Hot Springs Part 2

William Frid

Spring 2024

Introduction

"Hot Springs" part 2 is a continuation of a previous assignment for the KTH course **Programming 2 ID1019** where part 1 laid the foundation for this next step. The problem to be solved is as before: a list of entries, each containing a map and a pattern, is given. And the goal being to calculate the amount of possible solutions for the unknown data in the map, such that we still follow the pattern.

In part 1 the solution used pure recursion to solve the problem. Nothing too fancy. But in the long run, if we we're to give it longer entries, the time it would take to find each possible solution would grow and the program would run way to slow to be reasonable.

To solve this, we introduce dynamic programming. This means that we'll reuse the algorithm written in part one, that breaks the problem down into smaller problems, but we will also implement a map (using the library Map) to memoize the results. This will result in less redundant calculations and help us to tackle longer entries.

As this document contains a rather small amount of code, it's recommended to read it alongside the code written found at https://github.com/williamfridh/ID1019_VT19_TCOMK_Programming_2. Follow the link and go to Assignments/springs.

The Need of Dynamic Programming

Before digging into the changes and results of changing to dynamic programming (the introduction of memoize), let's explore why we really need this trough some benchmarking. In part 1 each entry was just equal to one row, but in part 2, we're expected to multiply the length of each map (e.g. "???.###") and integer list (e.g. "1,1,3"). Accomplishing this is simple and is well described in the snippet below. *Note that the function written for this multiplies the length by a given integer for, which will come in handy for benchmarks.*

```
def extendEntry({sym, num}, extend) do
    extendEntry({sym, num}, {sym, num}, extend)
end

def extendEntry({sym, num}, {symNew, numNew}, extend) do
    if extend > 1 do
        extendEntry({sym, num},
        {symNew ++ [:unk] ++ sym, numNew ++ num}, extend - 1)
    else
        {symNew, numNew}
    end
end
```

For the benchmark we use a short list of just 6 rows which we increase the length of (using the function `extendEntry/2`). After running the benchmark simply recursive implementation it's clear that the time it takes to solve the problem increases too drastically to handle the real test (which is 1000 rows). This in clearly displayed in table 1 below.

Length multiplier	**Time (us)**	**Time/Prev. time**
1	170	-
2	200	1.2
3	780	3.9
4	5400	6.9
5	56000	10.3

Table 1: Benchmark showing the increase of time for the 6 line test file without memoize.

A quick test with the 1000 entries file showed was done but canceled due to too slow results. The simply multiplying the length of each entry by 3 resulted in a computation time of 118 second. Estimating the time it would take for a multiplier of 5 is hard as the time it took to handle a multiplication by 2 increased by 31 times, and a change from 2 to 3 lead to an increase of 552 times.

Meomoize

To convert the program to a dynamic one, we simply need to a storage for where we put each computation alongside is solution (memoize). This storage will be checked before starting each computation to save time. To do this, the library Map is of great help as it offers all the functionality needed plus a fast lookup (as is uses hashed keys).

For keys, we need to generate tuples containing the remaining symbol list (map), integer list (pattern), and the boolean forceBad (which was part of the solution for part 1). The function `eval/3` is reused with minor adjustments (now called `evalMem/3`) for passing around the storage used as well ass call upon the wrapper function `evalMemCheck/2` (seen below).

```
def evalMemCheck({sym, num}, mem) do
    evalMemCheck({sym, num}, false, mem)
end
def evalMemCheck({sym, num}, forceBad, mem) do
    memRes = Map.get(mem, {sym, num, forceBad})
    if memRes == nil do
        {res, mem} = evalMem({sym, num}, forceBad, mem)
        {res, Map.put(mem, {sym, num, forceBad}, res)}
    else
        {memRes, mem}
    end
end
```

This function does a very cleaver thing. As a wrapper of `evalMem/3` it takes the same arguments, but before it calls for evaluation, it checks if the results already in the storage. If it is, it simply returns that and save tons of computations. Otherwise, it calls for evaluation and inserts the new entry into the memory using the key mentioned earlier.

For the function `evalMem/3`, the adjustments is that is now also takes the storage/memory as an argument as these are needed now that it too calls upon the wrapper instead of directly upon itself during recursion as seen in the snippet below. Also pay notice to how it now returns a tuple with the result as well as the updated memory and how this memory is passed further in each if the if-cases.

```
def evalMem({[:unk | st] = sym, [nh | nt] = num}, false, mem) do
    num_dec = [nh - 1 | nt]
    if nh - 1 == 0 do
        {r1, m1} = evalMemCheck({st, num_dec}, false, mem)
        {r2, m2} = evalMemCheck({st, num}, false, m1)
        {r1 + r2, m2}
    else
        {r1, m1} = evalMemCheck({st, num_dec}, true, mem)
        {r2, m2} = evalMemCheck({st, num}, false, m1)
        {r1 + r2, m2}
    end
end
```

Result

Now to the result of this new dynamic programming implementation. The same benchmark as before is used, as well as one using the 1000 entries file. Looking at table 2 showing the benchmark of the short test file tells us that this solution is somewhat slower on really small sets of data. This is to be expected as the memory doesn't get to be used and just mostly adds are being performed on the memory and very few reads.

Length multiplier	**Time (us)**	**Time/Prev. time**
1	450	-
2	520	1.2
3	720	1.4
4	920	1.3
5	1350	1.5

Table 2: Benchmark showing the increase of time for the 6 line test file with memoize.

Table 3 however, shows us the results for the file of 1000 entries. Notice how this solution could perform all 5 cases rather fast, and that the time increase became smaller and smaller thanks to the memory got filled with more thanks to the larger amount of data solutions and less computations had to be done.

Length multiplier	**Time (us)**	**Time/Prev. time**
1	40000	-
2	270000	6.6
3	700000	2.6
4	2900000	4.1
5	2500000	0.8

Table 3: Benchmark showing the increase of time for the 1000 line test file with memoize.

Conclusion

Dynamic programming is a powerful method each programmer should master as it allows for faster computations as in this case. It's required to write code that can handle scalable problems and becomes more important on less powerful hardware.