csci 210: Data Structures

Stacks and Queues in Solution Searching
Summary

Topics
- Using Stacks and Queues in searching
- Applications:
  - In-class problem: missionary and cannibals
  - In-class problem: finding way out of a maze
- Searching a solution space: Depth-first and breadth-first search (DFS, BFS)

READING:
- GT textbook chapter 5
Searching in a Solution Space

- Remember the problems:
  - Permutations: Write a function to print all permutations of a given string.
  - Subsets: Write a function to enumerate all subsets of a given string
  - Subset sum: Given an array of numbers and a target value, find whether there exists a subset of those numbers that sum up to the target value.

- We saw how to solve them recursively.
  - Idea: A recursive solution takes as parameters the partial solution so far. Given this partial solution, it finds all possible ways to build new solutions.
void recPermute(String soFar, String remaining) {

    //base case
    if (remaining.length() == 0)
        System.out.println(soFar);
    else {
        for (int i=0; i< remaining.length(); i++) {
            String nextSoFar = soFar + remaining[i];
            String nextRemaining = remaining.substring(0,i) +
                                   remaining.substring(i+1);
            recPermute(nextSoFar, nextRemaining)
        }
    }
}
Tree of recursive calls

```
  "", abc
   /   \
  /     \ 
 a, bc  b, ac  c, ab
   /     \
  /       \ 
 ab, c  ac, b ba, c bc, a ca, b cb, a
   /     \
  /       \ 
 abc     acb bac bca cab cba
```
Searching in a Solution Space

Permutations: Write a function to print all permutations of a given string.

Subsets: Write a function to enumerate all subsets of a given string.

Subset sum: Given an array of numbers and a target value, find whether there exists a subset of those numbers that sum up to the target value.

We saw how to solve them recursively.
• Idea: A recursive solution takes as parameters the partial solution so far. Given this partial solution, it finds all possible ways to build new solutions.

Another way to look at it:
• let S = the set of all possible partial solutions so far.
  • e.g. S = \{a, b, c, d\} //all possible partial solutions of one letter
  • for each partial solution p in S
    • move one step forward and find all possible next solutions from p. Add all these to a new set S’.
    • e.g. partial solution p = “a” gives 3 new solutions: “ab”, “ac”, “ad”
• repeat with S = S’
  • e.g. S’ = \{ab, ac, ad, ba, bc, bd, ca, cb, cd, da, db, dc\}
Permutations

**Recursive permute:**
- recPermute(soFar, remaining)
- the function knows about the “current” partial solution
- the system keeps track of the active calls---the tree of recursive calls corresponds to all partial solutions

```
S = {“”}
S = {a, b, c}
S = {ab, ac, bc, ba, cb, ca}
S = {abc, acb, bca, bac, cba, cab}
```

**Non-recursive permute**
- construct explicitly the set of partial solutions

```
S = {“”}
S = {a, b, c}
S = {ab, ac, bc, ba, cb, ca}
S = {abc, acb, bca, bac, cba, cab}
```
Building a Solution

- Imagine that we encode the partial solution to a problem in some way
  - for e.g. for permutations a partial solution could be a tuple \( s = \langle \text{soFar}, \text{remaining} \rangle \)

- //\( S \) denotes the set of partial solutions
- \( S = \) empty set
- //create the initial state
- \( S = \{ \text{initial-state} \} \)
- while \( S \) is not empty
  - \( S' = \{ \} \)
  - go through all partial solution \( s \) from \( S \)
  - for each \( s \) generate all possible next solutions from \( s \) and add them to \( S' \)
  - \( S = S' \)

- Think of \( S \) as the (partial) solution space. Our algorithm will construct it.
Building a Solution

- We do not need both S and S’
- Think of S as the (partial) solution space. Our algorithm will construct it.

- S = empty set
- //create the initial state
- S = { initial-state}
- while S is not empty
  - delete the next partial solution s from S
  - generate all possible next solutions from s and add them to S
The solution space

- Each solution is a state
- Each solution generates new solutions

```
  “”, abc
    /   \
   /     \ 
```

```
  a, bc
    /   \   /
   /     \ /  
  /       /   
 a, c  ac, b
```

```
  b, ac
    /   \   /
   /     \ /  
  /       /   
 ba, c  bc, a
```

```
  c, ab
    /   \   /
   /     \ /  
  /       /   
 ca, b  cb, a
```

```
  abc
  acb
  bac
  bca
  cab
  cba
```
Building a Solution

- Think of $S$ as the solution space. Our algorithm will construct it.

- $S = \text{empty set}$
- //create the initial state
- $S = \{ \text{initial-state} \}$
- while $S$ is not empty
  - delete the next partial solution $s$ from $S$
  - generate all possible next solutions from $s$ and add them to $S$

- $S$ is a set of states. How to store $S$?
  - Keep $S$ as a queue
    - delete next solution from the front
    - add new solutions to the end of queue
  - Keep $S$ as a stack
    - delete next solution from the top
    - add new solutions to the top
• S = empty set
• //create the initial state
• S = { initial-state}
• while S is not empty
  • delete the next partial solution s from S
  • generate all possible next solutions from s and add them to S

S as a queue
• S = { "", "abc" }
• partial solution s = "", abc generates 3 new solutions <a, bc>, <b, ac>, <c, ab>
• they are all put in S: S = {<a, bc>, <b, ac>, <c, ab>}
• partial solution s=<a,bc> generates 2 new solutions <ab,c> and <ac,b>; they are put in S
• S = { <b, ac>, <c, ab>, <ab,c>, <ac,b> }
• S = { <c, ab>, <ab,c>, <ac,b>, <ba,c>, <bc, a> }
• S = { <ab,c>, <ac,b>, <ba,c>, <bc, a>, <ca,b>, <cb, a> }
• ...
• S = { <abc,"">, <acb,"">, <bac,"">, <bca,"">, <cab,"">, <cba,""> }
• S = { }

How does the algorithm traverse and construct the solution space when S is a queue?
• S = empty set
• //create the initial state
• S = { initial-state}
• while S is not empty
  • delete the next partial solution s from S
  • generate all possible next solutions from s and add them to S

• S as a stack
  • S = {<"", "abc">}
  • partial solution s = <"", abc> generates 3 new solutions <a, bc>, <b, ac>, <c, ab>
  • they are all put in S: S = {<c, ab>, <b, ac>, <a, bc>}
  • partial solution s=<c,ab> generates 2 new solutions <ca,b> and <cb,a>; they are put in S
  • S = {<cb, a>, <ca, b>, <b, ac>, <a, bc>}
  • ...
How does the algorithm traverse and construct the solution space when S is a stack?
The solution space

- Using a stack mimics recursion ----- goes depth first
  - depth-first search (DFS)
- Using a queue goes level by level ----- goes breadth first
  - breadth-first search (BFS)
Example: The missionary and cannibal problem

- You have 3 missionaries, 3 cannibals and a boat sitting on, say, the left side of a river.
- They all need to cross to the other side.
- Find a set of moves that brings all 6 people on the other side safely.
  - The boat can take at most two people at a time (and at least one).
  - Anybody can row
  - If at any point there are more cannibals than missionaries, the missionaries get eaten.
Missionaries and Cannibals

- We want to frame it as a search in a solution space and use the previous skeleton

- How to encode a state?
  - write a class MCState

- What’s the initial state?

- What’s the final state?
  - write MCState:isFinal()

- When is a state valid?
  - write MCState:isValid()

- Given a state, what are the moves you can make?

- What will the set S contain?
Missionaries and Cannibals

- Queue<MCState> s = new Queue<MCState>();
- //add initial state
- s.insert(new MCState(3,3,0,0,1));
- while (!s.isEmpty()) {
  - MCState crt = s.delete();
  - if (crt.isFinal()) { //this is the goal state; break;}
  - //generate all possible next states and call s.insert() to add them to s
  - ...
- }
- //crt must be the final state; print it

- Are there duplicate states in S?
- Can a state be inserted in S several times? (This would correspond to a loop --- we go back to a state that we already explored). Why is this not a problem?
- The skeleton above uses a Queue for S. Would a Stack work? Why (not)?