CSC 427: Data Structures and Algorithm Analysis

Fall 2011

Decrease & conquer

- previous examples
- search spaces
  examples: travel, word ladder
- depth first search
  w/o cycle checking, w/ cycle checking
- breadth first search
  w/o cycle checking, w/ cycle checking

1

Divide & Conquer

RECALL: the divide & conquer approach tackles a complex problem by breaking it into proportionally-sized pieces

- e.g., merge sort divided the list into halves, conquered (sorted) each half, then merged the results
- e.g., to count number of nodes in a binary tree, break into counting the nodes in each subtree (which are smaller), then adding the results + 1

divide & conquer is a natural approach to many problems and tends to be efficient when applicable

sometimes, the pieces only reduce the problem size by a constant amount

- such decrease-and-conquer approaches tend to be less efficient

Cost(N) = Cost(N/2) + C ➔ O(log N)

Cost(N) = Cost(N-1) + C ➔ O(N)

2

Decrease & Conquer

previous examples of decrease-and-conquer

- *sequential search*: check the first item; if not it, search the remainder
- *linked list length*: count first node, then add to length of remainder

- *selection sort*: find the smallest item and swap it into the first location;
  then sort the remainder of the list
- *insertion sort*: traverse the items, inserting each into the correct position in a sorted list

some problems can be naturally viewed as a series of choices

- e.g., a driving trip, the N-queens problem

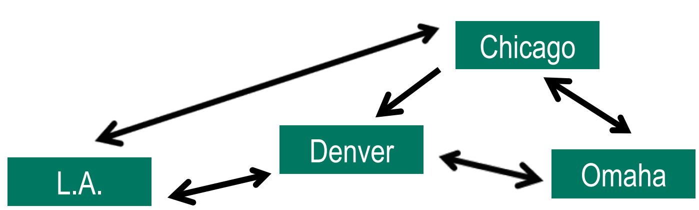
can treat these as decrease-and-conquer problems

- take the first step, then solve the problem from that point

Example: airline connections

suppose you are planning a trip from Omaha to Los Angeles

initial situation:	located in Omaha	
goal:	located in Los Angeles	
possible flights:	Omaha → Chicago Omaha → Denver Chicago → Denver Chicago → Los Angeles Chicago → Omaha	Denver → Los Angeles Denver → Omaha Los Angeles → Chicago Los Angeles → Denver

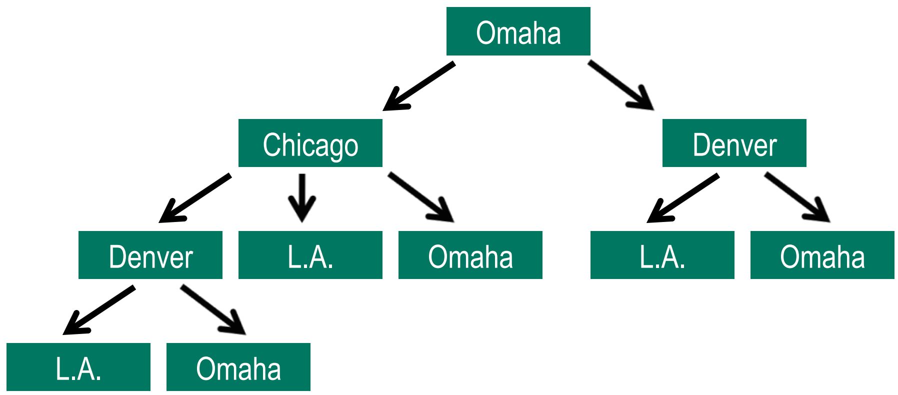


State space search

can view the steps in problem-solving as a tree

node: a situation or state
edge: the action moving between two situations/states

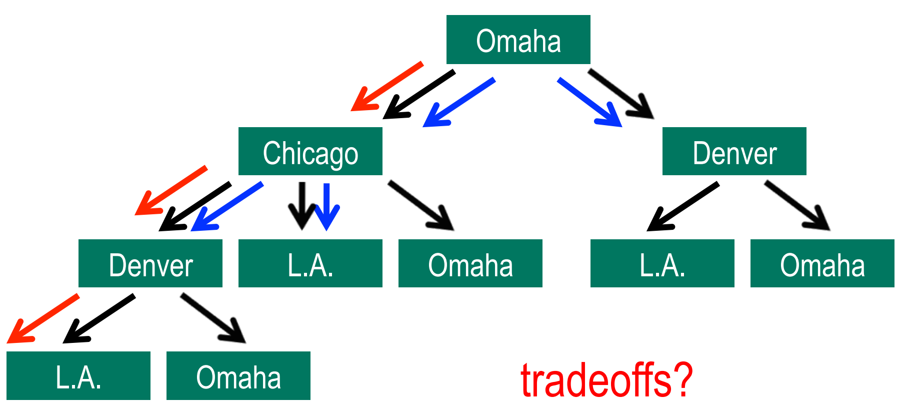
goal: find a path from the start (root) to a node with desired properties



DFS vs. BFS

two common strategies for conducting a search are:

- depth first search: boldly work your way down one path
- breadth first search: try each possible move at a level before moving to next



tradeoffs?

Example: word ladders

suppose we want to write a program for generating word ladders

- given start & end words, find a sequence of words that bridge them
- each word in the sequence should differ from the previous one by only one letter

white
while
whale
shale
shade

there can be multiple shortest ladders between 2 words

- can be many ladders that are longer

white → whine→ shine→ shone→ shore→ share→ shade

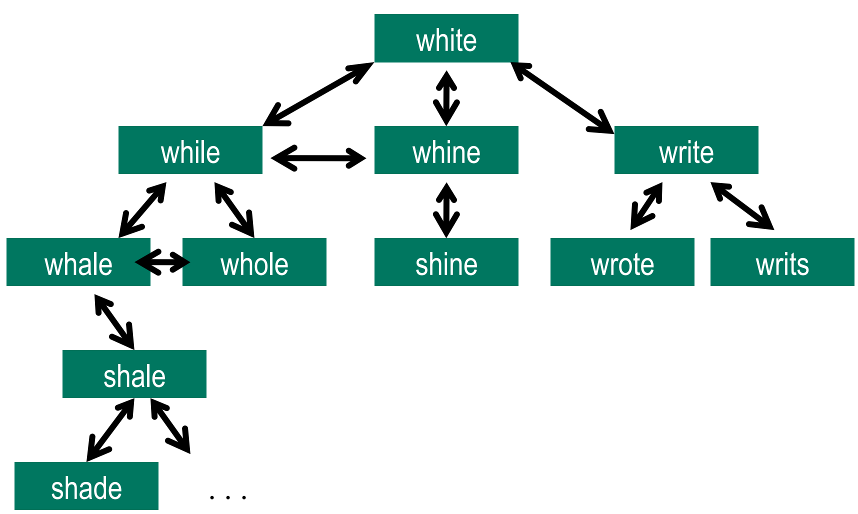
white → whine→ shine→ spine → spire → spore → shore→ share→ shade

7

Word ladder search

again, can think of words graphically

- two words are connected if they differ by one letter
- goal is to find a path from start word in ladder to end word



8

AdjacencyGraph

we could attempt to solve the travel and word ladder problems separately

- however, these are just generalizations of the same problem
- given a state-space graph, find a path from one state to another

travel: start state is "in Omaha", goal state is "in L.A."
word ladder: start state is "white", end state is "shade"

better: generalize the common behavior as an interface

```
import java.util.Set;

/**
 * Interface that defines basic operations on a graph.
 *   @author Dave Reed
 *   @version 11/11/11
 */
public interface AdjacencyGraph<E> {
    public boolean contains(E item);
    public Set<E> adjacencies(E item);
}
```

9

FlightGraph

for travel problem, need to define a class that implements AdjacencyGraph

- will need to store city names as nodes in the graph
- cities are connected if there is a flight between them

- YOU WILL DO THIS AS PART OF HW5

```
public class FlightGraph implements AdjacencyGraph<String> {
    public FlightGraph(String fileName) {
       ???
    }

    public boolean contains(String word) {
        ???
    }

    public Set<String> adjacencies(String word) {
        ???
    }
}
```

10

DictionaryGraph

- for word ladders, we need to be able to get words that are off by 1 letter
- DictionaryGraph stores a dictionary of words as a list
- finds adjacent words by traversing the entire list, collecting each word that differs by one letter

```
public class DictionaryGraph implements AdjacencyGraph<String> {
    private ArrayList<String> dictionary;

    public DictionaryGraph(String fileName) {
       this.dictionary = new ArrayList<String>();
       try {
            Scanner infile = new Scanner(new File(fileName));
            while (infile.hasNext()) {
                this.dictionary.add(infile.next());
            }
        }
        catch (java.io.FileNotFoundException e) {
            System.out.println("DICTIONARY FILE NOT FOUND");
        }
    }

    public boolean contains(String word) {
        return this.dictionary.contains(word);
    }

    public Set<String> adjacencies(String word) {
        Set<String> adjSet = new HashSet<String>();
        for (String nextWord : this.dictionary) {
            if (differByOne(nextWord, word)) {
                adjSet.add(nextWord);
            }
        }
        return adjSet;
    }

    private boolean differByOne(String word1, String word2) {
        // CODE NOT SHOWN
    }
}
```


Problem-solving as search

finally, we can design our general purpose methods

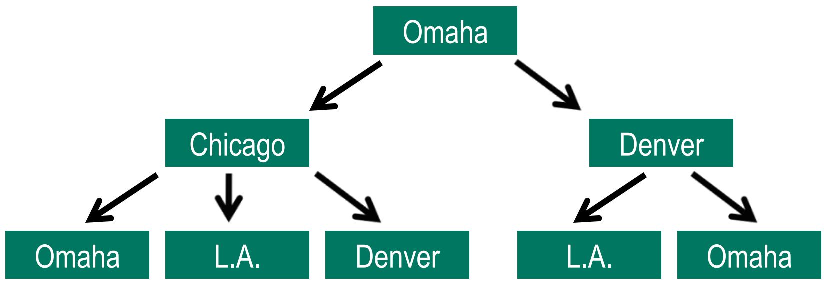
- will have a field to store the adjacency graph
- first try: simple depth-first search
- BASE CASE: if start == end, return [start]
- RECURSION: if can find path from a word adjacent to start, append start to that path

```
public class PathFinder<E> {
    private AdjacencyGraph<E> graph;

    public PathFinder(AdjacencyGraph<E> graph) {
        this.graph = graph;
    }


    public List<E> findDepth1(E startItem, E endItem) {
        if (startItem.equals(endItem)) {
            List<E> startPath = new ArrayList<E>();
            startPath.add(startItem);
            return startPath;
        }
        else {
            for (E adjItem : this.graph.adjacencies(startItem)) {
                List<E> restPath = findDepth1(adjItem, endItem);
                if (restPath != null) {
                    restPath.add(0, startItem);
                    return restPath;
                }
            }
            return null;
        }
    }

  …
}
```

DFS and cycles

since DFS moves blindly down one path, cycles are a SERIOUS problem

- what if Omaha was listed before Denver in the Chicago flights?



it usually pays to test for cycles:

- if you know the path so far, check each new node/state before extending
- if node/state is already on the path, abandon the path and try a different action/edge

13

Depth first search with cycle checking

```
public List<E> findDepth2(E startItem, E endItem) {
    List<E> path = new ArrayList<E>();
    path.add(startItem);
    if (this.findDepth2(path, endItem)) {
        return path;
    }
    else {
        return null;
    }
}

private boolean findDepth2(List<E> pathSoFar, E endItem) {
    E lastItemSoFar = pathSoFar.get(pathSoFar.size()-1);
    if (lastItemSoFar.equals(endItem)) {
        return true;
    }
    else {
        for (E adjItem : this.graph.adjacencies(lastItemSoFar)) {
            if (!pathSoFar.contains(adjItem)) {
                pathSoFar.add(adjItem);
                if (findDepth2(pathSoFar, endItem)) {
                    return true;
                }
                else {
                    pathSoFar.remove(pathSoFar.size()-1);
                }
            }
        }
        return false;
    }
}
```

pass the partial ladder along the recursion

- 1st parameter is the ladder so far (initially contains just the starting word)
- 2nd parameter is the ending word

since have the path, can check for cycle before adding

14

Breadth vs. depth

even with cycle checking, DFS may not find the shortest solution

- if state space is infinite, might not find solution at all

breadth first search (BFS)

- extend the search one level at a time
  - i.e., from the start state (root), try every possible action/edge (& remember them all)
    - if don't reach goal, then try every possible action/edge from those nodes/states
    - . . .
- requires keeping a list of partially expanded search paths
- ensure breadth by treating the list as a queue
  - when want to expand shortest path: take off front, extend & add to back

```
[ [Omaha] ]

[ [Omaha, Chicago], [Omaha, Denver] ]

[ [Omaha, Denver], [Omaha, Chicago, Omaha], [Omaha, Chicago, LA], [Omaha, Chicago,
  Denver] ]

[ [Omaha, Chicago, Omaha], [Omaha, Chicago, LA], [Omaha, Chicago, Denver], [Omaha, Denver,
  LA], [Omaha, Denver, Omaha] ]

[ [Omaha, Chicago, LA], [Omaha, Chicago, Denver], [Omaha, Denver, LA], [Omaha, Denver,
  Omaha], [Omaha, Chicago, Omaha, Chicago], [Omaha, Chicago, Omaha, LA] ]
```

15

Breadth first search

BFS is trickier to code since you must maintain an ordered queue of all paths currently being considered

```
public List<E> findBreadth(E startItem, E endItem) {
    Queue< List<E> > pathQ = new LinkedList< List<E> >();

    List<E> startPath = new ArrayList<E>();
    startPath.add(startItem);
    pathQ.add(startPath);

    while (!pathQ.isEmpty()) {
        List<E> shortestPath = pathQ.remove();

        E lastItem = shortestPath.get(shortestPath.size()-1);
        if (lastItem.equals(endItem)) {
            return shortestPath;
        }
        else {
            for (E adjItem : this.graph.adjacencies(lastItem)) {
                List<E> copy = new ArrayList<E>();
                copy.addAll(shortestPath);
                copy.add(adjItem);
                pathQ.add(copy);
            }
        }
    }
    return null;
}
```

16

Breadth first search w/ cycle checking

as before, can add cycle checking to avoid wasted search

- don't extend path if new state already occurs on the path

WILL CYCLE CHECKING AFFECT THE ANSWER FOR BFS?

IF NOT, WHAT PURPOSE DOES IT SERVE?

```
[ [Omaha] ]

[ [Omaha, Chicago], [Omaha, Denver] ]

[ [Omaha, Denver], [Omaha, Chicago, LA], [Omaha, Chicago, Denver] ]

[ [Omaha, Chicago, LA], [Omaha, Chicago, Denver], [Omaha, Denver, LA] ]

[ [Omaha, Chicago, LA], [Omaha, Chicago, Denver], [Omaha, Denver, LA], [Omaha, Chicago,
  Omaha, LA] ]
```

17

Breadth first search w/ cycle checking

again, since you have the partial ladder stored, it is easy to check for cycles

- can greatly reduce the number of paths stored, but does not change the answer

```
public List<E> findBreadth(E startItem, E endItem) {
       Queue< List<E> > pathQ = new LinkedList< List<E> >();

       List<E> startPath = new ArrayList<E>();
       startPath.add(startItem);
       pathQ.add(startPath);

       while (!pathQ.isEmpty()) {
           List<E> shortestPath = pathQ.remove();

           E lastItem = shortestPath.get(shortestPath.size()-1);
           if (lastItem.equals(endItem)) {
               return shortestPath;
           }
           else {
               for (E adjItem : this.graph.adjacencies(lastItem)) {
                   if (!shortestPath.contains(adjItem)) {
                       List<E> copy = new ArrayList<E>();
                       copy.addAll(shortestPath);
                       copy.add(adjItem);
                       pathQ.add(copy);
                   }
               }
           }
       }
       return null;
   }
```

18

Transform & conquer twist

can further optimize breadth-first search if all you care about is the shortest ladder

CLAIM: as soon as an item has been used in some path, you can disregard it for all future paths

JUSTIFICATION?

how much difference would this have on word ladder program?

19

Breadth first search w/ no reuse

```
public List<E> findBreadth(E startItem, E endItem) {
       Queue< List<E> > pathQ = new LinkedList< List<E> >();

       List<E> startPath = new ArrayList<E>();
       startPath.add(startItem);
       pathQ.add(startPath);

       HashSet<E> usedItems = new HashSet<E>();
       usedItems.add(startItem);

       while (!pathQ.isEmpty()) {
           List<E> shortestPath = pathQ.remove();

           E lastItem = shortestPath.get(shortestPath.size()-1);
           if (lastItem.equals(endItem)) {
               return shortestPath;
           }
           else {
               for (E adjItem : this.graph.adjacencies(lastItem)) {
                   if (!usedItems.contains(adjItem)) {
                       List<E> copy = new ArrayList<E>();
                       copy.addAll(shortestPath);
                       copy.add(adjItem);
                       pathQ.add(copy);

                       usedItems.add(adjItem);
                   }
               }
           }
       }
       return null;
   }
```

20

DFS vs. BFS

Advantages of DFS

- requires less memory than BFS since only need to remember the current path
- if lucky, can find a solution without examining much of the state space
- with cycle-checking, looping can be avoided

Advantages of BFS

- guaranteed to find a solution if one exists – in addition, finds optimal (shortest) solution first
- will not get lost in a blind alley (i.e., does not require backtracking or cycle-checking)
- can add cycle-checking or reuse-checking to reduce wasted search

note: just because BFS finds the optimal *solution*, it does not necessarily mean that it is the optimal *control strategy*!