CSC 321: Data Structures

Fall 2018

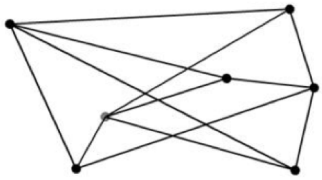
Graphs & search

- graphs, isomorphism
- simple vs. directed vs. weighted
- adjacency matrix vs. adjacency lists
- graph traversals
  depth-first search, breadth-first search
- finite automata

Graphs

trees are special instances of the more general data structure: graphs

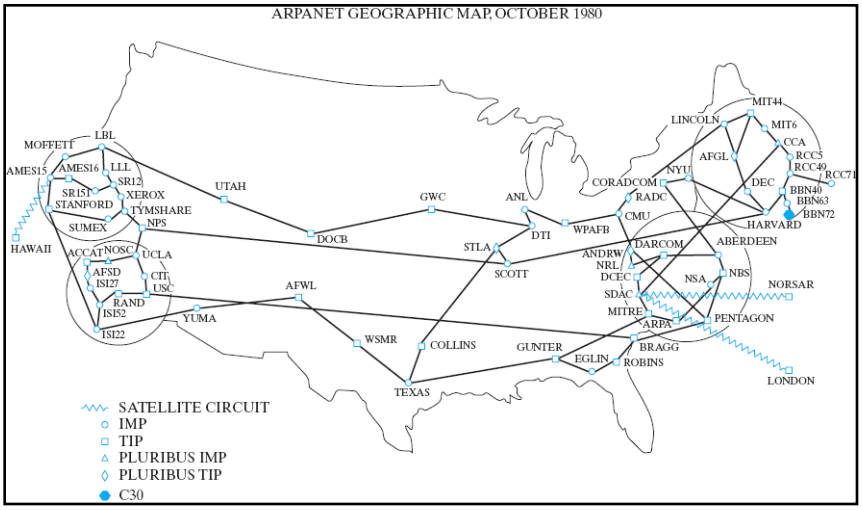
- informally, a graph is a collection of nodes/data elements with connections



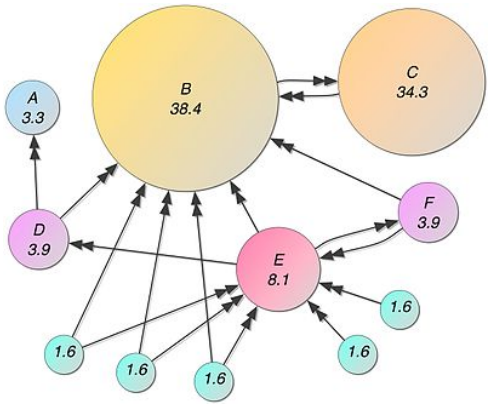
many real-world and CS-related applications rely on graphs

- the connectivity of computers in a network
- cities on a map, connected by roads
- cities on a airline schedule, connected by flights
- Facebook friends
- finite automata

Interesting graphs



Interesting graphs



Google pagerank algorithm

- the pagerank of a page is based on the number of pages that link to it and their pageranks

https://en.wikipedia.org/wiki/PageRank

Interesting graphs

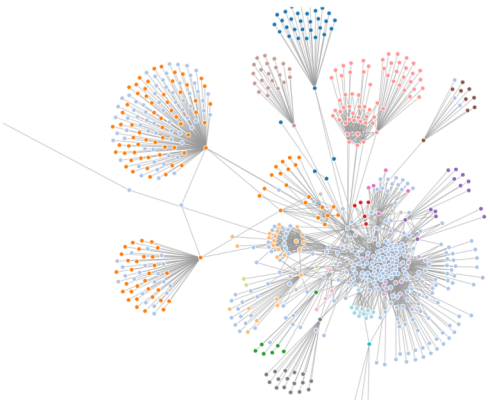
TouchGraph Facebook Browser

- allows you to view all of your friends and groups as a graph

http://www.touchgraph.com/facebook

5

Interesting graphs



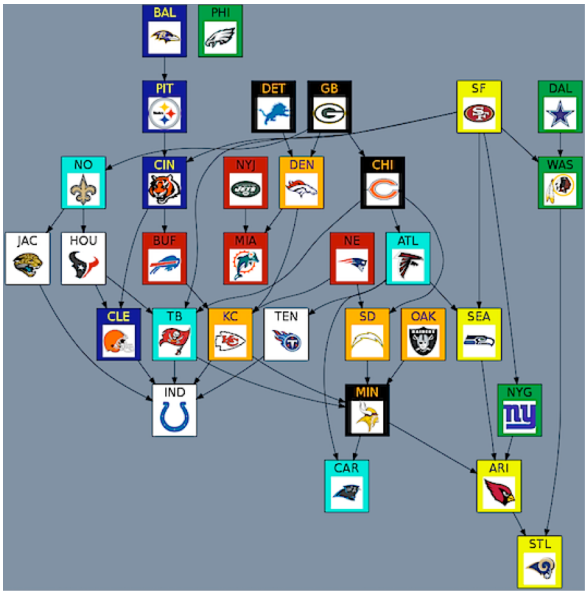
Enron email graph

- was used during prosecution to identify conspirators

http://www.contextsmith.com/blog/enron-emails-help-improve-project-communication/

6

Interesting graphs



NFL 2011 @ week 11

- edges from one team to another show a clear beat path
- e.g., Chicago > Tampa > Indy

http://beatpaths.com/

7

Formal definition

a *simple graph* G consists of a nonempty set V of vertices, and a set E of edges (unordered vertex pairs)

- can write simply as G = (V, E)

e.g., consider G = (V, E) where V = {a, b, c, d} and E = {{a,b}, {a, c}, {b,c}, {c,d}}
➔ DRAW IT!

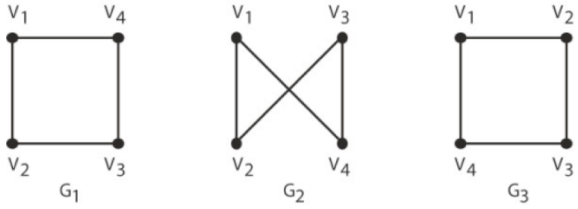
graph terminology for G = (V, E)

- vertices u and v are *adjacent* if {u, v} ∈ E (i.e., connected by an edge)
- edge {u,v} is *incident* to vertices u and v
- the *degree* of v is the # of edges incident with it (or # of vertices adjacent to it)
- |V| = # of vertices |E| = # of edges
- a path between v_1 and v_n is a sequence of edges from E:
  [$\{v_1, v_2\}, \{v_2, v_3\}, \ldots, \{v_{n-2}, v_{n-1}\}, \{v_{n-1}, v_n\}$]

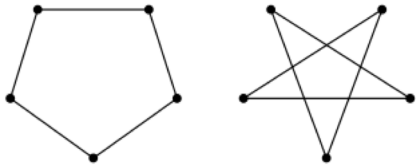
8

Depicting graphs

you can draw the same set of vertices & edges many different ways



are these different graphs?

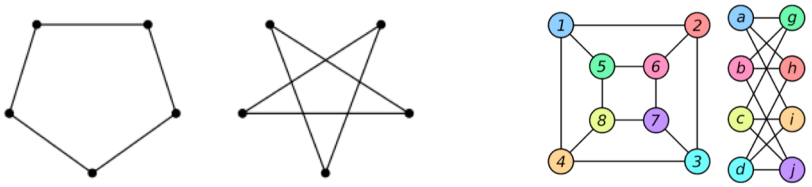


9

Isomorphism

how can we tell if two graphs are the same?

- $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are *isomorphic* if there is mapping $f:V_1 \rightarrow V_2$ such that $\{u,v\} \in E_1$ iff $\{f(u), f(v)\} \in E_2$
- i.e., G_1 and G2 are *isomorphic* if they are identical modulo a renaming of vertices



10

Simple vs. directed?

in a simple graph, edges are presumed to be bidirectional

- {u,v} ∈ E is equivalent to saying {v,u} ∈ E

for many real-world applications, this is appropriate

- if computer c_1 is connected to computer c_2, then c_2 is connected to c_1
- if person p_1 is Facebook friends with person p_2, then p_2 is friends with p_1

however, in other applications the connection may be 1-way

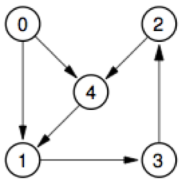
- a flight from city c_1 to city c_2 does not imply a return flight from c_2 to c_1
- roads connecting buildings in a town can be 1-way (with no direct return route)
- NFL beat paths are clearly directional

11

Directed graphs

a *directed graph* or *digraph* G consists of a nonempty set V of vertices, and a set E of edges (ordered vertex pairs)

- draw edge (u,v) as an arrow pointing from vertex u to vertex v



e.g., consider G = (V, E) where V = {a, b, c, d} and E = {(a,b), (a, c), (b,c), (c,d)}

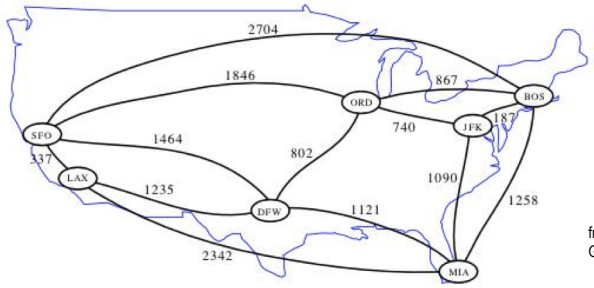
➔DRAW IT!

12

Weighted graphs

a *weighted graph* G = (V, E) is a graph/digraph with an additional weight function $\omega: E \rightarrow \Re$

- draw edge (u,v) labeled with its weight



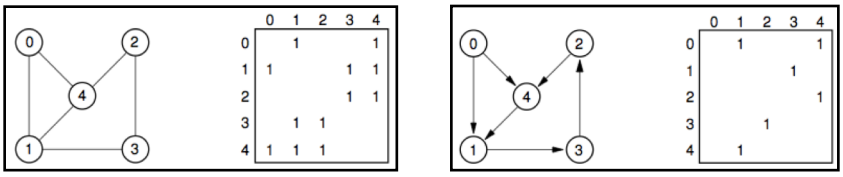
from *Algorithm Design* by Goodrich & Tamassia

- shortest path from BOS to SFO?
- does the fewest legs automatically imply the shortest path in terms of miles?

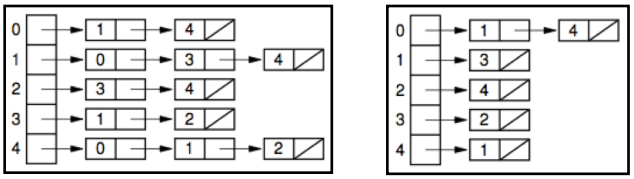
13

Graph data structures

- an *adjacency matrix* is a 2-D array, indexed by the vertices
  for graph G, adjMatrix[u][v] = 1 iff {u,v} ∈ E (or weights in place of 1s)



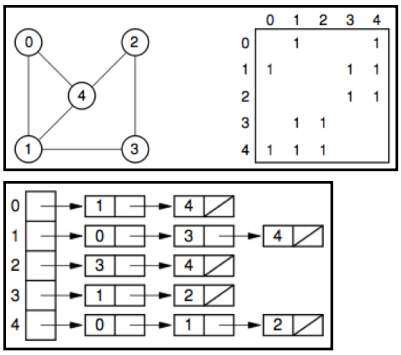
- an *adjacency list* is an array of linked lists
  for graph G, adjList[u] contains v iff {u,v} ∈ E (or weights in addition to vertex)



note: our HashMap of HashMaps was a smart implementation of an adjacency list

14

Matrix vs. list



space tradeoff

- adjacency matrix requires $O(|V|^2)$ space
- adjacency list requires $O(|V| + |E|)$ space

which is better?

- it depends
- if the graph is sparse (few edges), then $|V|+|E| < |V|^2$ → adjacency list
- if the graph is dense, i.e., $|E|$ is $O(|V|^2)$, then $|V|^2 < |V|+|E|$ → adjacency matrix

many graph algorithms involve visiting every adjacent neighbor of a node

- adjacency list makes this efficient, so tends to be used more in practice

15

Java implementations

to allow for alternative implementations, we will define a Graph interface

```
import java.util.Set;

public interface Graph<E> {
    public boolean isAdjacentTo(E v1, E v2);
    public Set<E> getAllAdjacent(E v);
}
```

many other methods are possible, but these suffice for now

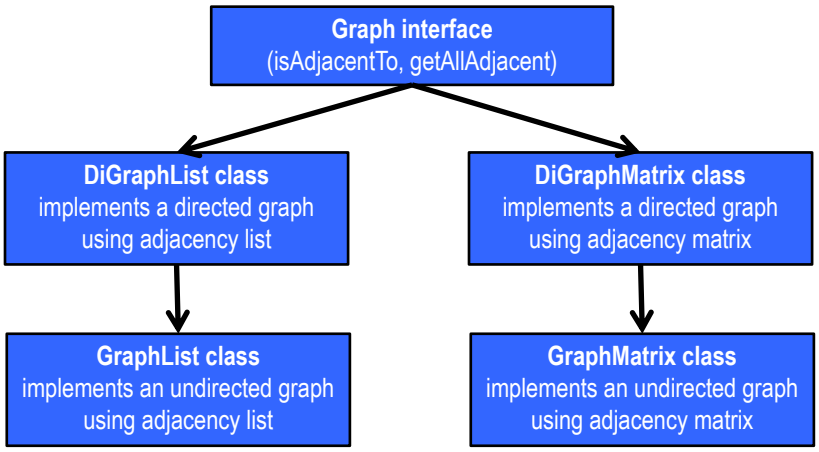
from this, we can implement simple graphs or directed graphs

- can utilize an adjacency matrix or an adjacency list

16

Class hierarchy

in addition to the Graph interface, can utilize inheritance to simplify the development of the graph variants



17

DiGraphList

```
public class DiGraphList<E> implements Graph<E>{
  private Map<E, Set<E>> adjLists;

  public DiGraphList(List<Pair<E, E>> edges) {
    this.adjLists = new HashMap<E, Set<E>>();
    for (Pair<E, E> edgePair : edges) {
      this.addEdge(edgePair.getKey(), edgePair.getValue());
    }
  }

  protected void addEdge(E v1, E v2) {
    if (!this.adjLists.containsKey(v1)) {
      this.adjLists.put(v1, new HashSet<E>());
    }
    this.adjLists.get(v1).add(v2);
  }

  public boolean isAdjacentTo(E v1, E v2) {
    return this.adjLists.containsKey(v1) &&
           this.getAllAdjacent(v1).contains(v2);
  }

  public Set<E> getAllAdjacent(E v) {
    if (!this.adjLists.containsKey(v)) {
      return new HashSet<E>();
    }
    return this.adjLists.get(v);
  }
}
```

we could implement the adjacency list using a standard array of LinkedLists

- conceptually equivalent but simpler approach: use a HashMap to map each vertex to a Set of adjacent vertices
- (but it does waste memory)
- constructor takes a list of edges (stored as Pairs)
- helper method `addEdge` handles storing an edge in the map

18

DiGraphMatrix

```
public class DiGraphMatrix<E> implements Graph<E>{
  private E[] vertices;
  private Map<E, Integer> lookup;
  private boolean[][] adjMatrix;

  public DiGraphMatrix(List<Pair<E, E>> edges) {
    Set<E> vertexSet = new HashSet<E>();
    for (Pair<E, E> edgePair : edges) {
      vertexSet.add(edgePair.getKey());
      vertexSet.add(edgePair.getValue());
    }

    this.vertices = (E[])(new Object[vertexSet.size()]);
    this.lookup = new HashMap<E, Integer>();

    int index = 0;
    for (E nextVertex : vertexSet) {
      this.vertices[index] = nextVertex;
      lookup.put(nextVertex, index);
      index++;
    }

    this.adjMatrix = new boolean[index][index];

    for (Pair<E, E> edgePair : edges) {
      this.addEdge(edgePair.getKey(), edgePair.getValue());
    }
  }

  ...
```

adjacency matrix implementation is tricky in Java

- have to first determine the set of vertices in order to create the table
- since Java arrays are only accessible via indices, have to map to/from labels

- `vertices` array converts index → label
- `lookup` map converts label → index

19

DiGraphMatrix

```
  ...

  protected void addEdge(E v1, E v2) {
    Integer index1 = this.lookup.get(v1);
    Integer index2 = this.lookup.get(v2);
    if (index1 != null && index2 != null) {
      this.adjMatrix[index1][index2] = true;
    }
  }

  public boolean isAdjacentTo(E v1, E v2) {
    Integer index1 = this.lookup.get(v1);
    Integer index2 = this.lookup.get(v2);
    return index1 != null && index2 != null &&
           this.adjMatrix[index1][index2];
  }

  public Set<E> getAllAdjacent(E v) {
    Integer row = this.lookup.get(v);

    Set<E> neighbors = new HashSet<E>();
    if (row != null) {
      for (int c = 0; c < this.adjMatrix.length; c++) {
        if (this.adjMatrix[row][c]) {
          neighbors.add(this.vertices[c]);
        }
      }
    }
    return neighbors;
  }
}
```

`addEdge` is easy

- lookup indices of vertices
- then store `true` in matrix at row/column

`isAdjacentTo` also easy

- lookup indices of vertices
- then access row/column

`getAllAdjacent` is more work

- lookup index of vertex
- traverse row
- collect all adjacent vertices in a set

20

GraphMatrix & GraphList

can utilize inheritance to implement simple graph variants

- can utilize the same internal structures, just add edges in both directions

```
public class GraphList<E> extends DiGraphList<E> {
  public GraphList(List<Pair<E, E>> edges) {
    super(edges);
  }

  protected void addEdge(E v1, E v2) {
    super.addEdge(v1, v2);
    super.addEdge(v2, v1);
  }
}
```

```
public class GraphMatrix<E> extends DiGraphMatrix<E> {
  public GraphMatrix(List<Pair<E, E>> edges) {
    super(edges);
  }

  protected void addEdge(E v1, E v2) {
    super.addEdge(v1, v2);
    super.addEdge(v2, v1);
  }
}
```

constructors simply call the superclass constructors

override the `addEdge` methods so that they add edges in both directions (using the superclass `addEdge`)

21

Graph traversal

many applications involve systematically searching through a graph

- starting at some vertex, want to traverse the graph and inspect/process vertices along paths

e.g., list all the computers on the network

e.g., find a sequence of flights that get the customer from BOS to SFO (perhaps with some property?)

similar to tree traversals, we can follow various patterns

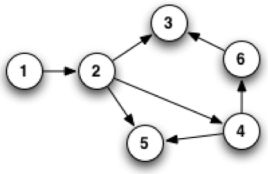
- recall for trees: inorder, preorder, postorder

for arbitrary graphs, two main alternatives

- depth-first search (DFS): boldly follow a single path as long as possible
- breadth-first search (BFS): tentatively try all paths level-by-level

22

DFS vs. BFS



consider the following digraph

- suppose we wanted to find a path from 1 to 5

depth-first search would:

▪ start with the starting vertex	1
▪ select an adjacent vertex from there	1 → 2
▪ select an adjacent vertex from there	1 → 2 → 3
▪ dead-end, so back up	1 → 2
▪ select a different adjacent vertex	1 → 2 → 4
▪ select an adjacent vertex from there	1 → 2 → 4 → 5

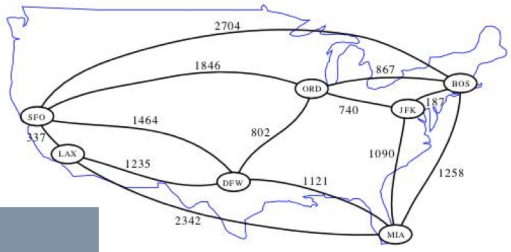
breadth-first search would:

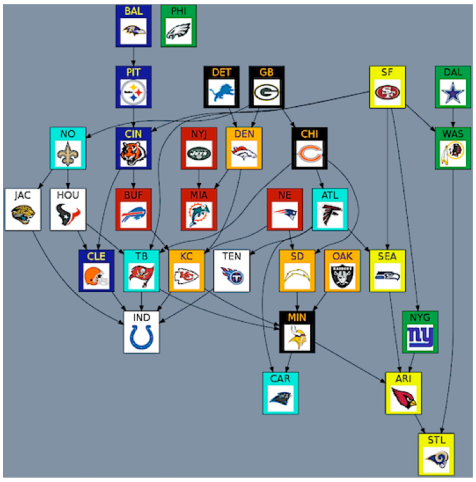
▪ start with the starting vertex	1
▪ expand to length-1 paths	1 → 2
▪ expand to length-2 paths	1 → 2 → 3
	1 → 2 → 4
	1 → 2 → 5

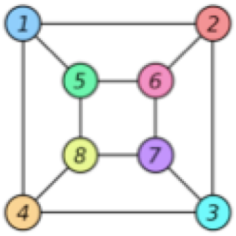
23

Bigger examples

DFS? BFS?







24

DFS implementation

DFS can be implemented recursive backtracking (more in CSC421)

```
public static <E> List<E> DFSrec(Graph<E> g, E start, E goal) {
    List<E> path = new ArrayList<E>();
    path.add(start);
    if (GraphSearch.DFSrec(g, path, goal)) {
        return path;
    }
    return null;
}

private static <E> boolean DFSrec(Graph<E> g,
                                  List<E> pathSoFar, E goal) {
    E lastVertex = pathSoFar.get(pathSoFar.size()-1);
    if (lastVertex.equals(goal)) {
        return true;
    }
    else {
        for (E adj : g.getAllAdjacent(lastVertex)) {
            if (!pathSoFar.contains(adj)) {
                pathSoFar.add(adj);
                if (GraphSearch.DFSrec(g, pathSoFar, goal)) {
                    return true;
                }
                pathSoFar.remove(pathSoFar.size()-1);
            }
        }
        return false;
    }
}
```

recursive method is passed path so far and goal state

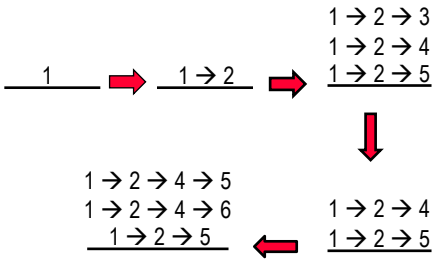
- BASE CASE: when path so far ends in goal state
- RECURSIVE: extend the path for each adjacent vertex and recurse
- to avoid cycles, don't extend if the adjacent vertex is already in the path

25

Stacks & Queues

DFS can alternatively be implemented using a stack of paths

- initially contains a single path (w/ start)
- at each step, pop the path at the top of the stack
- extend that path for each adjacent vertex, and push that extended path on the stack



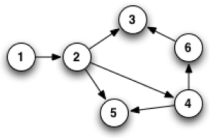
BFS is most naturally implemented using a queue of paths

- initially contains a single path (w/ start)
- at each step, remove the path at the front of the queue
- extend that path for each adjacent vertex, and add that extended path to the back of the queue

[1]

[1 → 2]

[1→ 2 → 5, 1 → 2 → 4, 1 → 2 → 3]



26

Implementations

```
public static <E> List<E> DFS(Graph<E> g, E start, E goal) {
    List<E> startPath = new ArrayList<E>();
    startPath.add(start);

    Stack<List<E>> paths = new Stack<List<E>>();
    paths.push(startPath);

    while (!paths.isEmpty()) {
        List<E> pathToExtend = paths.pop();
        E current = pathToExtend.get(pathToExtend.size()-1);
        if (current.equals(goal)) {
            return pathToExtend;
        }
        else {
            for (E adj : g.getAllAdjacent(current)) {
                if (!pathToExtend.contains(adj)) {
                    List<E> copy = new ArrayList<E>(pathToExtend);
                    copy.add(adj);
                    paths.push(copy);
                }
            }
        }
    }
    return null;
}
```

27

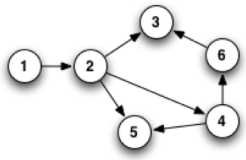
Implementations

```
public static <E> List<E> BFS(Graph<E> g, E start, E goal) {
    List<E> startPath = new ArrayList<E>();
    startPath.add(start);

    Queue<List<E>> paths = new LinkedList<List<E>>();
    paths.add(startPath);

    while (!paths.isEmpty()) {
        List<E> pathToExtend = paths.remove();
        E current = pathToExtend.get(pathToExtend.size()-1);
        if (current.equals(goal)) {
            return pathToExtend;
        }
        else {
            for (E adj : g.getAllAdjacent(current)) {
                if (!pathToExtend.contains(adj)) {
                    List<E> copy = new ArrayList<E>(pathToExtend);
                    copy.add(adj);
                    paths.add(copy);
                }
            }
        }
    }
    return null;
}
```

28

DFSrec vs. DFS vs. BFS

```
public static void main(String[] args) throws java.io.FileNotFoundException {
    List<Pair<Integer, Integer>> edges = new ArrayList<Pair<Integer, Integer>>();
    edges.add(new Pair<Integer, Integer>(1, 2));
    edges.add(new Pair<Integer, Integer>(2, 3));
    edges.add(new Pair<Integer, Integer>(2, 4));
    edges.add(new Pair<Integer, Integer>(2, 5));
    edges.add(new Pair<Integer, Integer>(4, 5));
    edges.add(new Pair<Integer, Integer>(4, 6));
    edges.add(new Pair<Integer, Integer>(6, 3));

    Graph<Integer> g = new DiGraphList<Integer>(edges);

    System.out.println("DFSrec " + GraphSearch.DFSrec(g, 1, 3));
    System.out.println("DFSrec " + GraphSearch.DFSrec(g, 1, 5));
    System.out.println();
    System.out.println("DFS " + GraphSearch.DFS(g, 1, 3));
    System.out.println("DFS " + GraphSearch.DFS(g, 1, 5));
    System.out.println();
    System.out.println("BFS " + GraphSearch.BFS(g, 1, 3));
    System.out.println("BFS " + GraphSearch.BFS(g, 1, 5));
}
```




```
DFSrec [1, 2, 3]
DFSrec [1, 2, 4, 5]

DFS [1, 2, 4, 6, 3]
DFS [1, 2, 5]

BFS [1, 2, 3]
BFS [1, 2, 5]
```

29

HW6: PathFinder

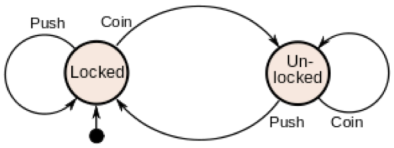
```
public List<StateLabel> findPath(StateLabel startState, StateLabel endState) {
    List<StateLabel> startPath = new ArrayList<StateLabel>();
    startPath.add(startState);

    Queue<List<StateLabel>> paths = new LinkedList<List<StateLabel>>();
    paths.add(startPath);

    while (!paths.isEmpty()) {
        List<StateLabel> shortestPath = paths.remove();
        StateLabel current = shortestPath.get(shortestPath.size()-1);
        if (current.equals(endState)) {
            return shortestPath;
        }
        else {
            for (StateLabel s : this.getAllAdjacentStates(current)) {
                if (!shortestPath.contains(s)) {
                    List<StateLabel> copy = new ArrayList<StateLabel>(shortestPath);
                    copy.add(s);
                    paths.add(copy);
                }
            }
        }
    }
    return null;
}
```

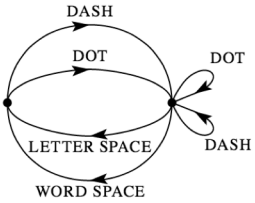
HW6 utilizes a modified BFS to find shortest paths in a FSM

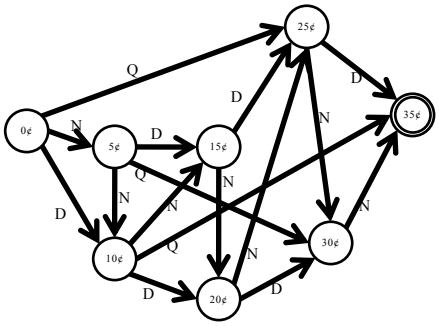
30

Finite State Machines (revisited)

recall: a *Finite State Machine* (a.k.a. *Finite Automaton*) defines a finite set of states (i.e., vertices) along with transitions between those states (i.e., edges)



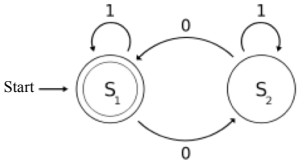


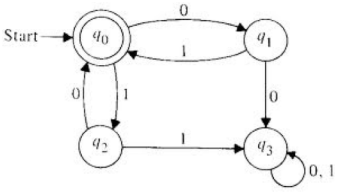


31

Recognition/acceptance

we say that a FSM *recognizes a pattern* if all sequences that meet that pattern terminate in an accepting state (drawn with double circle)





accepts
- 1
- 00
- 0101
- 0001110

in general?

accepts
- 10
- 01
- 1010
- 100110

in general?

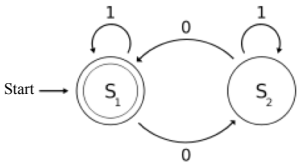
32

Regular expressions

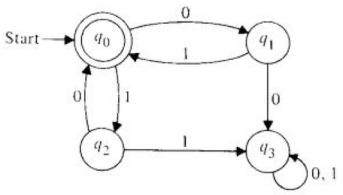
alternatively, we say that a FSM *defines a language* made up of the sequences accepted by the FSM

- a language is known as a *regular language* if there exists a FSM that accepts it
- a *regular expression* denotes a regular language using * for arbitrary repetition and + for OR

L_1 = 1*(01*0)*1*



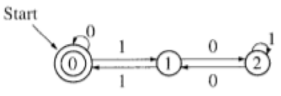
L_2 = (01 + 10)*



33

Old GRE CS exam – sample question

28.



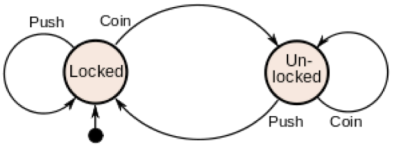
State 0 is both the starting state and the accepting state.

Each of the following is a regular expression that denotes a subset of the language recognized by the automaton above EXCEPT

(A) 0*(11)*0*
(B) 0*1(10*1)*1
(C) 0*1(10*1)*10*
(D) 0*1(10*1)0(100)*
(E) (0*1(10*1)*10* + 0*)*

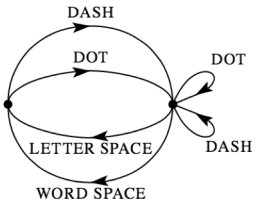
34

Regular language examples



if `locked` is initial state and `unlocked` is goal state:

```
Coin (Push Push* Coin)* Coin*
```



if `betweenLetters` is initial and goal state:

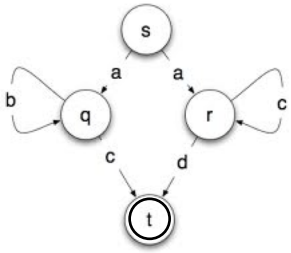
```
((Dot + Dash) (Dot + Dash)* (Lspace + Wspace))*
```

35

Deterministic vs. nondeterministic

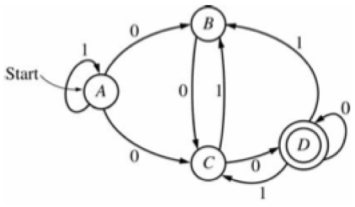
the examples so far are *deterministic*, i.e., each transition is unambiguous

- a FSM can also be *nondeterministic*, if there are multiple transitions from a state with different labels/weights
- a nondeterministic FSM accepts a sequence if there exists a path that accepts
- nondeterministic FSMs are especially useful for defining complex regular languages



36

MFT CS exam – sample question



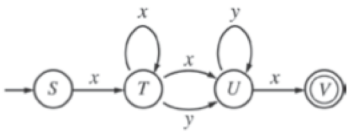
1. If D is the accepting state of the nondeterministic finite automaton above, which of the following does the automaton accept?

   (A) 001
   (B) 1101
   (C) 01100
   (D) 000110
   (E) 100100

37

GRE CS exam – sample question

Consider the following nondeterministic finite state automaton over alphabet $\{x, y\}$ with start state S.



4. Which of the following is the regular expression corresponding to the automaton above?

   (A) $xxx + yyx$

   (B) x^3y^2x

   (C) x^*y^*x

   (D) $xx^*(x + y)y^*x$

   (E) x^*xy^*x

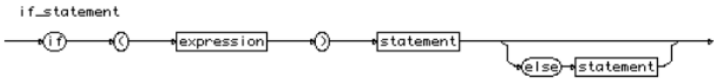
38

Uses of FSM

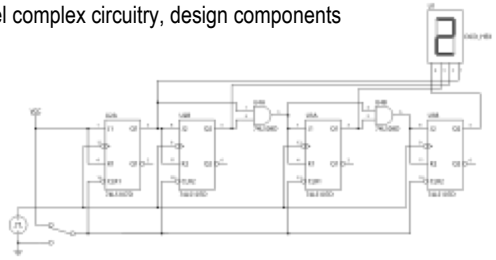
FSM machines are useful in many areas of CS

- designing software for controlling devices (e.g., vending machine, elevator, …)
  first identify states, constructing a table of state transitions, then implement in code

- parsing symbols for programming languages or text processing
  may define the language using rules or via a regular expression, then build a FSM to parse the characters into tokens & symbols



- designing circuitry
  commonly used to model complex circuitry, design components



39