CSCI 136
Data Structures & Advanced Programming

Jeannie Albrecht
Lecture 34
~~May 12, 2014~~
May 14, 2014

Administrative Details

- Final exam - self scheduled
  - You get 2.5 hours to complete it
  - Covers everything, with strong emphasis on Ch 14-16 (BSTs, HashTables, Maps, Graphs)
  - Study guide on handouts page
- Makeup class today 1:10-2 in Wege
- Lab 11 is also today (optional) from 2-4
- You'll get midterms and Lab 9 (Darwin) back this afternoon (I'll grade Lab 10 next week)
- Extra credit accepted through Tue, May 20 at 5pm
- I will be out of town this weekend

Last Time

- Briefly discussed DFS and BFS
- Darwin
  - Nice job everyone
  - Congrats to Riley!

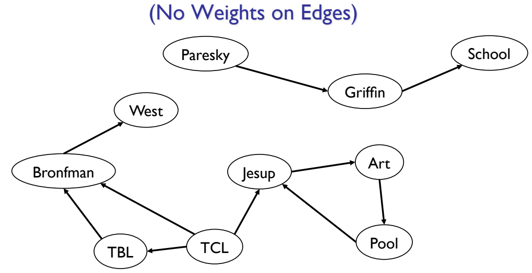
Today's Outline

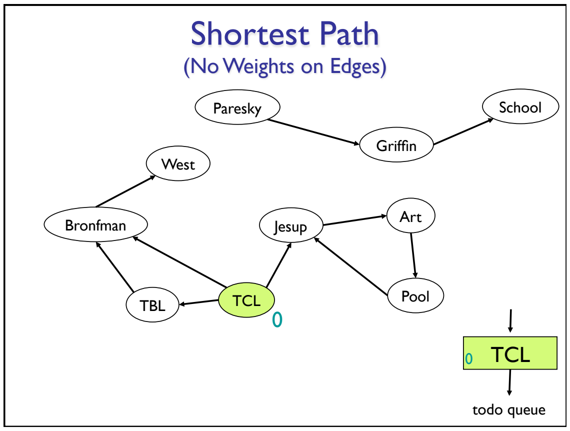
- Finish discussing graph traversal algorithms
  - Cycle detection
  - Dijkstra's algorithm (least cost shortest path)

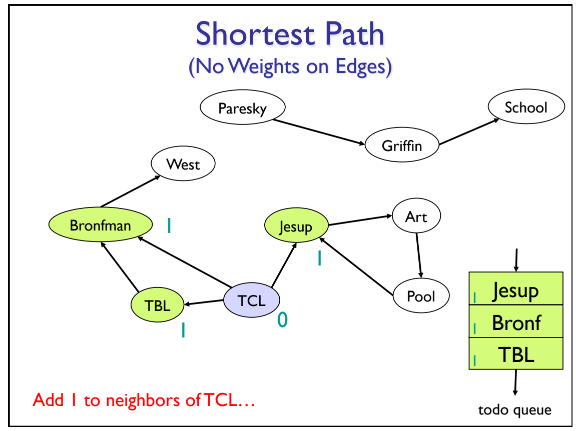
Shortest Path and Cycle Detection

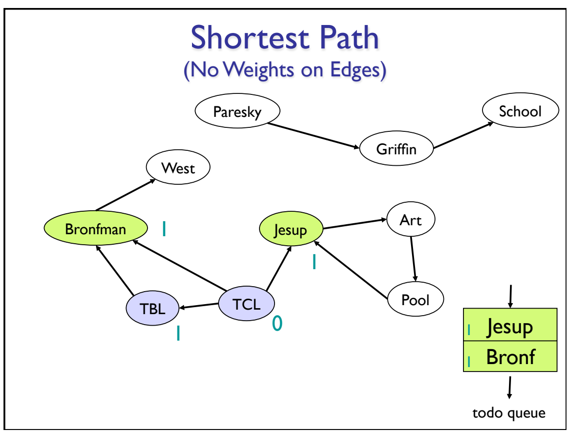
- Shortest path
  - How do we find shortest path from src to all nodes?
    - Could find all paths and then pick smallest…
    - …this is bad—there are many paths (O(n!))!
  - Can we use BFS or DFS to find *shortest* path to all nodes?
    - BFS (with labels on nodes that indicate "cost" from src)
- Cycle detection
  - Is there a path starting at src that contains a cycle?
  - Should we use BFS or DFS?
    - DFS (stack tells us path from src to current node; see if path leads to node already on stack)

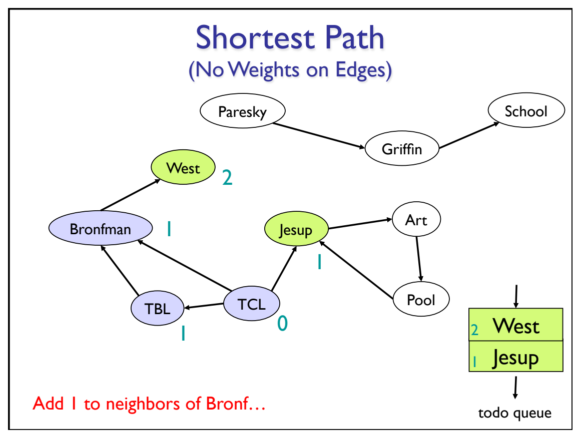
Shortest Path
(No Weights on Edges)

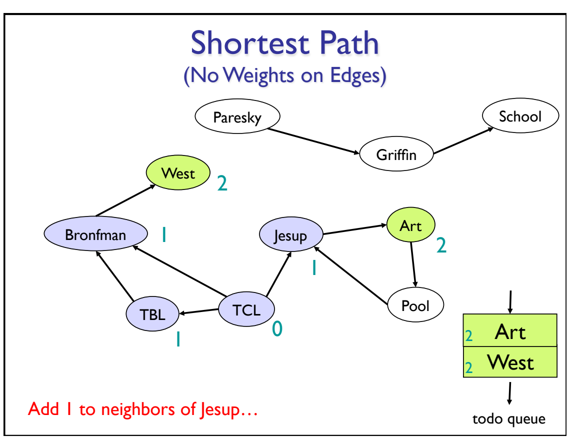


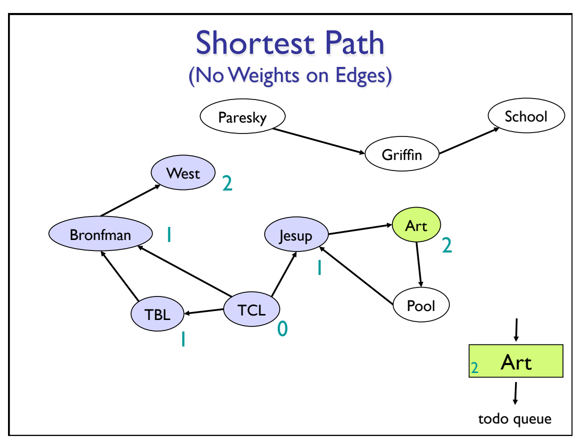

Shortest Path
(No Weights on Edges)
Paresky
School
Griffin
West
Bronfman
Jesup
Art
TCL
TBL
Pool
0
0
TCL
todo queue

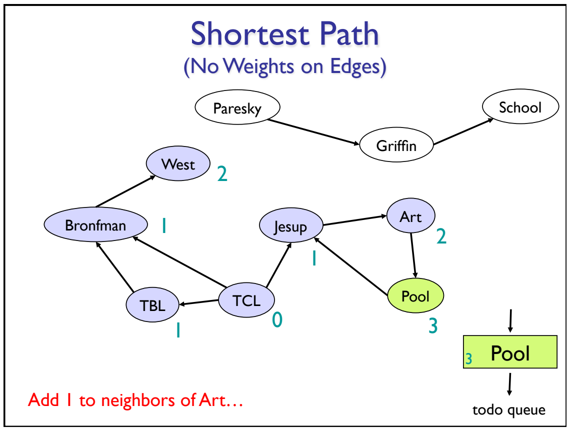

Shortest Path
(No Weights on Edges)
Paresky
School
Griffin
West
Bronfman
1
Jesup
1
Art
TBL
1
TCL
0
Pool
1 Jesup
1 Bronf
1 TBL
Add 1 to neighbors of TCL…
todo queue

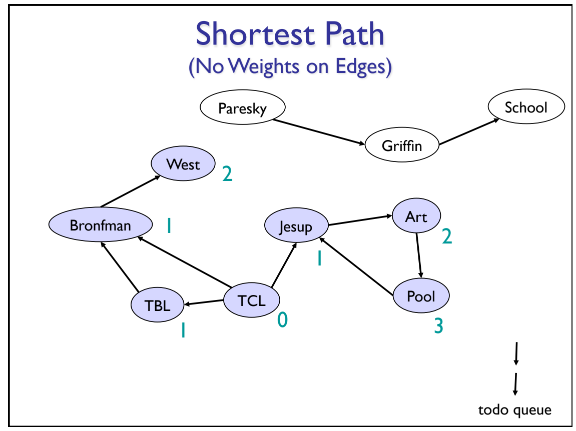

Shortest Path
(No Weights on Edges)
Paresky
School
Griffin
West
Bronfman
1
Jesup
1
Art
TBL
1
TCL
0
Pool
1 Jesup
1 Bronf
todo queue

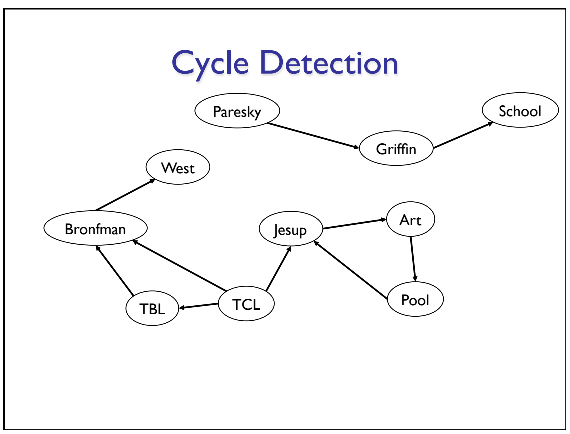

Shortest Path
(No Weights on Edges)
Paresky
School
Griffin
West
2
Bronfman
1
Jesup
1
Art
TBL
1
TCL
0
Pool
2 West
1 Jesup
Add 1 to neighbors of Bronf…
todo queue

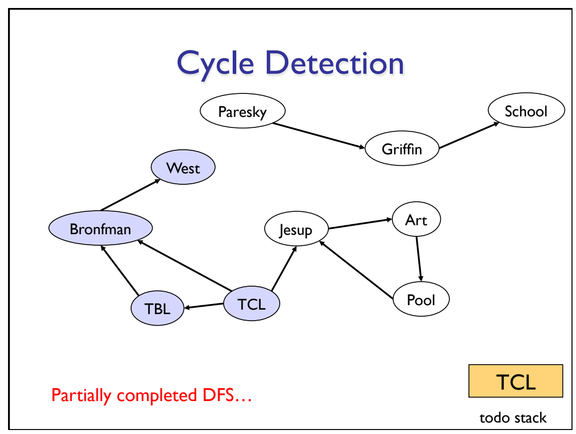

Shortest Path
(No Weights on Edges)
Paresky
School
Griffin
West
2
Bronfman
1
Jesup
1
Art
2
TBL
1
TCL
0
Pool
2 Art
2 West
Add 1 to neighbors of Jesup…
todo queue

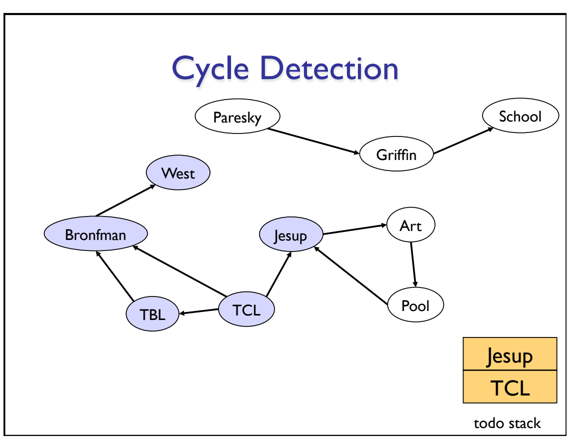

Shortest Path
(No Weights on Edges)
Paresky
School
Griffin
West
2
Bronfman
1
Jesup
1
Art
2
TBL
1
TCL
0
Pool
2 Art
todo queue

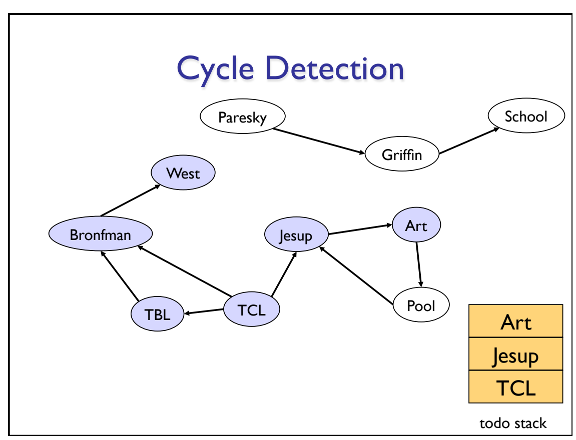

Shortest Path
(No Weights on Edges)
Paresky
School
Griffin
West
2
Bronfman
1
Jesup
Art
2
1
Pool
TBL
TCL
0
3
1
3 Pool
Add 1 to neighbors of Art…
todo queue


Shortest Path
(No Weights on Edges)
Paresky
School
Griffin
West
2
Bronfman
1
Jesup
Art
2
1
Pool
TBL
TCL
0
3
1
todo queue

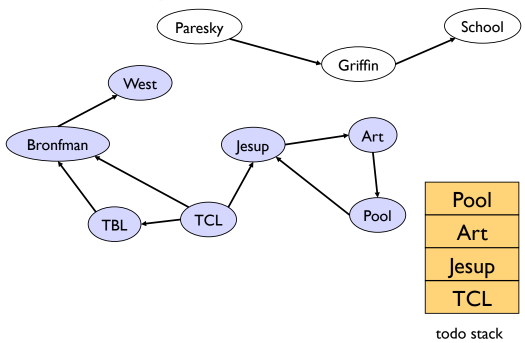

Cycle Detection
Paresky
School
Griffin
West
Bronfman
Jesup
Art
Pool
TBL
TCL


Cycle Detection
Paresky
School
Griffin
West
Bronfman
Jesup
Art
Pool
TBL
TCL
Partially completed DFS…
TCL
todo stack

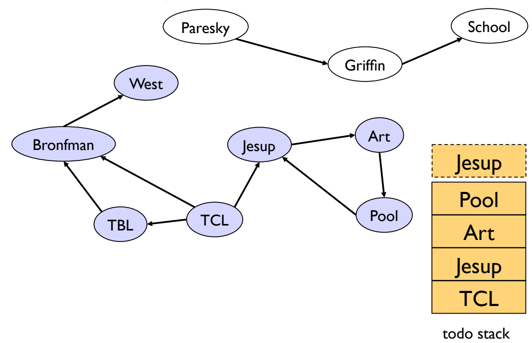

Cycle Detection
Paresky
School
Griffin
West
Bronfman
Jesup
Art
Pool
TBL
TCL
Jesup
TCL
todo stack


Cycle Detection
Paresky
School
Griffin
West
Bronfman
Jesup
Art
Pool
TBL
TCL
Art
Jesup
TCL
todo stack

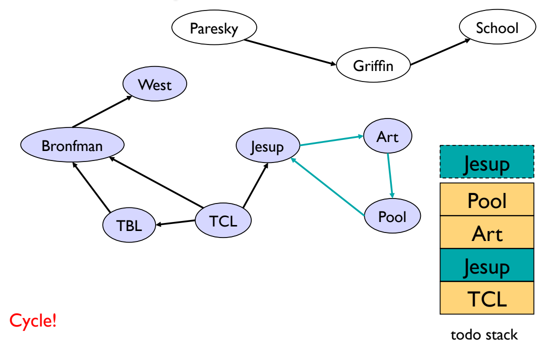
Cycle Detection



Cycle Detection

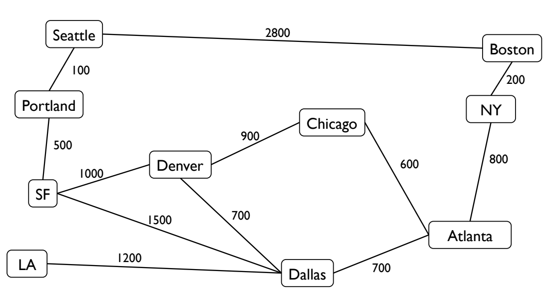


Cycle Detection

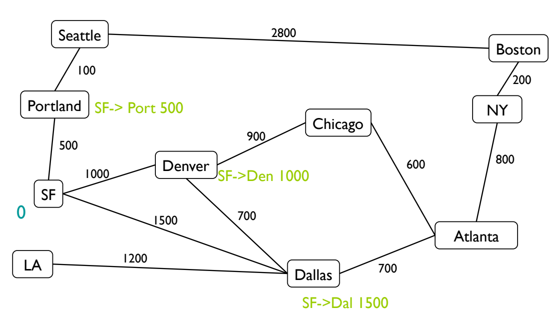


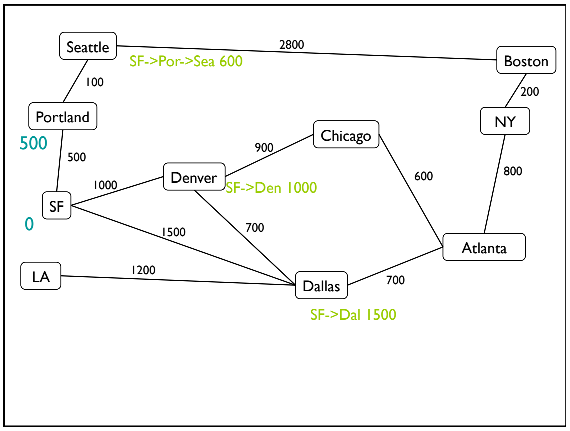
Shortest Path Revisited

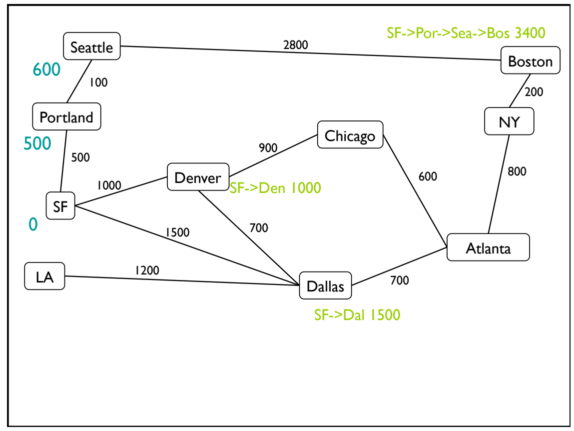
- What if there are weights on our edges?
- Will BFS still work?
  - No! BFS processes nodes according to number of edges/hops from src to node
- Need something else…

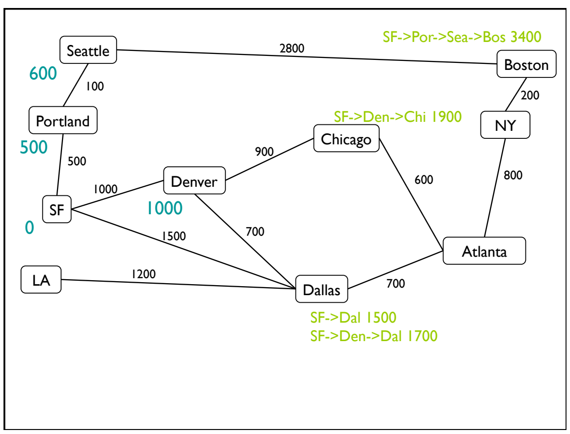


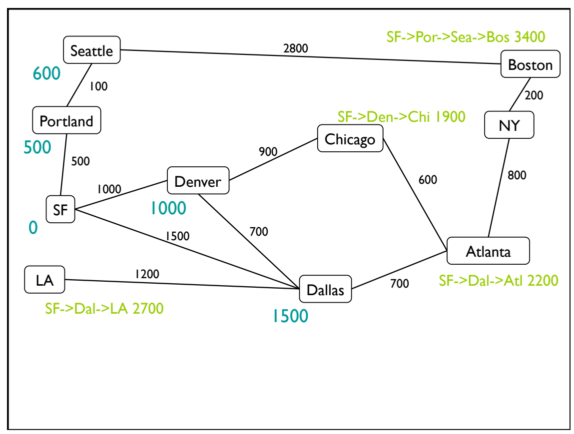
Intuition

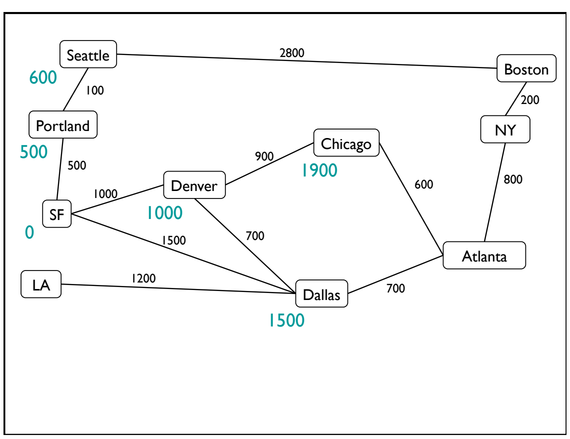


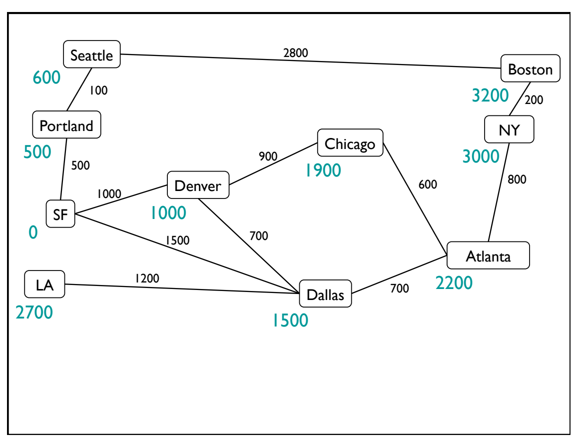

Seattle
Boston
Portland
NY
Chicago
Denver
SF
Atlanta
LA
Dallas
2800
100
200
500
900
600
800
1000
700
1500
1200
700
SF->Por->Sea 600
500
0
SF->Den 1000
SF->Dal 1500


Seattle
Boston
Portland
NY
Chicago
Denver
SF
Atlanta
LA
Dallas
2800
100
200
500
900
600
800
1000
700
1500
1200
700
SF->Por->Sea->Bos 3400
600
500
0
SF->Den 1000
SF->Dal 1500


Seattle
Boston
Portland
NY
Chicago
Denver
SF
Atlanta
LA
Dallas
2800
100
200
500
900
600
800
1000
700
1500
1200
700
SF->Por->Sea->Bos 3400
SF->Den->Chi 1900
600
500
1000
0
SF->Dal 1500
SF->Den->Dal 1700

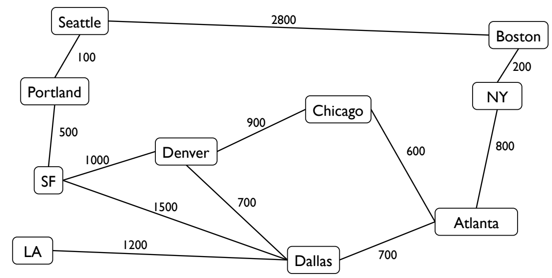

Seattle
Boston
Portland
NY
Chicago
Denver
SF
Atlanta
LA
Dallas
2800
100
200
500
900
600
800
1000
700
1500
1200
700
SF->Por->Sea->Bos 3400
SF->Den->Chi 1900
600
500
1000
0
SF->Dal->Atl 2200
SF->Dal->LA 2700
1500


Seattle
Boston
Portland
NY
Chicago
Denver
SF
Atlanta
LA
Dallas
2800
100
200
500
900
600
800
1000
700
1500
1200
700
600
500
1900
1000
0
1500

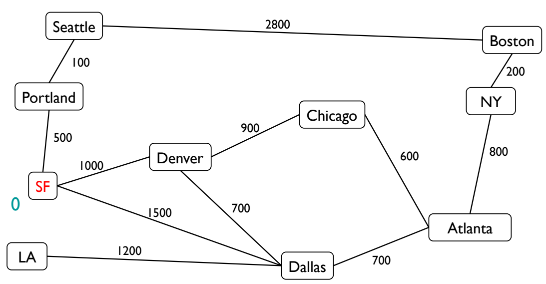

Seattle
Boston
Portland
NY
Chicago
Denver
SF
Atlanta
LA
Dallas
2800
100
200
500
900
600
800
1000
700
1500
1200
700
600
3200
500
3000
1900
1000
0
2200
2700
1500

Dijkstra's Algorithm

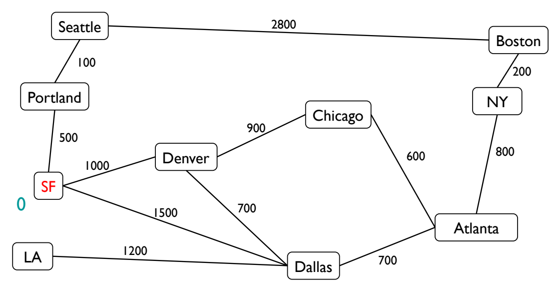
- Basic idea
  - Explore paths from src in order of increasing total cost
- Algorithm
  - Keep map from node to shortest-distance-to-node
  - Keep PQ of paths from SRC, ordered by total distance
  - While PQ is not empty
    - Take shortest path SRC-> … -> DEST from PQ
      - If DEST has not been visited
        - » Fix distance to DEST in map
        - » Extend path to all neighbors of DEST
        - » Add new paths to PQ



Dijkstra's Algorithm

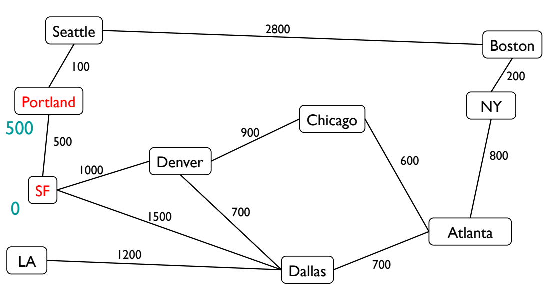


Priority Queue



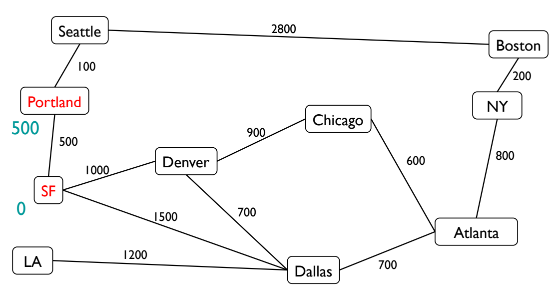
Priority Queue

SF->Port; 500	SF->Den; 1000	SF->Dal 1500



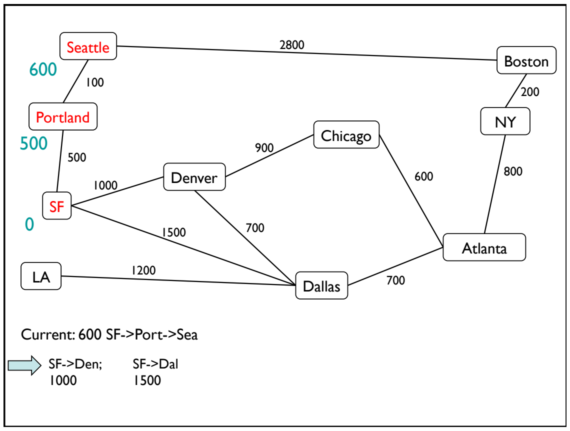
Current: 500 SF->Port (need to add Port's neighbors to PQ)

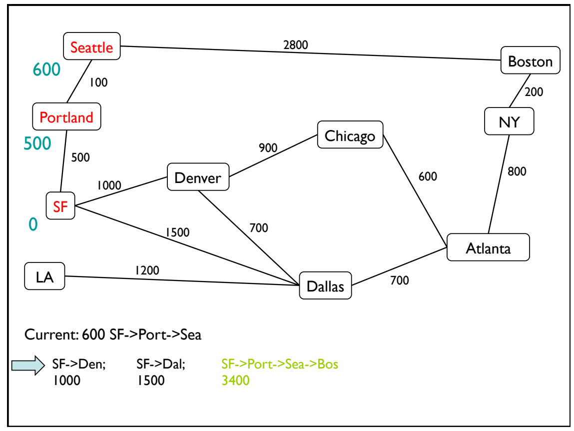
SF->Den; 1000	SF->Dal 1500

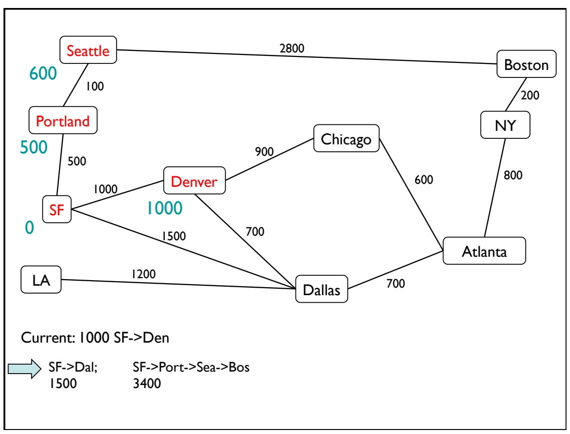


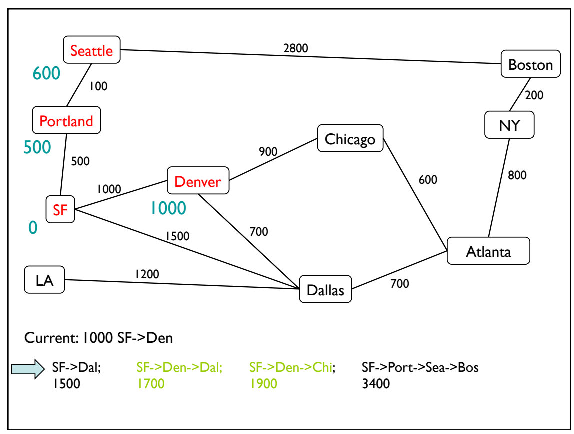
Current: 500 SF->Port

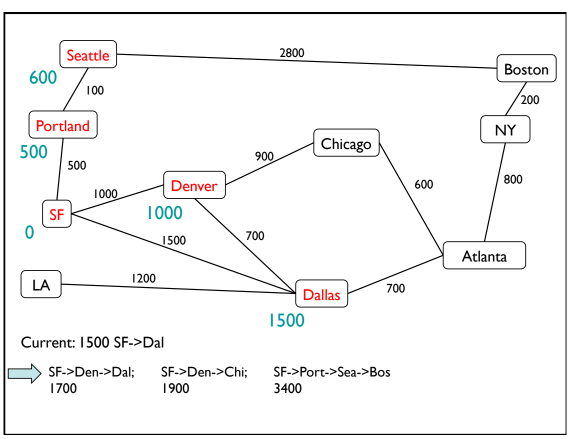
SF->Port->Sea; 600	SF->Den; 1000	SF->Dal 1500

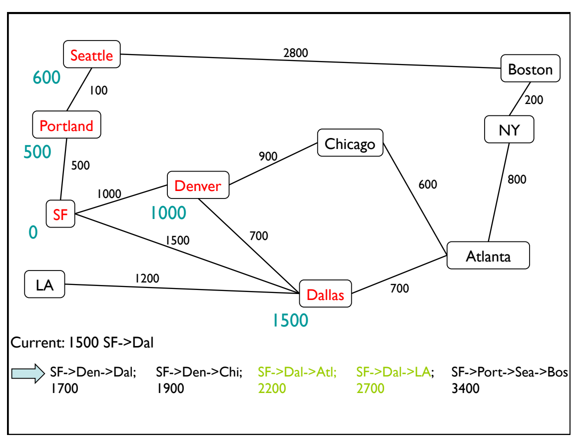

Seattle
2800
Boston
600
100
200
Portland
NY
500
900
Chicago
500
800
1000
Denver
600
SF
0
1500
700
Atlanta
LA
1200
Dallas
700
Current: 600 SF->Port->Sea
SF->Den; 1000
SF->Dal 1500

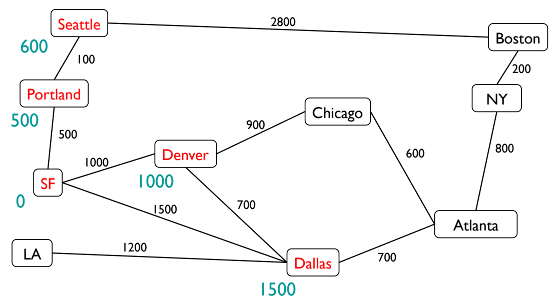

Seattle
2800
Boston
600
100
200
Portland
NY
500
900
Chicago
500
800
1000
Denver
600
SF
0
1500
700
Atlanta
LA
1200
Dallas
700
Current: 600 SF->Port->Sea
SF->Den; 1000
SF->Dal; 1500
SF->Port->Sea->Bos 3400


Seattle
2800
Boston
600
100
200
Portland
NY
500
900
Chicago
500
800
1000
Denver
600
SF
1000
0
1500
700
Atlanta
LA
1200
Dallas
700
Current: 1000 SF->Den
SF->Dal; 1500
SF->Port->Sea->Bos 3400


Seattle
2800
Boston
600
100
200
Portland
NY
500
900
Chicago
500
800
1000
Denver
600
SF
1000
0
1500
700
Atlanta
LA
1200
Dallas
700
Current: 1000 SF->Den
SF->Dal; 1500
SF->Den->Dal; 1700
SF->Den->Chi; 1900
SF->Port->Sea->Bos 3400

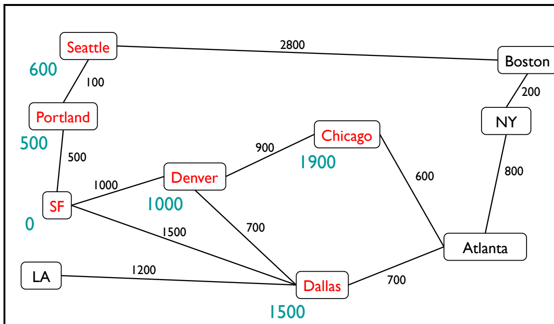

Seattle
2800
Boston
600
100
200
Portland
NY
500
900
Chicago
500
800
1000
Denver
600
SF
1000
0
1500
700
Atlanta
LA
1200
Dallas
700
1500
Current: 1500 SF->Dal
SF->Den->Dal; 1700
SF->Den->Chi; 1900
SF->Port->Sea->Bos 3400


Seattle
2800
Boston
600
100
200
Portland
NY
500
900
Chicago
500
800
1000
Denver
600
SF
1000
0
1500
700
Atlanta
LA
1200
Dallas
700
1500
Current: 1500 SF->Dal
SF->Den->Dal; 1700
SF->Den->Chi; 1900
SF->Dal->Atl; 2200
SF->Dal->LA; 2700
SF->Port->Sea->Bos 3400



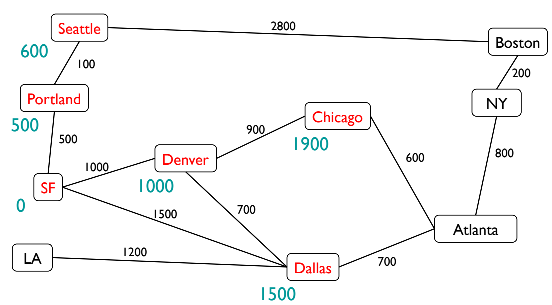
Current: 1700 SF->Den->Dal (we already have Dallas!)

SF->Den->Chi;	SF->Dal->Atl;	SF->Dal->LA;	SF->Port->Sea->Bos
1900	2200	2700	3400



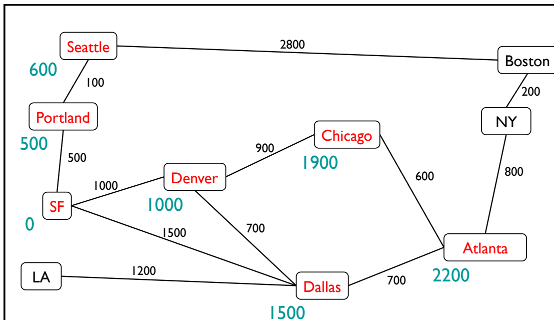
Current: 1900 SF->Den->Chi

SF->Dal->Atl;	SF->Dal->LA;	SF->Port->Sea->Bos
2200	2700	3400



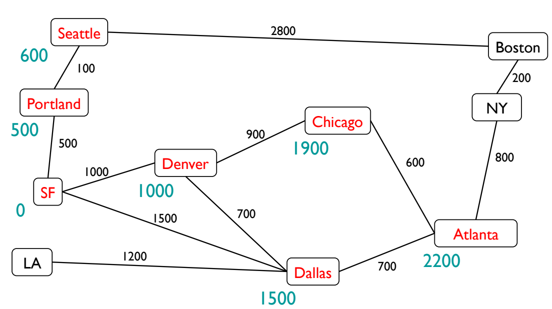
Current: 1900 SF->Den->Chi

SF->Dal->Atl;	SF->Den->Chi->Atl;	SF->Dal->LA;	SF->Port->Sea->Bos
2200	2500	2700	3400



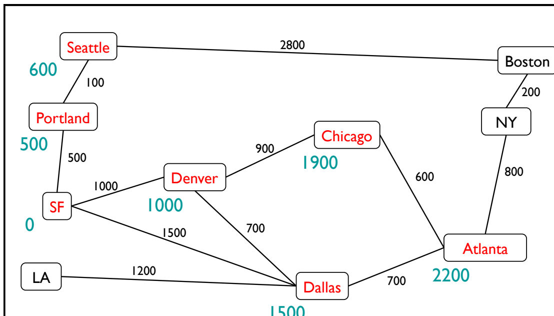
Current: 2200 SF->Dal->Atl

SF->Den->Chi->Atl;	SF->Dal->LA;	SF->Port->Sea->Bos
2500	2700	3400



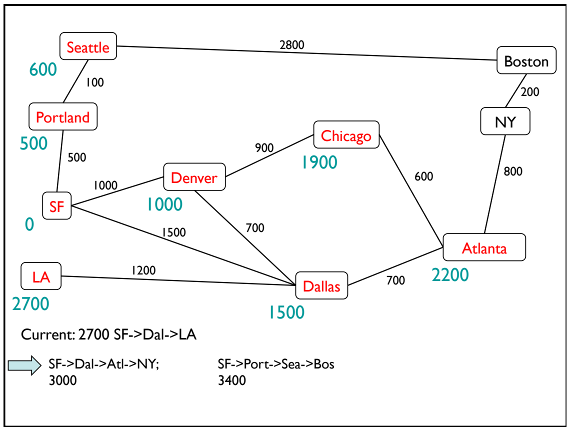
Current: 2200 SF->Dal->Atl

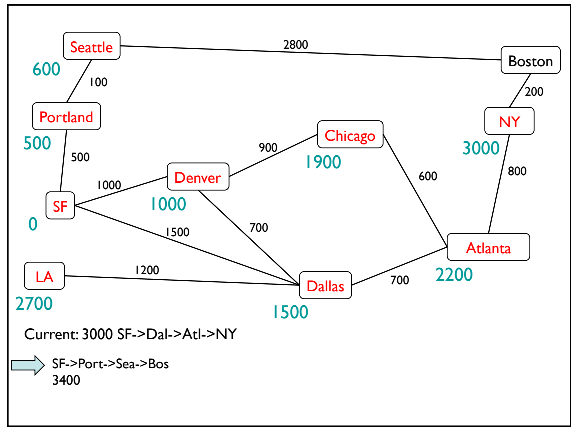
SF->Den->Chi->Atl;	SF->Dal->LA;	SF->Dal->Atl->NY;	SF->Port->Sea->Bos
2500	2700	3000	3400

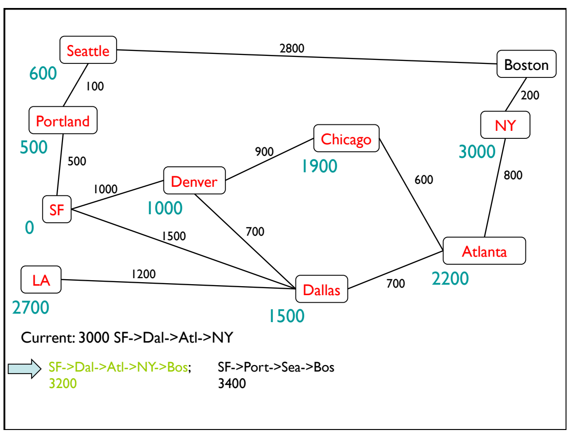


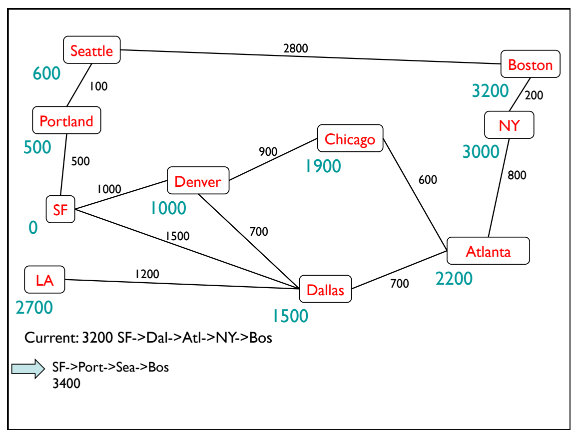
Current: 2500 SF->Den->Chi->Atl

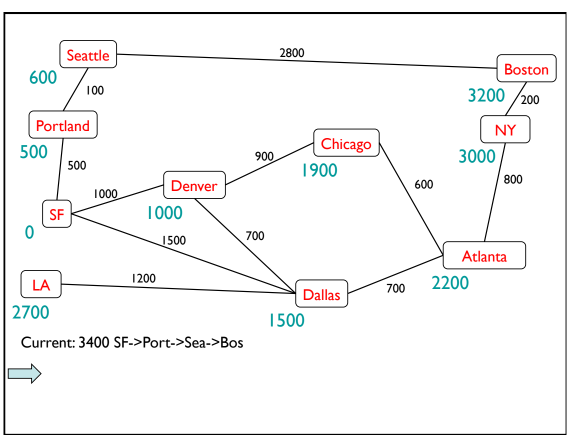
SF->Dal->LA;	SF->Dal->Atl->NY;	SF->Port->Sea->Bos
2700	3000	3400

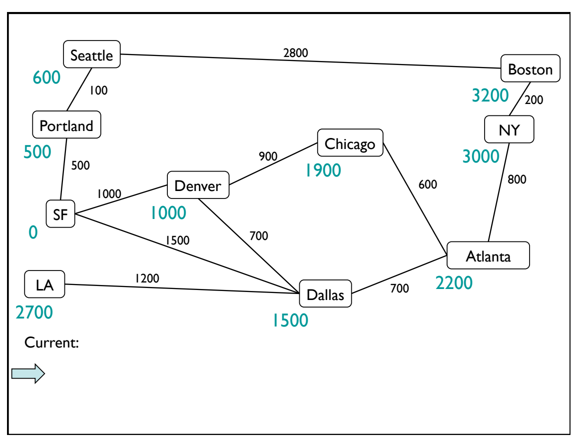

Seattle
2800
Boston
600
100
200
Portland
NY
500
Chicago
500
900
1900
600
800
1000
Denver
SF
1000
0
1500
700
Atlanta
LA
1200
Dallas
700
2200
2700
1500
Current: 2700 SF->Dal->LA
SF->Dal->Atl->NY; 3000
SF->Port->Sea->Bos 3400


Seattle
2800
Boston
600
100
200
Portland
NY
500
3000
Chicago
500
900
1900
600
800
1000
Denver
SF
1000
0
1500
700
Atlanta
LA
1200
Dallas
700
2200
2700
1500
Current: 3000 SF->Dal->Atl->NY
SF->Port->Sea->Bos 3400


Seattle
2800
Boston
600
100
200
Portland
NY
500
3000
Chicago
500
900
1900
600
800
1000
Denver
SF
1000
0
1500
700
Atlanta
LA
1200
Dallas
700
2200
2700
1500
Current: 3000 SF->Dal->Atl->NY
SF->Dal->Atl->NY->Bos; 3200
SF->Port->Sea->Bos 3400


Seattle
2800
Boston
600
100
3200
200
Portland
NY
500
3000
Chicago
500
900
1900
600
800
1000
Denver
SF
1000
0
1500
700
Atlanta
LA
1200
Dallas
700
2200
2700
1500
Current: 3200 SF->Dal->Atl->NY->Bos
SF->Port->Sea->Bos 3400

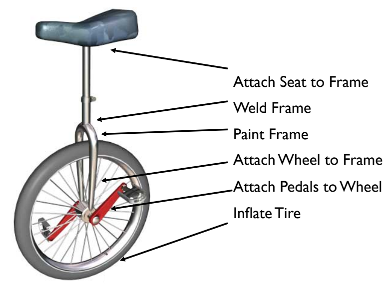

Seattle
2800
Boston
600
100
3200
200
Portland
NY
500
3000
Chicago
500
900
1900
600
800
1000
Denver
SF
1000
0
1500
700
Atlanta
LA
1200
Dallas
700
2200
2700
1500
Current: 3400 SF->Port->Sea->Bos

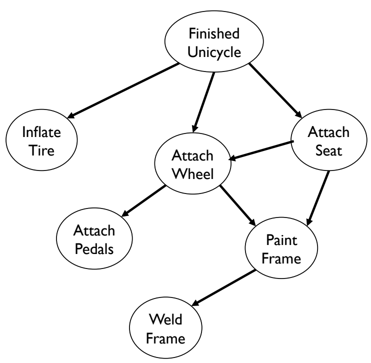

Seattle
2800
Boston
600
100
3200
200
Portland
NY
500
3000
Chicago
500
900
1900
600
800
1000
Denver
SF
1000
0
1500
700
Atlanta
LA
1200
Dallas
700
2200
2700
1500
Current:

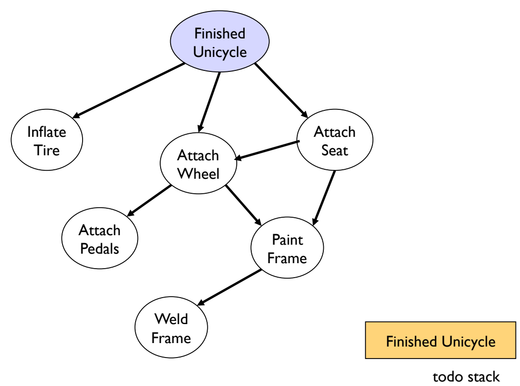
More Graph Algorithms!

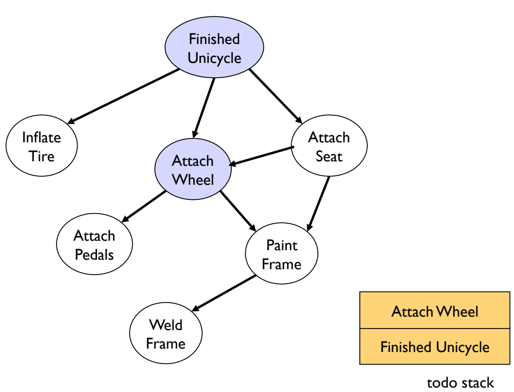
- Topological sorting (DFS)
  - Only valid for directed and acyclic graphs
  - List vertices in such a way as to make the edges point in one direction (i.e., back to beginning)
  - Commonly used to solve scheduling problems, assembly lines steps, etc.
- Minimum cost spanning tree (MST)
  - Find edges with least total weight that connects all nodes
  - Not quite DFS or BFS…uses a greedy algorithm to select edges
  - Important for phone, cable, water systems, etc.

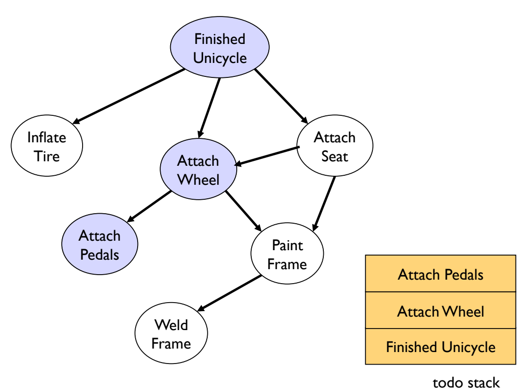
Topological Sort: Unicycle Factory

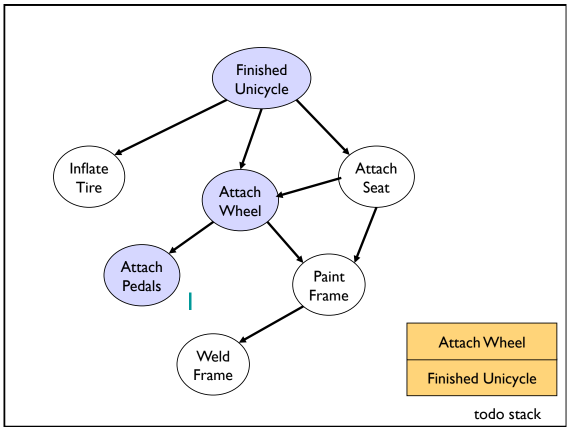


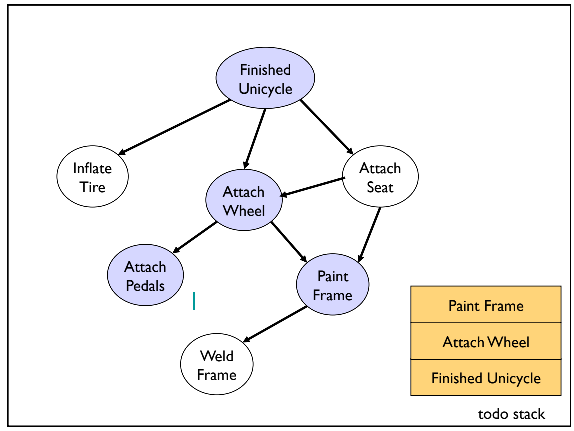


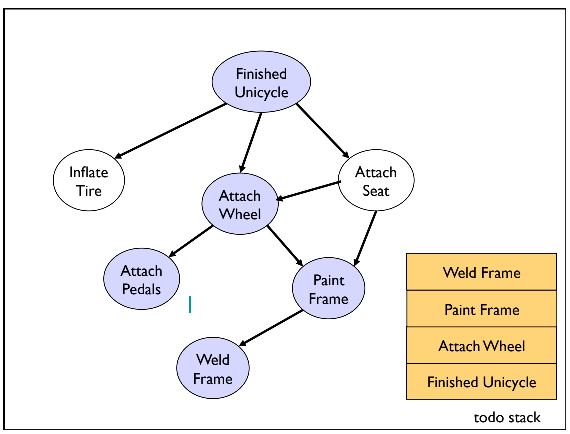


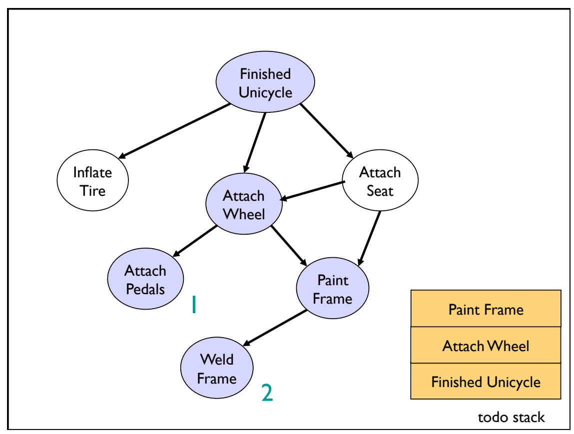


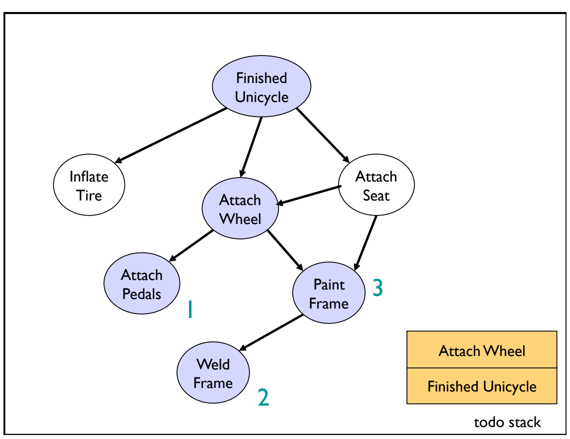


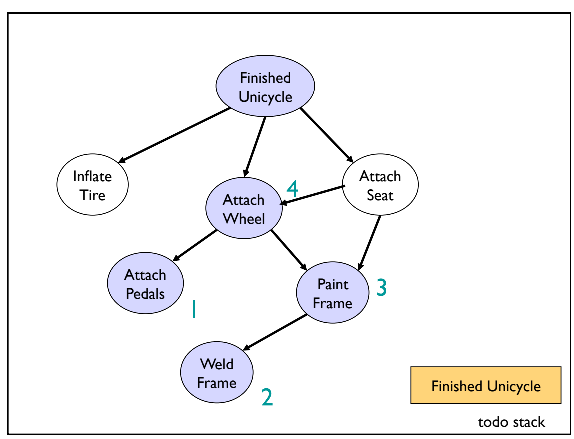

Finished Unicycle
Inflate Tire
Attach Wheel
Attach Seat
Attach Pedals
1
Paint Frame
Weld Frame
Attach Wheel
Finished Unicycle
todo stack

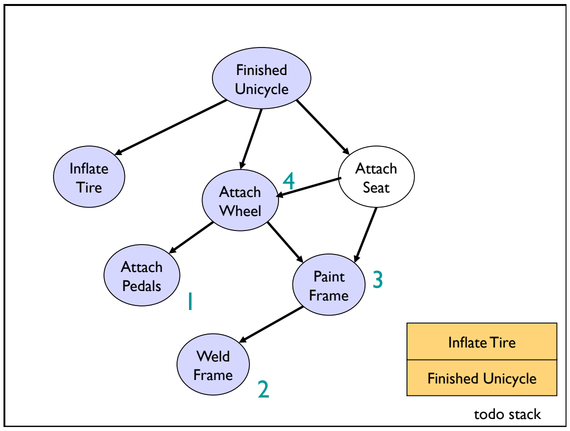

Finished Unicycle
Inflate Tire
Attach Wheel
Attach Seat
Attach Pedals
1
Paint Frame
Weld Frame
Paint Frame
Attach Wheel
Finished Unicycle
todo stack

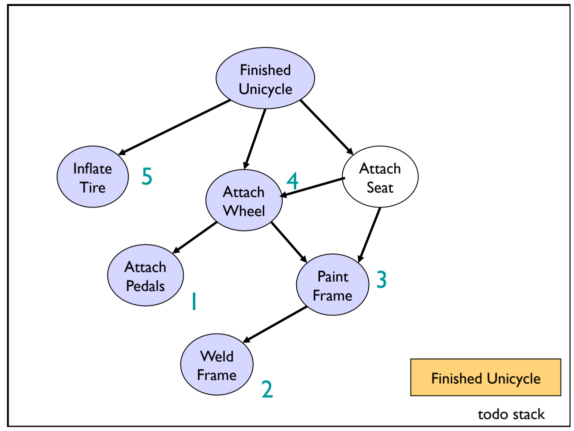

Finished Unicycle
Inflate Tire
Attach Wheel
Attach Seat
Attach Pedals
1
Paint Frame
Weld Frame
Weld Frame
Paint Frame
Attach Wheel
Finished Unicycle
todo stack

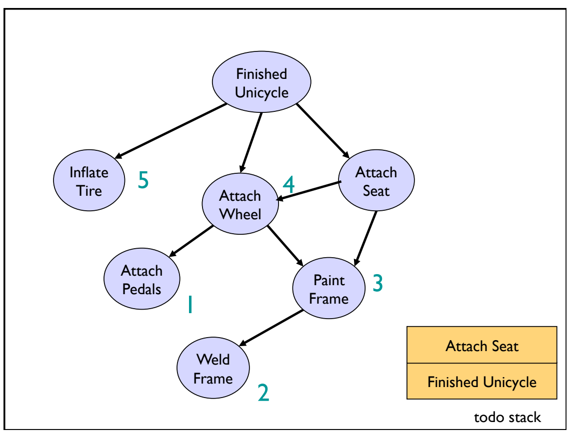

Finished Unicycle
Inflate Tire
Attach Wheel
Attach Seat
Attach Pedals
1
Paint Frame
Weld Frame
2
Paint Frame
Attach Wheel
Finished Unicycle
todo stack

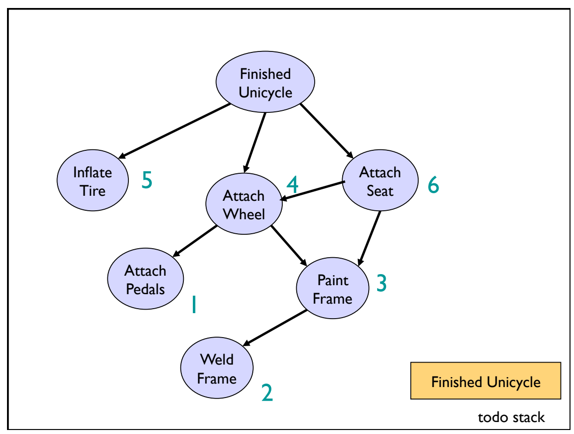

Finished Unicycle
Inflate Tire
Attach Wheel
Attach Seat
Attach Pedals
1
Paint Frame
3
Weld Frame
2
Attach Wheel
Finished Unicycle
todo stack

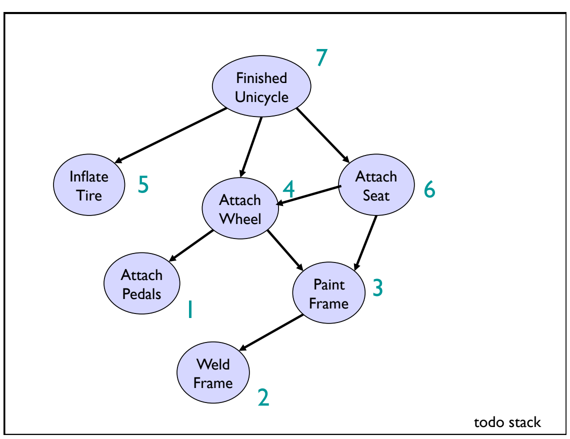

Finished Unicycle
Inflate Tire
Attach Wheel
4
Attach Seat
Attach Pedals
1
Paint Frame
3
Weld Frame
2
Finished Unicycle
todo stack

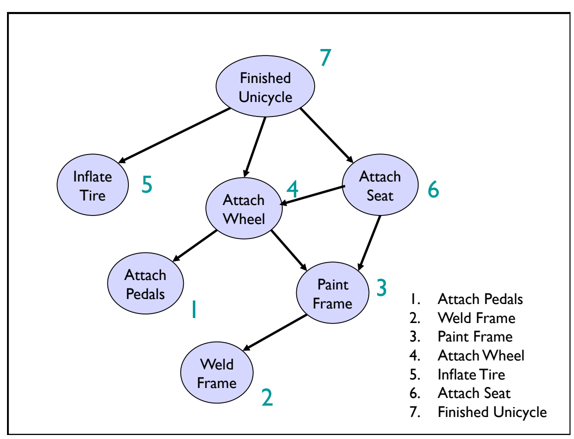

Finished Unicycle
Inflate Tire
Attach Wheel
4
Attach Seat
Attach Pedals
1
Paint Frame
3
Weld Frame
2
Inflate Tire
Finished Unicycle
todo stack


Finished Unicycle
Inflate Tire
5
Attach Wheel
4
Attach Seat
Attach Pedals
1
Paint Frame
3
Weld Frame
2
Finished Unicycle
todo stack

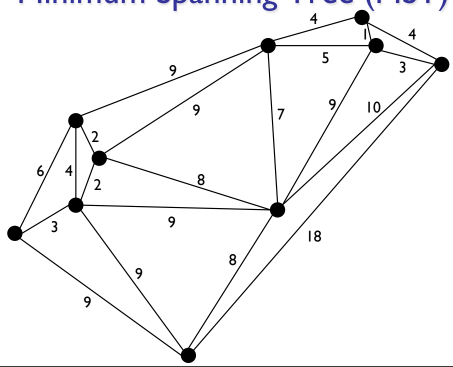

Finished Unicycle
Inflate Tire
5
Attach Wheel
4
Attach Seat
Attach Pedals
1
Paint Frame
3
Weld Frame
2
Attach Seat
Finished Unicycle
todo stack


Finished Unicycle
Inflate Tire
5
Attach Wheel
4
Attach Seat
6
Attach Pedals
1
Paint Frame
3
Weld Frame
2
Finished Unicycle
todo stack

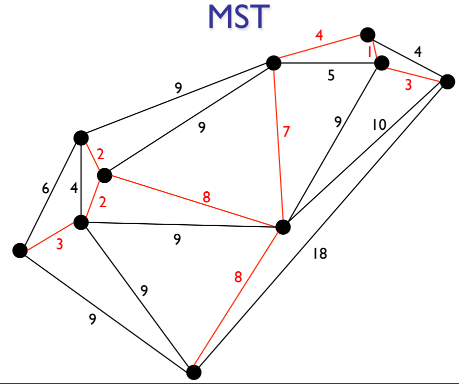

7
Finished Unicycle
Inflate Tire
5
Attach Wheel
4
Attach Seat
6
Attach Pedals
1
Paint Frame
3
Weld Frame
2
todo stack


7
Finished Unicycle
Inflate Tire
5
Attach Wheel
4
Attach Seat
6
Attach Pedals
1
Paint Frame
3
Weld Frame
2
1. Attach Pedals
2. Weld Frame
3. Paint Frame
4. Attach Wheel
5. Inflate Tire
6. Attach Seat
7. Finished Unicycle

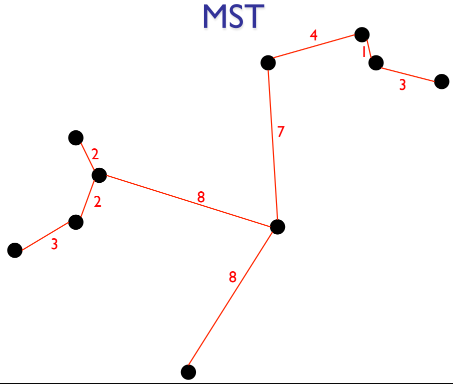
Minimum Spanning Tree (MST)



MST



MST



Moving on…Maps!

- Maps are data structures that map keys to values
  - Sometimes called dictionaries
  - Where have we seen maps before?
    - Associations!
    - An Association is a single key-value pair. Maps may contain lots of key-value pairs. (Kinda like a Vector of Associations, but better!)
- In Java, Maps are found in java.util package
- What methods are needed in the Map interface?

Map Interface

- Methods for Map<K, V>
  - int size() - returns number of entries in map
  - boolean isEmpty() - true iff there are no entries
  - boolean containsKey(K key) - true iff key exists in map
  - boolean containsValue(V val) - true iff val exists at least once in map
  - V get(K key) - get value associated with key
  - V put(K key, V val) - insert mapping from key to val, returns value replaced (old value) or null
  - V remove(K key) - remove mapping from key to val
  - void clear() - remove all entries from map

Map Interface

- Other methods for Map<K,V>:
  - void putAll(Map<K,V> other) - puts all key-value pairs from Map other in map
  - Set<K> keySet() - return set of keys in map
  - Set<Association<K,V>> entrySet() - return set of key-value pairs from map
  - Structure<V> valueSet() - return set of values
  - boolean equals() - used to compare two maps
  - int hashCode() - returns hash code associated with map (stay tuned…)

Sample Usage

- See Dictionary.java

Simple Map Implementation

- A simple implementation of the Map interface is the MapList class
- Uses a SinglyLinkedList of Associations as underlying data structure
- How would we implement put(K key, V val)?
- What is the running time of:
  - containsKey(K key)?
  - containsValue(V val)?
- Bottom line: not O(1)!

Search/Locate Revisited

- How long does it take to search for objects in Vectors and Lists?
  - O(n) on average
- How about in BSTs?
  - O(log n)
- Can this be improved?
  - With hash tables, YES!
  - Can locate objects in roughly O(1) time
  - …to be continued!