CS 106B
Lecture 9: Recursive Backtracking 1: Decision Trees
Friday, April 21, 2017

reading:
Programming Abstractions in C++, Chapter 8.2-8.3
Today's Topics

• Logistics:
  • If your Fractals or GrammarSolver program won't run, see the comment at the bottom of Piazza @317
  • YEAH hours: 0% attendance
  • Pair programming? What is it?

• Recursion and Decision Trees
  • Folders and Directories
  • Reducible Words

• Recursive Backtracking: Exhaustive Search
  • Permutations
More Recursion!

• So far, you might be thinking to yourself: *why do I need recursion, when I can solve lots of problems using simple loops?*

• Example: A factorial is a recursively defined number:

\[ n! = n \times (n-1)!, \text{ where } 1! = 1 \]

\[
4! \\
= 4 \times 3! \\
= 4 \times 3 \times 2! \\
= 3 \times 2 \times 1! \\
= 3 \times 2 \times 1 \\
= 24
\]
More Recursion!

• Let's write the factorial function recursively

\[ n! = n \times (n-1)!, \text{ where } 1! = 1 \]

```c
long factorial(long n) {
}
```
Let's write the factorial function recursively

\[ n! = n \times (n-1)!, \text{ where } 1! = 1 \]

```c
long factorial(long n) {
    // base case
    if (n == 1) {
        return 1;
    }
    // recursive case
    return n * factorial(n-1);
}
```
More Recursion!

• But wait...we could have just written this iteratively, using a loop!

\[ n! = n \times (n-1)!, \text{ where } 1! = 1 \]

```c
long factorial(long n) {
}
```
More Recursion!

But wait...we could have just written this iteratively, using a loop!

\[ n! = n \times (n-1)!, \text{ where } 1! = 1 \]

```cpp
long factorial(long n) {
  long answer = 1;
  while (n > 1) {
    answer *= n;
    n--;
  }
  return answer;
}
```
• These relatively easy recursive problems may have beautiful solutions, but there isn't anything special about solving the problem recursively.

• Today, we will discuss problems that deal with "iterative branching" -- and it is these problems that demonstrate the power of a recursive solution.

• Let's go!
Recursion and Decision Trees

• The following is a graphical depiction of the files in a folder on my computer:

• The top-level folder is called "ExampleFolder", and it has three children folders, called "child1", "child2", and "child3".

• child1 has two files, "i_dont_wanna_grow_up.doc" and "kid_stuff.txt"

• etc.
Let's re-draw that structure a bit, into a "tree" format.
Recursion and Decision Trees

If we flip it over...there is a root at the bottom and leaves where there are no more branches.
Recursion and Decision Trees

Flipped back, this is what we call a tree in computer science.
A folder is just a recursive container!

- A folder is a tree!
A folder is just a recursive container!

• All children are also complete trees!
A folder is just a recursive container!

• All children are also complete trees!
A folder is just a recursive container!

- All children are also complete trees!
A folder is just a recursive container!

- All children are also complete trees!
Let's write a program to output all files in a folder

• All children are also complete trees!
Here is a word puzzle: "Is there a nine-letter English word that can be reduced to a single-letter word one letter at a time by removing letters, leaving a legal word at each step?"
Another Example: Reducible Words

4-letter example:

cart ➠ art ➠ at ➠ a

can you think of a nine letter word?
Another Example: Reducible Words

startling

is there really just one nine-letter word with this property?
All Reducible 9-letter words

can we do this iteratively?

it would be very messy!
All Reducible 9-letter words

can we do this recursively?

yes!

what is the decision tree?
Reducability Decision Tree

cart

art ———— art

crt

cat

car
Reducability Decision Tree
Reducability Decision Tree

cart

art

rt

tr

rt

at

ar

rt

crt

rt

cr

ct

cat

at

cr

ca

at

car

ar

ca

cr

ac

ra

rc

ac
Reducability Decision Tree
Reducability Decision Tree
bool search(currentState) {
    if (isSolution(currentState)) {
        return true;
    } else {
        for (option : moves from currentState) {
            nextState = takeOption(curr, option);
            if (search(nextState)) {
                return true;
            }
        }
        return false;
    }
}
Reducible Word

Let's define a reducible word as a word that can be reduced down to one letter by removing one character at a time, leaving a word at each step.

- **Base case:**
  - A one letter word in the dictionary.

- **Recursive Step:**
  - Any multi-letter word is reducible if you can remove a letter (legal move) to form a shrinkable word.
How the algorithm works

art: is a word
How the algorithm works

rt: not a word
How the algorithm works

at: is a word
How the algorithm works

t: not a word
How the algorithm works

a: is a word
there is a solution!
How the algorithm works

a: is a word
there is a solution!
How the algorithm works

a: is a word
there is a solution!
How the algorithm works

art

crt

cat

car

rt at ar rt ct cr at ct ca ar cr ca

t r t c r t a t c a c r a

a: is a word
there is a solution!
Is there really just one nine-letter word?
There are basically five different problems you might see that will require recursive backtracking:

- Determine whether a solution exists
- Find a solution
- Find the best solution
- Count the number of solutions
- Print/find all the solutions
• Since 1954, the JUMBLE has been a staple in newspapers.
• The basic idea is to unscramble the anagrams for the words on the left, and then use the letters in the circles as another anagram to unscramble to answer the pun in the comic.
• As a kid, I played the puzzle every day, but some days I just couldn't descramble the words. Six letter words have $6! = 720$ combinations, which can be tricky!
• I figured I would write a computer program to print out all the permutations!
Since 1954, the JUMBLE has been a staple in newspapers.

The basic idea is to unscramble the anagrams for the words on the left, and then use the letters in the circles as another anagram to unscramble to answer the pun in the comic.

As a kid, I played the puzzle every day, but some days I just couldn’t descramble the words. Six letter words have 6! == 720 combinations, which can be tricky!

I figured I would write a computer program to print out all the permutations!
void permute4(string s) {
    for (int i = 0; i < 4; i++) {
        for (int j = 0; j < 4; j++) {
            if (j == i) {
                continue; // ignore
            }
            for (int k = 0; k < 4; k++) {
                if (k == j || k == i) {
                    continue; // ignore
                }
                for (int w = 0; w < 4; w++) {
                    if (w == k || w == j || w == i) {
                        continue; // ignore
                    }
                    cout << s[i] << s[j] << s[k] << s[w] << endl;
                }
            }
        }
    }
}
void permute5(string s) {
    for (int i = 0; i < 5; i++) {
        for (int j = 0; j < 5; j++) {
            if (j == i) {
                continue; // ignore
            }
            for (int k = 0; k < 5; k++) {
                if (k == j || k == i) {
                    continue; // ignore
                }
                for (int w = 0; w < 5; w++) {
                    if (w == k || w == j || w == i) {
                        continue; // ignore
                    }
                    for (int x = 0; x < 5; x++) {
                        if (x == k || x == j || x == i || x == w) {
                            continue;
                        }
                        cout << s[i] << s[j] << s[k] << s[w] << s[x] << endl;
                    }
                }
            }
        }
    }
}
Permutations

And a permute6() function...

```cpp
void permute6(string s) {
    for (int i = 0; i < 5; i++) {
        for (int j = 0; j < 5; j++) {
            if (j == i) {
                continue; // ignore
            }
            for (int k = 0; k < 5; k++) {
                if (k == j || k == i) {
                    continue; // ignore
                }
                for (int w = 0; w < 5; w++) {
                    if (w == k || w == j || w == i) {
                        continue; // ignore
                    }
                    for (int x = 0; x < 5; x++) {
                        if (x == k || x == j || x == i || x == w) {
                            continue;
                        }
                        for (int y = 0; y < 6; y++) {
                            if (y == k || y == j || y == i || y == w || y == x) {
                                continue;
                            } else {
                                cout << "  " << s[i] << s[j] << s[k] << s[w] << s[x] << s[y] << endl;
                            }
                        }
                    }
                }
            }
        }
    }
}
```

This is not tenable!

What has been seen cannot be un-seen
Permutations do not lend themselves well to iterative looping because we are really *rearranging* the letters, which doesn't follow an iterative pattern.

Instead, we can look at a recursive method to do the rearranging, called an *exhaustive algorithm*. We want to investigate all possible solutions. We don't need to know how many letters there are in advance!

In pseudocode:

```plaintext
If you have no more characters left to rearrange, print current permutation
for (every possible choice among the characters left to rearrange) {
    Make a choice and add that character to the permutation so far
    Use recursion to rearrange the remaining letters
}
```

In English:

- The permutation starts with zero characters, as we have all the letters in the original string to arrange. The base case is that there are no more letters to arrange.
- Take one letter from the letters left, add it to the current permutation, and recursively continue the process, decreasing the characters left by one.
The algorithm in C++:

```c++
void permute(string soFar, string rest) {
    if (rest == "") {
        cout << soFar << endl;
    } else {
        for (int i = 0; i < rest.length(); i++) {
            string remaining = rest.substr(0, i) + rest.substr(i+1);
            permute(soFar + rest[i], remaining);
        }
    }
}
```

Example call:

- `recPermute("", "abcd");`
This is a tree!

✓ Exhaustive
✓ Works for any length string
✓ \( N! \) different results
✓ Can think of this as a "call tree" or a "decision tree"
Here is the algorithm again:

```cpp
void permute(string soFar, string rest) {
    if (rest == "") {
        cout << soFar << endl;
    } else {
        for (int i = 0; i < rest.length(); i++) {
            string remaining = rest.substr(0, i) + rest.substr(i+1);
            permute(soFar + rest[i], remaining);
        }
    }
}
```

Some might argue that this isn't a particularly good function, because it requires the user to always start the algorithm with the empty string for the `soFar` parameter. It's ugly, and it exposes our internal parameter.

What we really want is a `permute(string s)` function that is cleaner.

We can re-name the function above `permuteHelper()` (and change the inner call, as well!), and have a cleaner permute function that calls this one.
The cleaner interface:

```cpp
void permuteHelper(string soFar, string rest) {
    if (rest == "") {
        cout << soFar << endl;
    } else {
        for (int i = 0; i < rest.length(); i++) {
            string remaining = rest.substr(0, i) + rest.substr(i+1);
            permuteHelper(soFar + rest[i], remaining);
        }
    }
}

void permute(string s) {
    permuteHelper("", s);
}
```

Now, a user only has to call `permute("tuvedo")`, which hides the helper recursion parameter.
References and Advanced Reading

• References:
  • Understanding permutations: http://stackoverflow.com/questions/7537791/understanding-recursion-to-generate-permutations
  • Maze algorithms: https://en.wikipedia.org/wiki/Maze_solving_algorithm

• Advanced Reading:
  • Exhaustive recursive backtracking: https://see.stanford.edu/materials/icspacs106b/h19-recbacktrackexamples.pdf
  • Backtracking: https://en.wikipedia.org/wiki/Backtracking