University of Alberta
Computing Science Department
Program Synthesis in XAI - W21

Top-Down Search
Levi Lelis
levi.lelis@ualberta.ca

1 Top-Down Search

Let us consider the following DSL as our running example.

$$S \rightarrow x \mid y \mid (S + S) \mid \text{if } B \text{ then } S \text{ else } S$$
$$B \rightarrow (S \leq S) \mid (S == S) \mid (S \geq S)$$

The specification we consider in this example is the set $\{(x = 0, y = 1, out = 1), (x = 1, y = 2, out = 3)\}$. Here, for each pair, the tuple (e.g., (x, y)) is the input for a function $f(x, y)$ and the second value, *out*, is the desired output. A program that satisfies the specification is $x + y$. In this lecture we will study top-down search algorithms that explore the space defined by the DSL. Top-down algorithms start the search from the DSL's initial symbol and try all possible combinations given by the production rules of the DSL. The first top-down algorithm we will study is Breadth-First Search (BFS).

1.1 Breadth-First Search (BFS)

A **partial program** p is a program with non-terminal symbols; p is complete if it is not partial. All complete programs are leaf nodes in the tree expanded by top-down search algorithms. The **children** of a partial program p are all programs c that can be generated by applying a single production rule on a single non-terminal symbol from p. This way, c differs from p by a single transformation. Usually we follow a rule such as "generate the children according to the leftmost non-terminal symbol of p." Choosing the leftmost or rightmost is arbitrary and implementation-dependent. While implementing top-down search we implement a function for returning all children of a partial program. This function is often referred to as the **successor function** or simply as the **children function**. A program p is **expanded** when its children are generated.

```
1 BFS(CFG G, Specification T)
2   _open.push(G.start_symbol) # priority queue with programs to be explored
3   while _open is not empty:
4     p = _open.pop()
5     children = G.children(p)
6     for c in children:
7       if is_complete(c) and is_solution(c, T): # c is complete and is a solution
8         return c
9       _open.push(c) # add to the open list
```

The algorithm is **complete** as it is guaranteed to find a solution if one exists. If the priority queue OPEN is sorted from the shortest to the longest program. The length of a program p is equivalent to the size of its AST and to the number of production rules applied to the initial symbol to obtain p. For example, program x requires only one rule to be applied from the initial symbol, while program $x + y$ requires three rules to be applied from the initial symbol $S \rightarrow (S + S) \rightarrow (x + S) \rightarrow (x + y)$. The algorithm is **optimal** because it finds the shortest program that satisfies the input-output pairs.

Here is a BFS execution example with a simplified grammar and specification $\{(x = 0, y = 1, out = 1), (x = 1, y = 2, out = 3)\}$.

$$S \to x \mid y \mid S + S$$

p	OPEN
	S
S	x, y, S + S
x	y, S + S
y	S + S
S + S	x + S, y + S, S + S + S
x + S	y + S, S + S + S, x + x, x + S + S, x + y

The search can stop as soon as program $x + y$ is generated, as it satisfies all the input-output pairs.

1.1.1 Equivalence and Duplicate Detection

BFS can be equipped with a hash table to store all programs visited during search. That way, every program p is only expanded once during search. This can potentially incur in major computational savings as each partial program p can root large subtrees. Often, however, the DSLs are free of duplicated programs. The DSL used above does not generate duplicated programs if we use a rule such as "generate the children of each program p by applying all possible production rules to p's leftmost non-terminal symbol." Other DSLs might allow duplicated programs. Let's consider the following example.

$$\begin{aligned} S &\to E + E \mid D + D \\ E &\to T \mid \cdots \\ D &\to T \mid \cdots \\ T &\to \cdots \end{aligned}$$

Using the DSL above we might generate the partial program $T + T$ following either the branch $E + E$ or $D + D$. If the DSL has a large list of production rules for T, then partial program $T + T$ can root a large subtree and expanding it only once might save a large number of programs evaluated. Note that the DSL above could be simplified to remove unit productions (e.g., $E \to T$) and that would solve the problem of generating duplicated programs. Unambiguous DSLs are duplication-free, but the verification of ambiguity is undecidable.

Equivalence detection isn't of help for top-down search as it is for bottom-up search. This is because it is expensive to verify if two partial programs are equivalent. In a top-down search, if we evaluate a complete program p and discover that p is equivalent to another program evaluated prior to p, then we already paid the computational cost of generating p. This is because p is complete and thus is a leaf in the search tree.

1.1.2 Memory Complexity

One of the key disadvantages of BFS is its memory requirements (this is also true for bottom-up search). Let's first look at the version that stores all programs encountered during search and performs duplicate detection.

If b is the number of children generated by each partial program considered during search and if the solution is found with length d, then the algorithm's memory complexity is given by $1+b+b^2+b^3+\cdots+b^d = O(b^d)$. The time complexity is also $O(b^d)$ by the same reasoning. The version that only stores the programs in OPEN have memory complexity of b^d and has identical time complexity. Depending on the problem domain the memory complexity can be a showstopper for BFS.

1.2 Depth-First Search

One way of dealing with the memory complexity of BFS is by performing a depth-first search (DFS). DFS "dives" into the space of possible programs generating longer programs in every iteration of the algorithm. If DFS finds a solution, then it will have evaluated a number of programs equal to the length d of the program. Moreover, the DFS only has to store d sub-programs in memory before finding the solution.

However, this approach is problematic because it is not complete nor optimal. For example, the following infinity sequence is possible with DFS. $S \to (S+S) \to (S+S+S) \to (S+S+S+S) \to \cdots$. If following this path, DFS will never terminate, even if a solution exists.

1.2.1 Iterative Deepening Depth-First Search

The solution to DFS not being complete nor optimal is to perform an iterative deepening search. In the first iteration we will allow the application of at most one rule. If a solution isn't found, then we repeat the search and allow for the application of two rules. This process is repeated until a solution is found.

```
1 IDDFS(CFG G, Specification T)
2   d = 1
3   while True:
4     if not DFS(G.start_symbol, G, T, d):
5       d += 1
```

```
1 DFS(Program p, CFG G, Specification T, Depth d)
2   if d == 0:
3     return False
4   children = G.children(p)
5   for c in children:
6     if is_complete(c) and is_solution(c, T):
7       print(c)
8       return True
9     if DFS(c, G, T, d - 1):
10      return True
11  return False
```

IDDFS will evaluate the following programs in its first iteration with $d = 1$: $x, y, (S+S)$. Since none of them satisfies the specification, IDDFS increases d to 2 and now evaluates: $x, y, (S+S), (x+S), (y+S), (S+S+S)$. Not finding a solution, it increases d to 3, thus evaluating: $x, y, (S+S), (x+S), (y+S), (S+S+S), (x+x), (x+S+S), (x+y)$ and returning $x+y$ as a solution.

IDDFS is **complete** and **optimal**. The algorithm is complete because it analyses all possible programs reachable from the initial symbol. It is optimal because it searches over all programs of length N before

searching over programs of length $N + 1$.

The memory complexity of DFS is $O(d)$, where d is the size of the solution. This is because IDDFS stores only the path from the initial symbol to the program being evaluated.

What about the time complexity? How much does the iterative deepening approach hurts the algorithm's performance? The number of programs evaluated by the search can be computed as follows. The children of the initial symbol is generated d times. The grandchildren of the initial symbol is generated $d - 1$ times, and so on. This gives us the following.

$$bd + b^2(d-1) + b^3(d-2) + \cdots + b^d = O(b^d)$$

The last iteration of the algorithm dominates the time complexity of IDDFS. In terms of big-Oh notation IDDFS and BFS performs the same amount of work.

2 Research Project Ideas

1. It isn't clear how often duplicated programs appear in practice. I suspect that they are very rare and hash tables for verifying duplicated programs aren't needed in top-down search. An interesting project is to run existing top-down solvers from the Sygus Competition to verify how often duplicated programs appear during search.

2. Compare BFS, IDDFS, and Bottom-Up search in problems from the Sygus Competition, which one performs best?

3. How to create a memory-friendly version of BUS? Perhaps a variant of Beam Search might work, as we are able to keep in memory a set of programs to be combined from one iteration to the next while reducing the memory requirements.