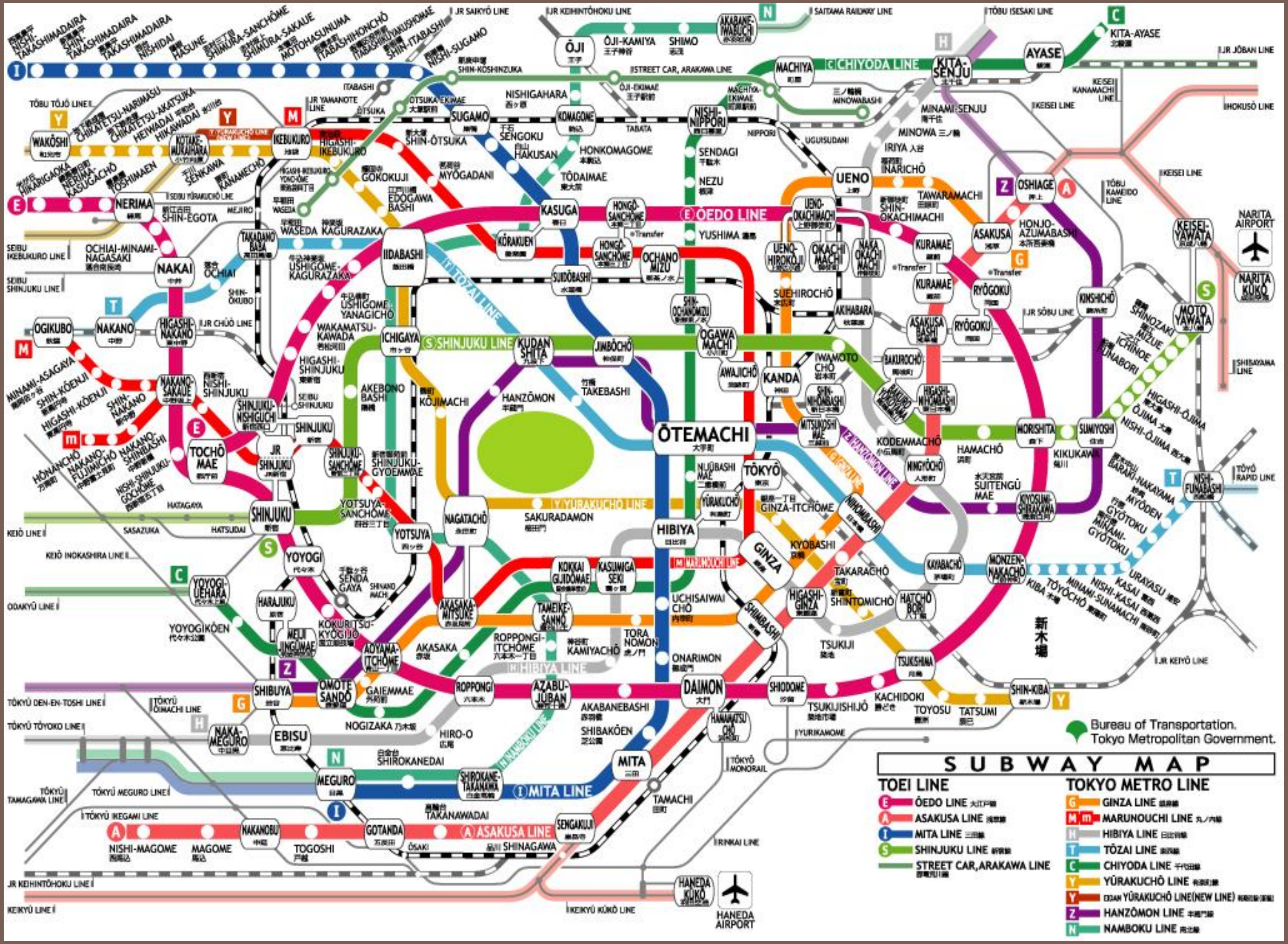


MORE GRAPHS

Lecture 19

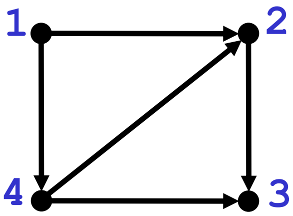
CS2110 – Fall 2013

Readings?

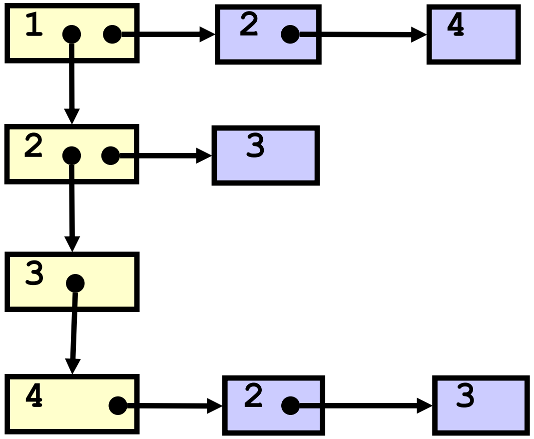
- This lecture is based on chapter 28

- Homework: (a simple self-test question): Suppose you were doing your own version of Google maps. You are writing code that tells the user how to get from Ithaca to Miami South Beach. Would you start by running Dijkstra's, Prim's, or Kruskel's algorithm?

Representations of Graphs



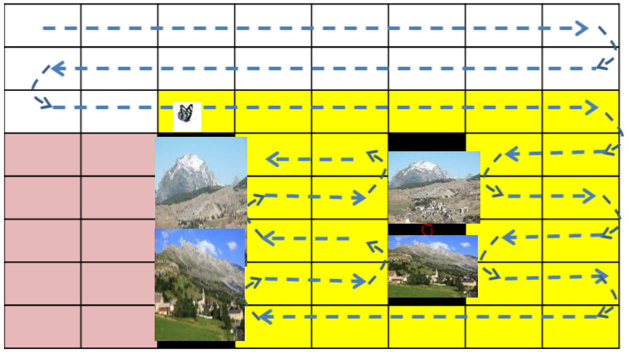
List



Matrix

	1	2	3	4
1	0	1	0	1
2	0	0	1	0
3	0	0	0	0
4	0	1	1	0

Danaus Park



Adjacency Matrix or Adjacency List or “Park”?

- Danaus is a kind of graph
  - In A3 and A5 we’ve simply captured it into a 2-D array
  - What graph would Danaus look like if you instead wanted to draw a picture of it as a graph?
    - Each tile would be a node
    - Each single move in a flyable path would be an edge
    - Edges present if you can get from [x][y] to [x’][y’]

  - Should the edges be weighted?
    - In A6 wind effects might argue for a weighted graph!

Representing *one* thing *two* ways

- In computer science we often build and use multiple representations of the same data
- For A5 this isn't really necessary, but in A6 (coming soon!) you'll need to work with both explicit graph representations of the park and with the 2-D form in order to have a high quality solution
  - For a lower quality solution this won't be needed
  - Best solutions might be 100x or more faster…

Shortest Paths in Graphs

- Finding the shortest (min-cost) path in a graph is a problem that occurs often
  - Find the shortest route between Ithaca and West Lafayette, IN
  - Result depends on our notion of cost
    - Least mileage… or least time… or cheapest
    - Perhaps, expends the least power in the butterfly while flying fastest
    - Many “costs” can be represented as edge weights
  - A butterfly that optimizes to fly in bright sunshine, or to most efficiently collect a list of flowers, is optimizing over possible path lengths that are computed using one or perhaps multiple such factors: *machine learning*
- How do we find a shortest path?

Dijkstra's shortest-path algorithm

Edsger Dijkstra, in an interview in 2010 (*CACM*):

… the algorithm for the shortest path, which I designed in about 20 minutes. One morning I was shopping in Amsterdam with my young fiance, and tired, we sat down on the cafe terrace to drink a cup of coffee, and I was just thinking about whether I could do this, and I then designed the algorithm for the shortest path. As I said, it was a 20-minute invention. [Took place in 1956]

Dijkstra, E.W. A note on two problems in Connexion with graphs. *Numerische Mathematik* 1, 269–271 (1959).

Visit http://www.dijkstrascry.com for all sorts of information on Dijkstra and his contributions. As a historical record, this is a gold mine.

Dijkstra's shortest-path algorithm

Dijsktra describes the algorithm in English:

□When he designed it in 1956, most people were programming in assembly language!

□Only *one* high-level language: Fortran, developed by John Backus at IBM and not quite finished.

No theory of order-of-execution time —topic yet to be developed. In paper, Dijsktra says, "my solution is preferred to another one … "the amount of work to be done seems considerably less."

Dijkstra, E.W. A note on two problems in Connexion with graphs. *Numerische Mathematik* 1, 269–271 (1959).

Dijkstra's shortest path algorithm

The n (> 0) nodes of a graph numbered 0..n-1.

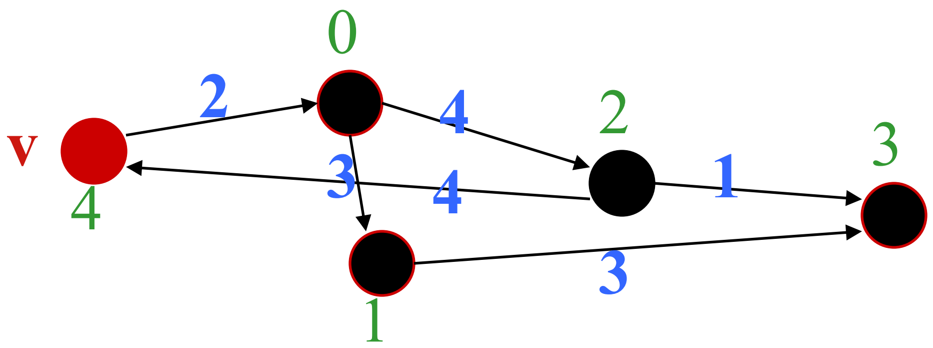
Each edge has a positive weight.

weight(v1, v2) is the weight of the edge from node v1 to v2.

Some node v be selected as the *start* node.

Calculate length of shortest path from v to each node.

Use an array L[0..n-1]: for **each** node w, store in L[w] the length of the shortest path from v to w.



L[0] = 2
L[1] = 5
L[2] = 6
L[3] = 7
L[4] = 0

Dijkstra's shortest path algorithm

Develop algorithm, not just present it.

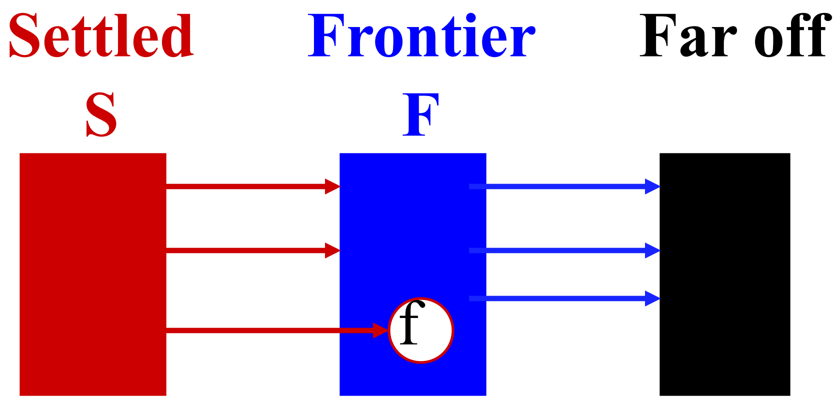
Need to show you the state of affairs —the relation among all variables— just before each node i is given its final value L[i].

This relation among the variables is an *invariant*, because it is always true.

Because each node i (except the first) is given its final value L[i] during an iteration of a loop, the *invariant* is called a *loop invariant*.

L[0] = 2
L[1] = 5
L[2] = 6
L[3] = 7
L[4] = 0

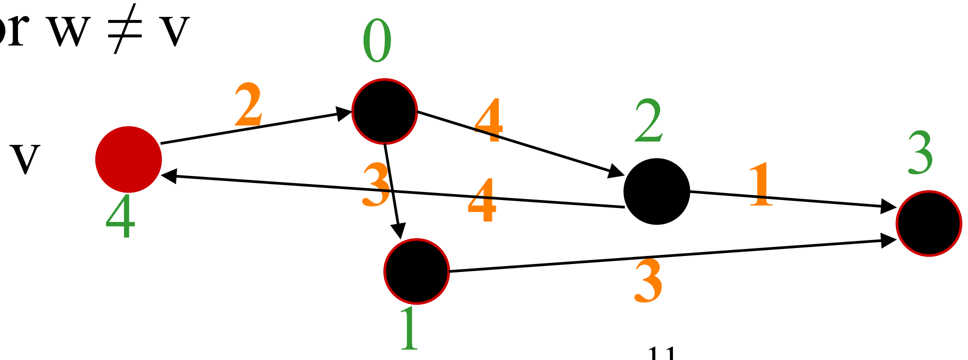
The loop invariant



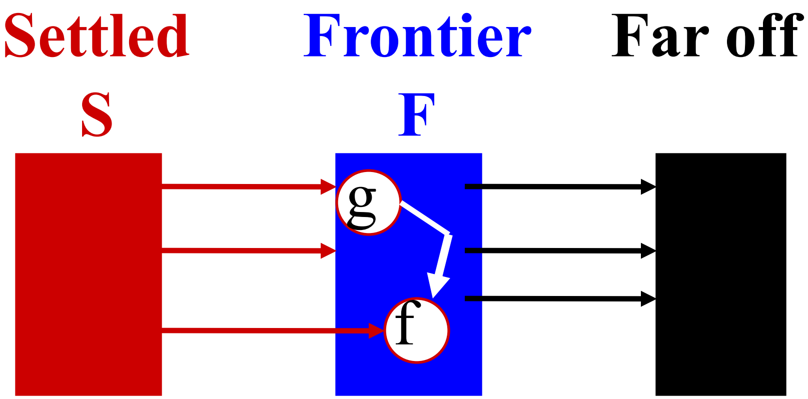
(edges leaving the black set and edges from the blue to the red set are not shown)

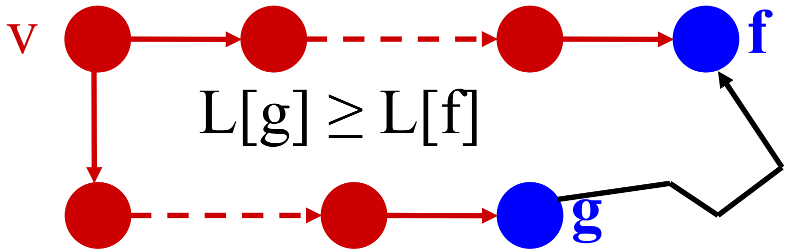
1. **For a Settled node s, L[s]** is length of shortest v → s path.
2. **All edges leaving S go to F.**
3. **For a Frontier node f, L[f] is length of shortest v → f path using only red nodes (except for f)**

4. **For a Far-off node b, L[b] = ∞**
5. L[v] = 0, L[w] > 0 for w ≠ v



Theorem about the invariant





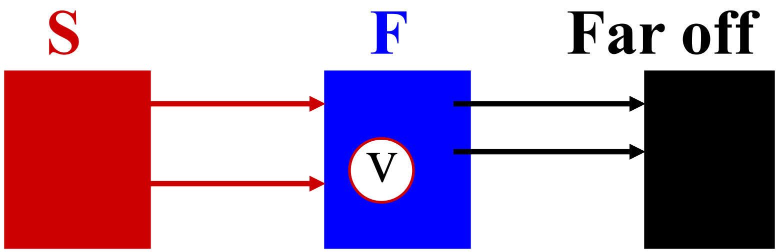
1. **For a Settled node s**, **L[s]** is length of shortest v → r path.
2. **All edges leaving S go to F.**
3. **For a Frontier node f, L[f]** is length of shortest v → f path using only Settled nodes (except for f).
4. **For a Far-off node b, L[b] = ∞.** **5.** L[v] = 0, L[w] > 0 for w ≠ v

Theorem. For a node **f** in **F** with minimum L value (over nodes in **F**), **L[f]** is the length of the shortest path from **v** to **f**.

Case 1: **v** is in **S**.

Case 2: **v** is in **F**. Note that L[v] is 0; it has minimum L value

The algorithm



1. **For s, L[s]** is length of shortest v→ s path.
2. **Edges leaving S go to F**.
3. **For f, L[f]** is length of shortest v → f path using red nodes (except for f).
4. **For b in Far off, L[b] = ∞**
5. L[v] = 0, L[w] > 0 for w ≠ v

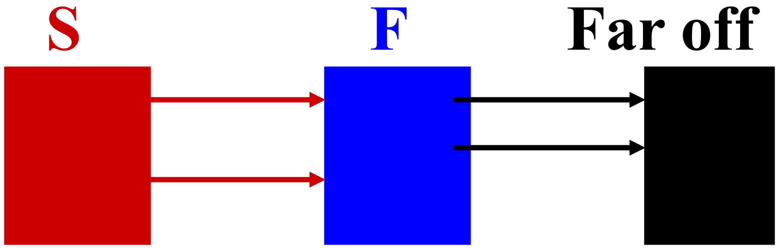
Theorem: For a node **f** in **F** with min L value, L[f] is shortest path length

For all w, L[w]= ∞; L[v]= 0;
F= { v }; S= { };

Loopy question 1:

How does the loop start? What is done to truthify the invariant?

The algorithm



1. **For s, L[s]** is length of shortest v → s path.
2. **Edges leaving S go to F**.
3. **For f, L[f]** is length of shortest v → f path using red nodes (except for f).
4. **For b in Far off, L[b] = ∞**
5. L[v] = 0, L[w] > 0 for w ≠ v

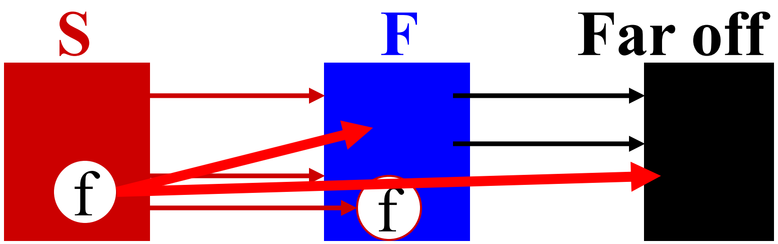
Theorem: For a node **f** in **F** with min L value, L[f] is shortest path length

```
For all w, L[w]= ∞;   L[v]= 0;
F= { v };  S= { };
while   F ≠ {}  {







}
```

Loopy question 2:

When does loop stop? When is array L completely calculated?

The algorithm



1. **For s, L[s]** is length of shortest v → s path.
2. **Edges leaving S go to F**.
3. **For f, L[f]** is length of shortest v → f path using red nodes (except for f).
4. **For b, L[b] = ∞**
5. L[v] = 0, L[w] > 0 for w ≠ v

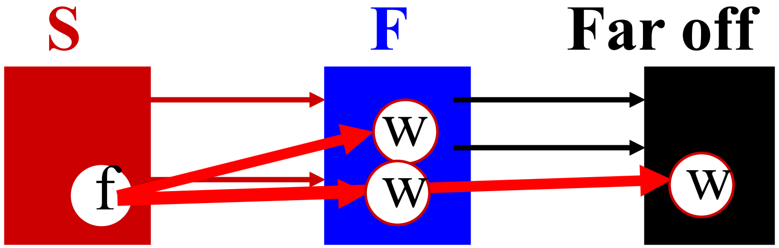
Theorem: For a node **f** in **F** with min L value, L[f] is shortest path length

```
For all w, L[w]= ∞;   L[v]= 0;
F= { v }; S= { };
while   F ≠ {}  {
    f= node in F with min L value;
    Remove f from F, add it to S;




}
```

Loopy question 3:

How is progress toward termination accomplished?

The algorithm



1. **For s, L[s]** is length of shortest v → s path.
2. **Edges leaving S go to F.**
3. **For f, L[f]** is length of shortest v → f path using red nodes (except for f).
4. **For b, L[b] = ∞**
5. L[v] = 0, L[w] > 0 for w ≠ v

Theorem: For a node **f** in **F** with min L value, L[f] is shortest path length

```
For all w, L[w]= ∞;   L[v]= 0;
F= { v }; S= { };
while  F ≠ {} {
    f= node in F with min L value;
    Remove f from F, add it to S;
    for each edge (f,w) {
      if (L[w] is ∞) add w to F;

      if (L[f] + weight (f,w) < L[w])
          L[w]= L[f] + weight(f,w);
    }
}
```

Algorithm is finished

Loopy question 4:

How is the invariant maintained?

About implementation

1. No need to implement **S**.
2. Implement **F** as a min-heap.
3. Instead of ∞, use
   Integer.MAX_VALUE.

```
For all w, L[w]= ∞;  L[v]= 0;
F= { v }; ~~S= { };~~
while F ≠ {} {
    f= node in F with min L value;
    Remove f from F, ~~add it to S;~~
    for each edge (f,w) {
        ~~if (L[w] is ∞) add w to F;~~
        ~~if (L[f] + weight (f,w) < L[w])~~
            ~~L[w]= L[f] + weight(f,w);~~
    }
}
```

```
if (L[w] == Integer.MAX_VAL) {
    L[w]= L[f] + weight(f,w);
    add w to F;
} else L[w]= Math.min(L[w],
                      L[f] + weight(f,w));
```

Execution time

n nodes, reachable from v. $e \geq n-1$ edges

$n-1 \leq e \leq n*n$

```
For all w, L[w]= ∞;  L[v]= 0;                  O(n)
F=  { v };                                      O(1)
while F ≠ {} {                                  O(n)
   f=  node in F with min L value;              O(n)
   Remove f from F;                             O(n log n)
   for each edge (f,w) {                        O(n + e)
      if (L[w] == Integer.MAX_VAL) {            O(e)
         L[w]=  L[f] + weight(f,w);             O(n-1)
         add w to F;                            O(n log n)
      }
      else L[w]=                                O((e-(n-1)) log n)
         Math.min(L[w], L[f] + weight(f,w));
   }
}
```

outer loop:
n iterations.
Condition
evaluated
n+1 times.

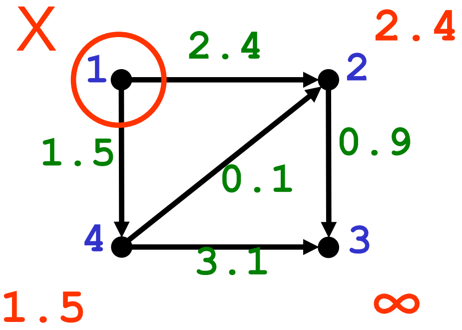
inner loop:
e iterations.
Condition
evaluated
n + e times.

Complete graph: $O(n^2 \log n)$. Sparse graph: $O(n \log n)$

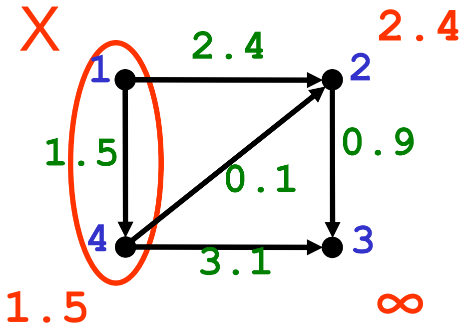
Dijkstra’s Algorithm

```
dijkstra(s) {
   // Note: weight(s,t) = cost of the s,t edge if present
   //                     Integer.MAX_VALUE otherwise

   D[s] = 0; D[t] = weight(s,t), t ≠ s;
   mark s;
   while (some vertices are unmarked) {
      v = unmarked node with smallest D;
      mark v;
      for (each w adjacent to v) {
         D[w] = min(D[w], D[v] + weight(v,w));
      }
   }
}
```

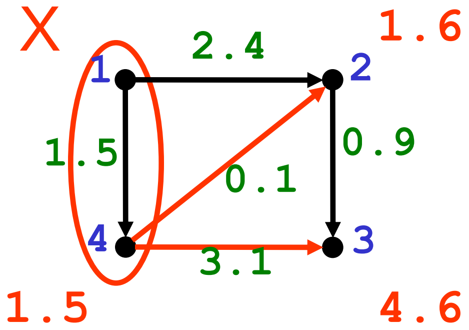
Dijkstra's Algorithm



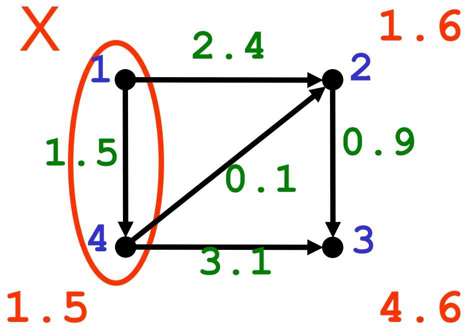
Dijkstra’s Algorithm



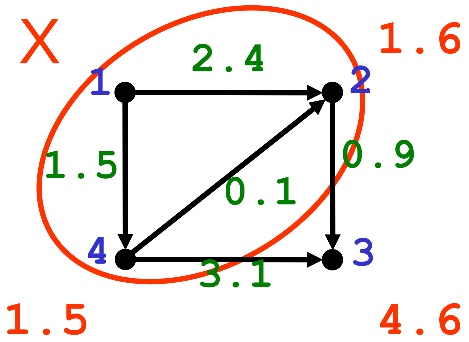
Dijkstra's Algorithm



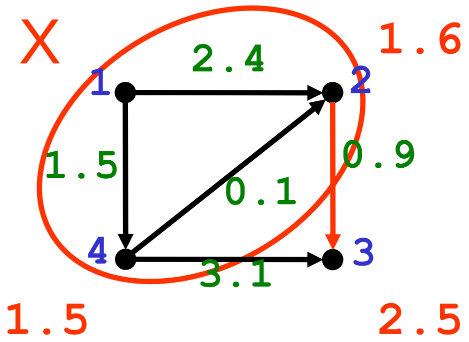
Dijkstra's Algorithm



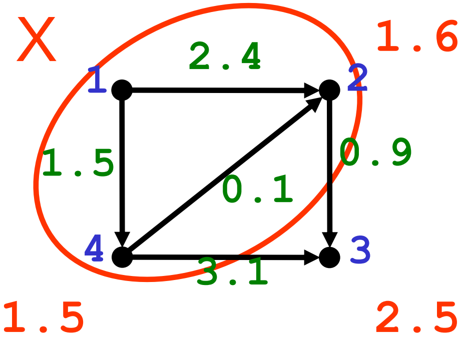
Dijkstra's Algorithm



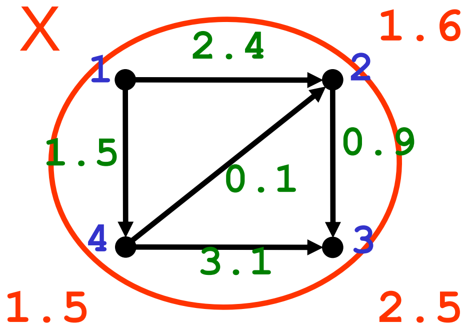
Dijkstra's Algorithm



Dijkstra's Algorithm



Dijkstra's Algorithm



Shortest Paths for Unweighted Graphs – A Special Case

- Use breadth-first search
- Time is O(n + m) in adj list representation, O(n^2) in adj matrix representation