The Lock Design Document

Stephen Rout

Introduction

This program is an experiment in solving combinatoric problems, and different methods of tree traversal. It centers around a “lock” object, called TheLock, which can be opened by a certain ordered combination of inputs. This program uses three different approaches to finding those inputs: a breadth-first algorithm, a depth-first one, and an iterative depth-first one. The program also records some information about the runs, allowing the different methods to be compared.

Specifications

Our lock is provided by the TheLock class, which hashes a string to generate a lock which can be opened by some combination of actions. In this case, there are four possible actions; shakeIt(), pullIt(), twistIt(), and pokeIt(). Any given TheLock can be opened by some ordered combination of these 4 actions. I can also create a lock while specifying the length of its answer, which is useful for testing.

Since I have no idea what combination of actions is needed, I must iterate through all possible combinations. There are many ways to approach this, but for this project I focus on 3 different techniques: breadth first, depth limited depth first, and iterative deepening.

Breadth first is an algorithm that, simply put, guarantees that all possible combinations of length *n* will be tried before any combination of length *n*+1. So, it will find, and try, all possible combinations of length 3 (such as Shake it, Pull it, Shake it) before it finds any of length 4.

Depth limited depth first search requires a stated maximum possible length for the current solution. The algorithm will then build a series of possible solutions in a depth-first way. For example, if the first solution I try has a first action of “Shake it”, then every other possible answer – of length from 1 to the stated maximum – that starts with “Shake it” will be attempted before any answer that starts with a different action. Unlike the breadth-first approach, the depth-limited algorithm requires a plausible estimate of how long the answer is, which can limit it’s utility.

Iterative deepening is a modification of depth limited search, and can be used when the length of the answer is unknown. Iterative deepening performs the same search as depth-limited, but iteratively: successively greater maximum sizes are used, guaranteeing that nothing is missed. For example, the iterative-deepening search will run for a maximum depth of 1, then of 2, then of 3. Within each iteration, it will perform an ordinary depth-first search. The iterations will continue until the answer is found.

It should be noted that this final algorithm does involve a large amount of repeated work; when it is iterated with a maximum depth of 3, the algorithm will still perform most of the node operations needed for an iterative search of depth 2. That said, due to the exponentially increasing nature of this problem, the majority of the algorithm will still be spent searching the correct depth.

Design Overview

The design can be broken down into two main categories: The individual algorithms, and the overarching structure used to represent the problem.

Structure

Structurally, I represent different actions by nodes. Each node has at least three attributes: an integer which represents the action it stores, the next node in the stack or queue, **and that node's parent node**. The parent node is key. It creates a visually convoluted data structure, but is very useful. These parent links mean that even when a node has been popped, it will remain in memory. Secondly, it lets us traverse through a set of actions in their own order, rather than being limited to the order of the stack or queue. *Figure 1* shows the tree visualization of this structure, while *Figure 2* shows a more complete, realistic diagram of the structure, as it might appear in a queue or stack. The longer lines with rectangles are the parent connections, while the shorter lines are "next" connections.

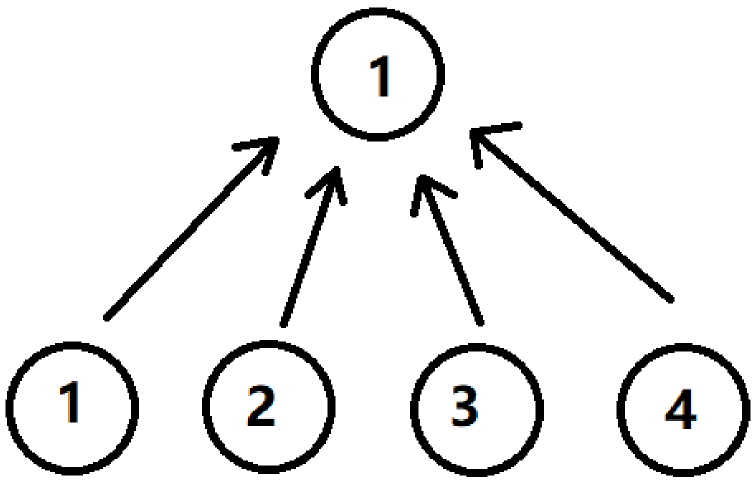

Figure 1

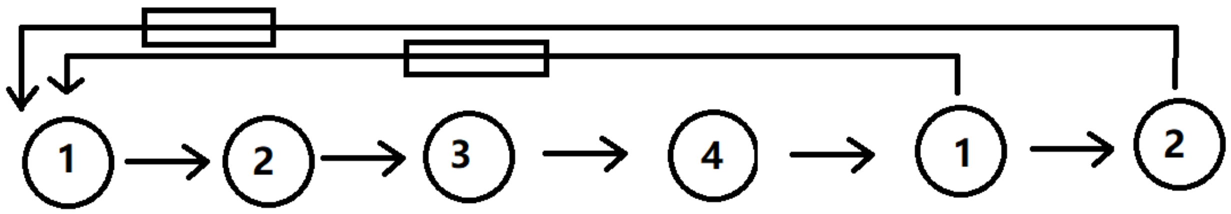

Figure 2

If *Figure 2* were expanded, then a "3" node and a "4" node would be added, both of which would have a parental link with the leftmost "1" node. The structure would then continue with an additional "1" node, which would be parentally linked with the first "2" node from the left. This pattern applies for both the queue and the stack, and will continue for the length of the structure.

This structure is critical to the unlock algorithm. To attempt unlocking a TheLock, the current node is passed into tryUnlock(). tryUnlock() then iterates from the current node to the root node, using each action on TheLock as it goes. This is aided by tryUnlockHelper(), which converts the node's integer representation of its action to a real action, which is then applied to TheLock. The result is that a sequence of actions, defined by the path between the current node and the root node, is performed on TheLock.

Algorithms

The breadth-first algorithm (found in BreadthFirstSearch) is relatively simple. An int is initialized to represent the number of nodes visited, and a queue is made. 4 "root" nodes are pushed onto the queue to start, and then the main loop begins. The basic structure is

1. Pop from the queue and increment the number of nodes visited.
2. Pass the current node into tryUnlock()
    a. tryUnlock() will perform the action of the current node, then will iterate upwards until the root node is reached, performing each action in turn.
3. Check if TheLock is now open.
    a. If it is, then some data is printed, and genAnswerString() is called to transform the sequence of integers into a human-readable string.
        i. genAnswerString() is almost identical to tryUnlockHelper, but with strings instead of methods.
    b. If it is not, 4 child nodes are generated for the current node, each of which are pushed onto the queue.
4. Repeat.

Depth-limited search (found in LimitedDepthFirstSearch) is similar, albeit with a stack instead. As with breadth-first, a structure and a counter are initialized, and four root nodes are pushed onto the stack. From there the loop is:

1. Pop from the queue and iterate the number of nodes visited.
2. Check that the current node is not null.
3. Pass the current node into tryUnlock()
4. Check if TheLock is now open.
    a. If it is, then some data is printed, and genAnswerString() is called to transform the sequence of integers into a human-readable string.
        i. genAnswerString() is almost identical to tryUnlockHelper, but with strings instead of methods.
    b. If it is not, check if the current depth is less than the maximum depth. If it is, the 4 child nodes are generated for the current node, each of which are pushed onto the stack.
5. Repeat.

It should be noted that the two algorithms use different node classes. The stack uses DLNodes, and the queue uses regular Nodes. DLNodes are similar to regular Nodes, but their nextNode and parentNode attributes are also DLNodes, and they have a depth attribute, which is used to ensure that the depth-limited algorithms stop at the appropriate time.

Iterative depth-limited search (found in IDLSearch) consists of a loop that wraps around simply depth-limited search. The only other addition is an array, which is used to move data between the programs.

Analysis

Table 1 details some statistics about the various algorithms. Three pieces of data were collected: the average number of nodes each algorithm had to visit to find the correct answer, the average size of the algorithm's queue/stack at the time of completion, and the average runtime. Each algorithm was run with solution sizes ranging from 1 to 13. The results are presented in *Table 1*.

	Breadth-first	Depth-limited	Iterative Depth-Limited
Average nodes visited	6,075,160.00	5,041,123.15	8,100,206.85
Average stack/queue	18,225,480.00	8.54	8.54
Average runtime	6,487.08	2,276.08	3,479.85

Table 1

The results have one overwhelming message: depth-first search does not seem to perform very well in this scenario. It runs significantly more slowly than the other algorithms, even with it's efficiency advantage over the iterative depth-first algorithm. It also uses a tremendous amount of memory, while the other two algorithms have essentially constant memory usage. I believe this is the reason for the breadth-first algorithm's poor time performance as well: clearly it spends more time creating new nodes than visiting them. In fact, 13 is the largest solution size the breadth-first algorithm can handle. Anything more than that produces a heap overflow.

Comparing the remaining two algorithms is more complicated. As expected, the iterative depth-limited algorithm performed slower than the regular depth-limited one, and required more computations. However, the gap is smaller than the one between depth-limited and breadth-first, and the iterative solution has the key advantage that it does not require the length of the solution to be known in advance.

When searching for an answer of unknown length, the iterative depth-limited algorithm finds the solution in about 20 minutes, exploring over 200 million nodes in the process, while the breadth-first algorithm had another heap overflow.

To summarize: if the length of the solution is known, depth-limited is by far the best performer. If the solution is not known, then the iterative depth-limited algorithm is preferable, and possibly the only algorithm capable of finding an answer at all.

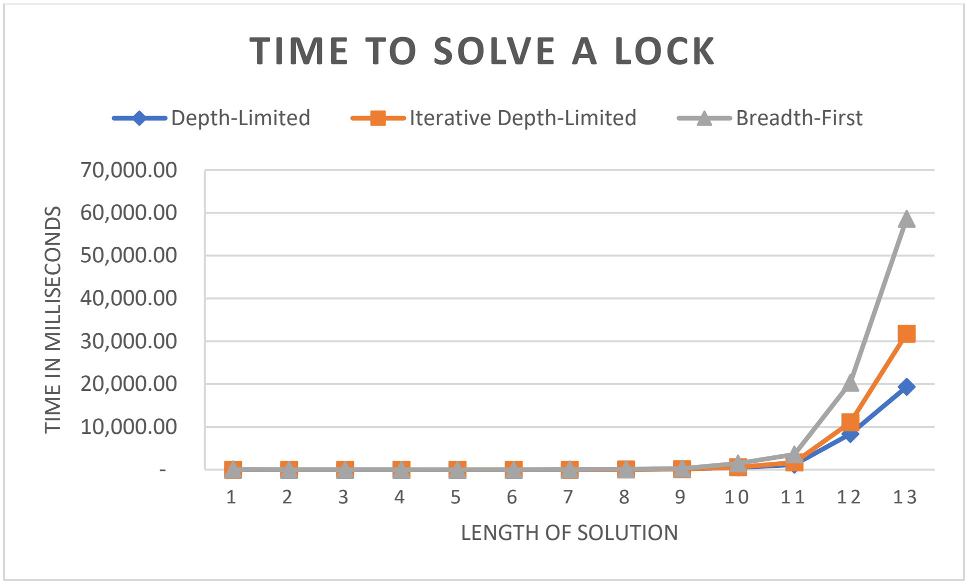

Figure 3

Figure 3 charts runtime in milliseconds against lock lengths. The chart makes it clear that, while the breadth-first algorithm is slower than the others, they all seem to be increasing in time complexity at a similar rate. The chart also drives home the fact that the difference between the depth-limited approach and the iterative-depth approach is not very large.

The single biggest improvement needed is a re-write of the depth-limited search. While the code works, it is unclear and difficult to modify. Further points for improvement include a more consistent naming system, especially in my files, greater use of inheritance in my data structures, and additional efforts to avoid redundant or nearly-redundant methods.

Figure 4 on the next page shows a UML for this program.

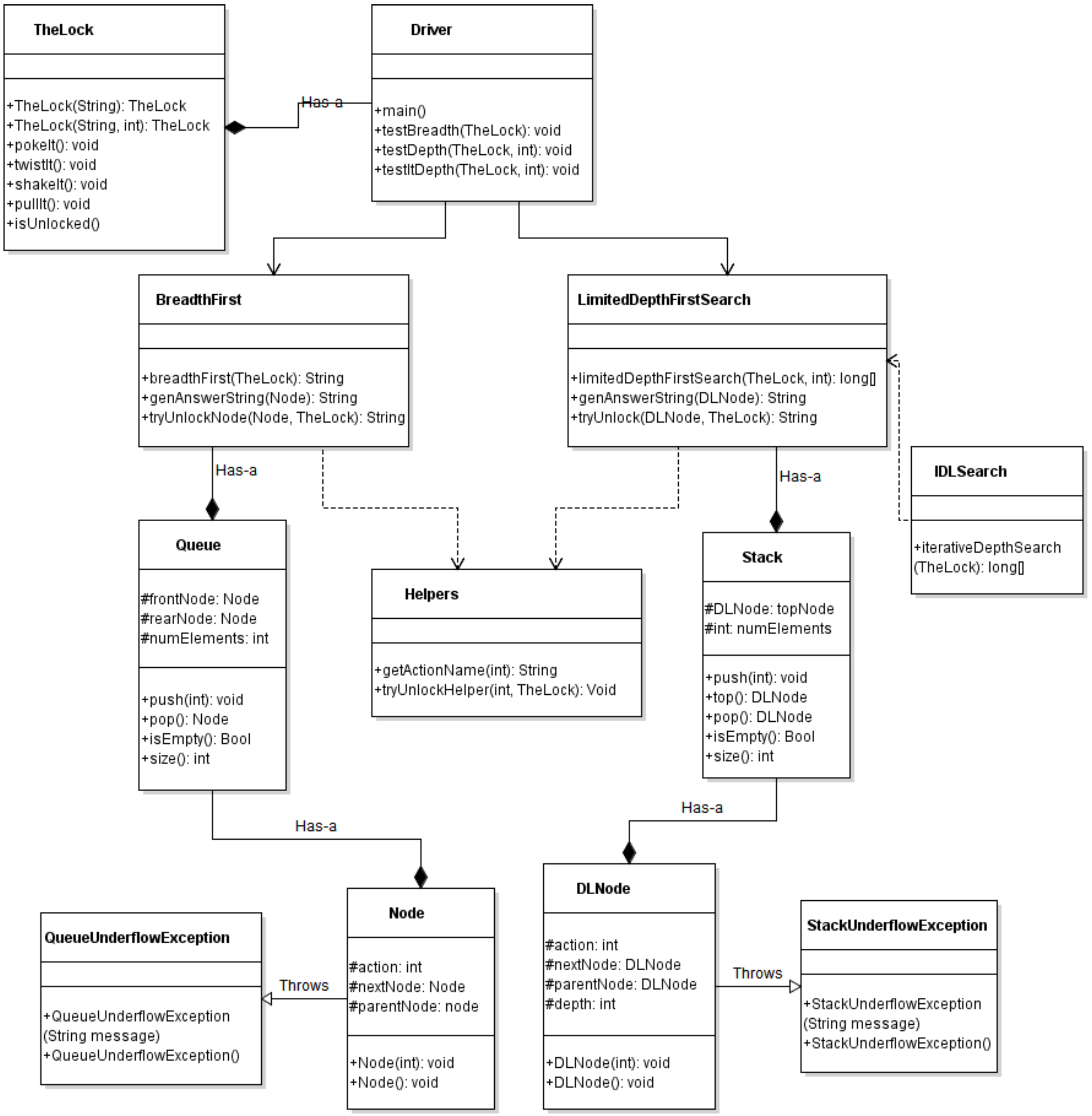
TheLock
+TheLock(String): TheLock
+TheLock(String, int): TheLock
+pokeIt(): void
+twistIt(): void
+shakeIt(): void
+pullIt(): void
+isUnlocked()
Has-a
Driver
+main()
+testBreadth(TheLock): void
+testDepth(TheLock, int): void
+testItDepth(TheLock, int): void
BreadthFirst
+breadthFirst(TheLock): String
+genAnswerString(Node): String
+tryUnlockNode(Node, TheLock): String
LimitedDepthFirstSearch
+limitedDepthFirstSearch(TheLock, int): long[]
+genAnswerString(DLNode): String
+tryUnlock(DLNode, TheLock): String
IDLSearch
+iterativeDepthSearch (TheLock): long[]
Has-a
Has-a
Queue
#frontNode: Node
#rearNode: Node
#numElements: int
+push(int): void
+pop(): Node
+isEmpty(): Bool
+size(): int
Helpers
+getActionName(int): String
+tryUnlockHelper(int, TheLock): Void
Stack
#DLNode: topNode
#int: numElements
+push(int): void
+top(): DLNode
+pop(): DLNode
+isEmpty(): Bool
+size(): int
Has-a
Has-a
QueueUnderflowException
+QueueUnderflowException (String message)
+QueueUnderflowException()
Throws
Node
#action: int
#nextNode: Node
#parentNode: node
+Node(int): void
+Node(): void
DLNode
#action: int
#nextNode: DLNode
#parentNode: DLNode
#depth: int
+DLNode(int): void
+DLNode(): void
Throws
StackUnderflowException
+StackUnderflowException (String message)
+StackUnderflowException()

Figure 4