Finding an Approximation Algorithm for Large Instances of Countdown Numbers

Daniel Gao, Class of 2019

Advisor: Zachary Kincaid

Motivation and Goal

Countdown is a British game show that first aired in 1982. The game is described as follows: the contestant in control first decides the amount of "large numbers" used, from 0 to 4. This amount of "large numbers" is then chosen randomly without replacement from the set of $\{25, 50, 75, 100\}$. Then, some "small numbers" are chosen without replacement from the set consisting of the numbers from 1 to 10 each repeated twice, for a total of 6 numbers. A three-digit target number is generated randomly. The contestants then have 30 seconds to try to construct the target number using each of the 6 numbers at most once and the operations add, subtract, multiply, and divide, with the additional rule that any intermediate step must result in a positive integer. When time is up, the winning contestants are the ones who have reached the target number, or if nobody can reach the target number, the one who has a solution closest to the target number.
There are many online solvers for the game show problem described above, and in fact the optimal solution to every possible Countdown problem has been already computed [2]. I am more interested in a generalized case where each number is chosen randomly from a range of $[0, k]$ for some positive integer k, and where the amount of numbers available, n, is larger than 6. I would like to build an approximation algorithm to solve generalized problems where the n is large enough that it is infeasible to use an exhaustive search. This is desirable in the original context of the game show problem because it is better to have an approximate solution in the limited time frame than to have no solution at all.

Problem Definition

Given n numbers chosen randomly from the range of $[0, k]$ and a randomly generated target number t, construct a solution s by using each of the n numbers at most once and the operations add, subtract, multiple, and divide, where each intermediate result must be a positive integer, such that s is as close to t as possible.

Problem Background and Related Work

The generalized Countdown problem is closely related to the subset sum and subset product problems, both of which are NP-complete. I have been able to prove that the generalized Countdown problems with only addition/subtraction operators or with only multiplication/division operators are NP-complete, but it is more difficult to show NP-completeness when all operators are available.

It is likely that the general case is also NP-complete.
The most relevant paper I have been able to find is "(The Final) Countdown" by Jean-Marc Alliot [1]. In this paper, Alliot describes two algorithms that he uses to solve Countdown: the depth-first search algorithm with hash tables, and the breadth-first search algorithm. He shows that, for $n < 9$, the DFS algorithm with hash tables performs the best. DFS operates by iteratively choosing 2 numbers from the search set, combining them with some operator, and inserting the result back into the set. However, for $n = 10$, each instance takes between 1 and 3 minutes for his algorithms to solve, which is very slow. Alliot shows that there is more likely to be a solution when the numbers are chosen from a larger range as opposed to from the original game show pool, which is good news for the generalized problem I'm trying to solve. The paper also contains useful statistics about the target numbers that are most likely to have solutions when $n = 6$.

Approach

My main approach would be to split a Countdown problem with a large n into smaller Countdown problems. For example, if $n = 10$, it would be split into two $n = 5$ problems. The idea is to precompute a probability distribution over all target numbers for the likelihood that it is solvable, and then try to find complements of the most likely numbers in the other subproblem. For example, if t is the target number for the $n = 10$ problem and x is the target number most likely to have a solution when $n = 5$, then we would randomly partition the 10 numbers into two sets of 5, and run Alliot's DFS algorithm [1] over one of the sets with $\{n + x, n - x, n + x, n/x\}$ as the target numbers. If these numbers can't be found, then we would search for numbers that are close. If there is a solution on this subset with one of these target numbers or their approximations, then there is a good probability that we can find x on the other subset and thus find a good approximation for n. Given that the problem is likely NP-complete, my approach exploits statistical patterns in subproblems to create a probabilistic algorithm. If I have time, I would also like make the algorithm into an anytime algorithm, meaning that it always maintains a candidate solution and iterably improves it over time. For example, given a candidate approximation that is higher than the target t, the algorithm might attempt to swap out larger numbers for smaller numbers, or addition operators for subtraction operators, until the approximation gets better. This would make it more likely that the approximation would get better the longer the algorithm runs.

Plan

I plan to have these things completed by the following dates:

Feb 26: Written proof of NP-completeness for the general case
March 5: Implementation of Alliot's DFS algorithm [1] in Python
March 19: Probability distribution for target numbers for a given k when $n = 5$
April 2: First implementation of approximation algorithm for a given k and $n = 10$
April 9: Algorithm improved to an anytime algorithm, along with other optimizations
April 23: Algorithm extended for other values of n and k
May 1: First draft of Written Report
May 7: Submit Written Final Report

If the approximation algorithm doesn't work as well as expected, I would experiment with different partitions of the $n = 10$ problem, for example into $n = 6$ and $n = 4$ subproblems.

Evaluation

I would first modify Alliot's DFS algorithm [1] to make it an "approximation algorithm" by having it record the closest number it has found to the target number t at any given time, and then output that number when the algorithm is stopped. To evaluate my approximation algorithm, I would run it parallel to the modified DFS algorithm on a Countdown problem with a large n, and then stop both algorithms at the same time before termination and compare the approximations. The numbers for the Countdown problem, the target number, and the time until stopping will all be randomly generated over a large number of trials, and the results will be an average of these trials. If the approximation algorithm were good, then it should beat the exhaustive algorithm that searches through all the possible solutions in order without any use of heuristics.

References

[1] Jean-Marc Alliot, '(The Final) Countdown', *alliot.fr/COMPTE/compte.html,* 2013

[2] Simon Colton, 'Countdown: Solved, Analyzed, Extended', *Proceedings of the AISB Symposium on AI and Games*, 2014

[3] Elliot Fenner, 'Solutions of Versions of the 'Countdown Numbers Game", *https://minerva.leeds.ac.uk/bbcswebdav/orgs/SCH_Computing/FYProj/reports/0001/fenner.pdf*, 2001