CSE 143

Computer Programming II

Recursive Backtracking
Recursive Backtracking
Recursive Backtracking

Outline

Recursive Backtracking 1

Definition (Recursive Backtracking)

Recursive Backtracking is an attempt to find solution(s) by building up partial solutions and abandoning them if they don't work.

Recursive Backtracking Strategy

- If we found a solution, stop looking (e.g. return)
- Otherwise for each possible choice c...
  - Make the choice c
  - Recursively continue to make choices
  - Un-make the choice c (if we got back here, it means we need to continue looking)

Client vs. Implementor, again 2

You may have noticed that many of the class examples I've been showing involve me using a class that I've already written. There are several reasons for this:

- Learning to read and use an API is a really important programming skill
- Switching between the client and implementor views is an important goal of this course
- The code I write is usually easy, but really tedious (so, it would be a waste of time to write in class)

Take-Away

Every time I print out an API for you, you should try to understand it from the comments. This will help you on the homework, on exams, and in any future programming endeavors.

`BooleanExpression` 3

Today's API is `BooleanExpression`.

What is a `BooleanExpression`?

The `BooleanExpression` class allows us to represent the conditions we write in `if` statements. For instance, to represent the following:

```
1 if (!(queue.size() > 0) && queue.peek() > 5) {
2     ...
3 }
```

We would do

```
new BooleanExpression("(!a && b)");
```

Notice that we use single letter variable names instead of `queue.size() > 0`. This is a simplification for implementation.

Evaluating `BooleanExpression` 4

Evaluating `BooleanExpressions`

Remember when we took $(1+2)*3$ and evaluated it to 9 recursively?
We can do a similar thing for `BooleanExpressions`:
Consider the `BooleanExpression` from above:

```
"(!a && b)"
```

Suppose we know the following:

- a is true.
- b is false.

What does this expression evaluate to?

`(!a && b)` ⟶ `(!true && false)` ⟶ `(false && false)` ⟶ `false`

Suppose we wanted to write a method:

```
public static boolean evaluate(BooleanExpression e, ???  assn)
```

where `assn` represents the truth values of the variables.

What type would `assn` be? It's a **mapping** from variables to truth values.

Evaluating `BooleanExpression` 5

Okay, so, we have:

```
public static boolean evaluate(BooleanExpression e,
        Map<String, Boolean> assignments)
```

Consider the following case:

`evaluate` return value?

- `e` is `a && b`
- `assignments` map is `{a=true}`.

What should `evaluate` return?

We can't answer the question. What seems like a good idea? **`null`**.

So, we change the return type to `Boolean`.

Who Should Implement `evaluate`? 6

Who Writes `evaluate`?

- The implementor of `BooleanExpression`
  ...if so, it should be inside the `BooleanExpression` class
- The client of `BooleanExpression`
  ...if so, it should be outside the `BooleanExpression` class

The implementor of `BooleanExpression` should write the method, because then all the clients can use it.

That pesky `static`...

- If the implementor writes `evaluate`, then the method signature is:

```
public Boolean evaluate(Map<String, Boolean> assn)
```

- If the client writes `evaluate`, then the method signature is:

```
public static Boolean evaluate(
    BooleanExpression e,
    Map<String, Boolean> assn
)
```

Finally, Back To Recursive Backtracking... 7

`canBeTrue`

Write a method

```
public static void canBeTrue(BooleanExpression b)
```

that returns true if it is possible for the input to to **evaluate to true** and false otherwise.

Some examples:

- `a && b` ⟶ if we have `{a=true, b=true}`, then it is true.
- `a && !a` ⟶ no matter what `a` is, this will always be false.

To do recursive backtracking, we need to answer these questions:

- What are the choices we're making incrementally?
  ... assignments of each variable to true/false
- How do we "undo" a choice?
  ... remove the assignment from the map
- What are the base case(s)?
  ... the assignment must be true/false

Uh Oh... How can we try assignments? 8

We don't have a way of passing assignments through to the function.
How can we fix this?

`public/private` pair!

Public/Private Recursive Pair

```
public static void canBeTrue(BooleanExpression b)

private static void canBeTrue(
    BooleanExpression b,
    Map<String, Boolean> m
)
```

`canBeTrue` Solution 9

```
1  public static void canBeTrue(BooleanExpression b) {
2      Map<String, Boolean> assignmentMap = new TreeMap<>();
3      canBeTrue(b, assignmentMap);
4  }
5
6  private static void canBeTrue(BooleanExpression b, Map<String, Boolean> m) {
7      Set<String> variables = b.getVariables();
8      if (variables.size() == m.keySet().size() && b.evaluate(m)) {
9          System.out.println(m);
10     }
11
12     for (String variable : variables) { // Try to assign any
13         if (!m.keySet().contains(variable)) { // variable we haven't
14             boolean[] choices = {true, false}; // already assigned.
15             for (boolean assignment : choices) {
16                 m.put(variable, assignment);
17                 canBeTrue(b, m);
18                 m.remove(variable);        // Otherwise, backtrack
19             }
20     }
21 }
```

BTW, Why does this problem matter?

Solving `canBeTrue` quickly is the **most important** open problem in Computer Science.

If you solve this problem in $\mathcal{O}(n^k)$ time for **any** k, the following happen:

- You get **one million** dollars.
- You get a PhD.
- You become the most famous Computer Scientist, pretty much ever
- You break all banks, credit cards, website encryption, etc.