Volcanic APL Advent-ure

Marvin Borner
marvin.borner@student.uni-tuebingen.de
Eberhard Karls Universität
Tübingen, Baden-Württemberg, Germany

ABSTRACT

As part of a programming seminar we are tasked to solve a problem of the yearly programming competition *Advent of Code*. We translate the given storyline to an abstract problem, which involves a grid and two slightly different graph searches. We then visualize the problem for better understanding, derive pseudocode solutions, and translate the pseudocode to the APL programming language. We confirm our solutions by visualizing the algorithms in animations. We argue that a structured approach to problem-solving is an efficient way of learning new programming languages.

KEYWORDS

Advent of Code, BFS, APL, problem-solving, proseminar

1 ADVENT OF CODE

Advent of Code is yearly created by Eric Wastl and contains 25 programming puzzles – one for each day until Christmas. The puzzles cover different programming techniques and appeal to programmers of varying skill set[11].

Each problem of a year is part of a coherent storyline and consists of two related parts. Approaching Christmas the problems get increasingly more difficult, judging by the average solving times on the public leaderboard. Still, the challenges often involve the same fundamental techniques[10][1]:

- Graph Traversal: e.g. days 12, 15, 18, 19, 21
- Map Parsing: e.g. days 8, 11, 13, 14
- Mathematics: e.g. days 2, 3, 6, 7, 9
- (Virtual) Machines: e.g. days 5, 10

A solution to any problem of Advent of Code is the composition of multiple steps:

1. *formalization* of an informal story to a precise problem description
2. *parsing* of the text-based input into a fitting data structure
3. implementation of *algorithms* that solve the problem
4. *integer outputs* that represent or combine some state of the algorithms

We use a new-to-us programming language. To make it easier, we achieve step (3) by first creating a pseudocode solution which we then translate to the target language.

2 PROBLEM

As part of a programming seminar we are given the 18th Advent of Code problem of the year 2022. Empirically, we can assign an estimated difficulty level of higher-than-average based on the distance to day 25.

The story is connected to the previous problem of day 17 and occurs at a similar location with the same characters. The protagonist, in the story denoted as *you*, is observing an erupting volcano from afar. They realize that some of the lava drops fall into nearby water and turn into *obsidian*. The story suggests that such reaction happens depending on the *cooling rate* of the lava drop, which in turn depends on the *surface area* of the drop[12].

The protagonist measures the lava drops using an imprecise measuring device. For each lava drop, the device returns a set of three-dimensional coordinates of *lava cubes* that resemble a rough grid of its volume.

Both parts of the task involve the calculation of the surface area of a grid-based approximation of one given lava drop. The second part, as described below, involves an edge case that is not dealt with in the first part.

3 APL

We use Dyalog's implementation of the APL (*A Programming Language*) programming language for the final implementation of the solutions for both tasks.

APL was introduced in 1962 by Kenneth E. and is closely related to notations of mathematics[6]. Only slowly the language was implemented to become an actual programming language. Dyalog's implementation is based on the original notation but extends upon it with various modern features[5].

The language is arguably very different from common languages, for example since it uses Unicode symbols instead of ASCII words for most functions, control structures and operators. We will therefore give a short overview of the language.

By default, APL includes many built-in functions, notably equivalents of "transformation" functions such as `map` or `fold`. Those functions are either *niladic*, *monadic*, *dyadic*, or any subset of the three. This adicity specifies the number of arguments a function gets, but also selects the actual computation of the function. In the dyadic case, one argument gets applied to both sides of the function:

```
      ⌈2.9 3.2 ⍝ monadic ceiling
3 4
      2.9⌈3.2  ⍝ dyadic maximum
3.2
```

By composing such symbols, APL code can be more compact than equivalent code in other languages. A famous example is the simulation of the Game of Life cellular automaton, originally implemented by Conor Hoekstra[9]:

```
life←{↑1 ω∨.∧3 4=+/,¯1 0 1∘.⊖¯1 0 1∘.⌽⊂ω}
```

Since functions get applied with right associativity, it can help to read lines of code from right to left.

APL is an *array-oriented* programming language, meaning that the prevalent data structure is an array. Since its functions typically work on multi-dimensional array structures, efficient implementations allow parallel evaluation of standard non-parallel algorithms[2]. This makes the language attractive to data scientists and programmers of high-performance software.

4 PARSING

The input consists of more than 2700 lines of comma-separated integers in the interval $[0, 20]$. Every line represents the coordinate of a lava cube inside the lava drop.

A prerequisite of solving the actual task is the parsing of the input data and its translation to a fitting data structure. For efficient traversal of the data we want the parser to yield a two-dimensional array of coordinates:

```
input          array
2,2,2        [[2,2,2],
1,2,2         [1,2,2],
3,2,2   -->   [3,2,2],
2,1,2         [2,1,2],
2,3,2         [2,3,2],
 ...             ...  ]
```

In Python-like pseudocode this translation could be accomplished using a nested list comprehension:

```
cubes = [
    [int(n) for n in line.strip().split(",")]
    for line in open("input").readlines()
]
```

In APL, we can use a more minimal technique based on evaluation:

```
⎕IO←0 ⍝ use zero-based indexing
cubes←⍎¨⊃⎕NGET'input'1
⍝ Symbols:
⍝   ⍎: Execute expression
⍝   ¨: Each
⍝   ⊃: First
```

From right to left: We read the input data and split it linewise into a nested array (implied by the 1 flag of NGET), then take the first element of the resulting array (the *content* of the file), and then execute each line. In APL, the execution of a string like 2,2,2 results in an array of the respective three numbers:

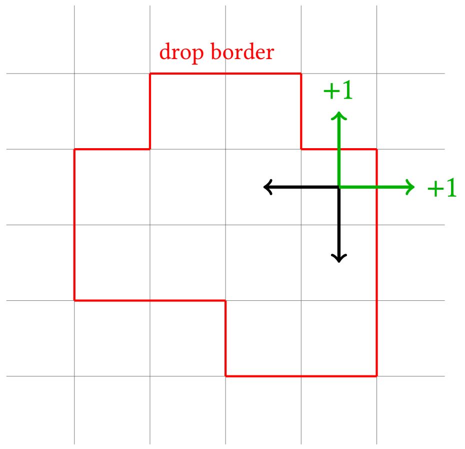
```
      cubes
┌─────┬─────┬─────┬─────┬─────┐
│2 2 2│1 2 2│3 2 2│2 1 2│2 3 2│
└─────┴─────┴─────┴─────┴─────┘...
```

5 TASK 1

In the first part we are tasked to calculate the surface area of the lava drop using the given coordinates of lava cubes.

All cubes are, by definition, the same size. The total surface area is therefore the result of adding all sides of the cubes that aren't directly connected to another cube. In an initial solution we do this by iterating over every cube, looking at its 6 sides, and incrementing a global counter variable if the side coordinate is not within the given cube set.

In a minimized two-dimensional setting this can be visualized as the following procedure, where the arrows indicate the sides the algorithm looks at (here only drawn for a single vertex):



Note that we deliberately ignore the fact that the technique also counts non-connected coordinates *inside* the lava drop. Not counting this is part of the second task and irrelevant for this section.

The following pseudocode will therefore be a valid solution:

```
count = 0
for cube in cubes:
    for neighbor in neighbors(cube):
        if neighbor not in cubes:
            count += 1
print(count)
```

We now derive a mathematical solution from the iterative code. Here, we first calculate the 6 movements, then create a multiset by adding every cube to every movement, subtract the original set of cubes, and finally count the elements of the resulting set:

$$\begin{aligned}
\text{cubes} &= \{(2,2,2),(1,2,2),(3,2,2),(2,1,2),(2,3,2),\ldots\}\\
\text{movements} &= \{I_3, -I_3\}\\
\text{sides} &= [c+m \mid \forall c \in \text{cubes}, \forall m \in \text{movements}]\\
\text{result} &= |\text{sides} - \text{cubes}|
\end{aligned}$$

The equations can be directly translated to APL code:

```
movements←{ω,-ω}↓(3 3⍴4↑1)
sides←,cubes∘.+movements
≢sides~cubes
```

In the first line we compute the identity matrix I_3 by creating a vector $(1,0,0,0)$ and then using the dyadic `Reshape` function that turns it into a 3×3 array by repeating it until the array is filled. `movements` is then the result of the concatenation of the positive and negative identity matrix – here created using a curly-bracket function with its right argument ω concatenated with the negated $-\omega$.

Secondly, we calculate the summed outer product of `cubes` and `movements` using ∘.+ which is then concatenated into a one-dimensional array.

The final step uses the dyadic `Without` function ~ to subtract the sets and the monadic `Tally` function to count the elements.

5.1 Complexity

The initial pseudocode solution has a linear time complexity of $O(6n) = O(n)$, since we iterate every cube once and the amount of neighbors is fixed to be 6. This assumes that the set inclusivity check is also in $O(n)$, which can be implemented to take constant time $O(1)$ [13].

The APL solution is a transformation of the pseudocode solution and uses the same amount of "loops", $O(6n) = O(n)$. Here, the set subtraction is done externally and would also require n steps of set inclusivity checks, therefore taking time in $O(n)$. However, since the code only operates on multi-dimensional arrays, we can expect compiler optimizations to take down the effective worst-case complexity to sublinear time[2].

5.2 Worst-Case

The complexity of our implemented algorithm is solely defined by the amount of given coordinates, ergo the size of the lava drop. Since we expect set inclusivity checks to take constant time regardless of inclusivity or exclusivity, it doesn't affect the performance whether the cubes are connected, have holes between them, or are not connected at all.

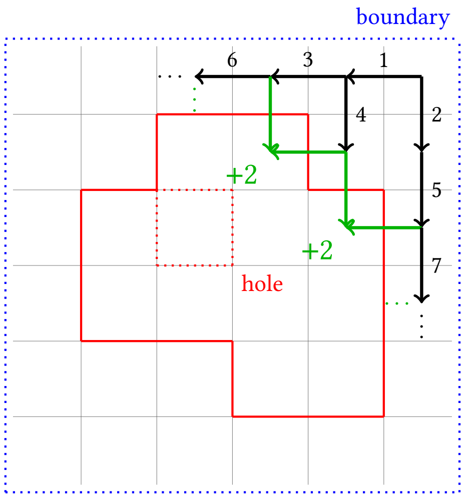
6 TASK 2

As stated before, the second task also considers *holes* filled with air inside the lava drop. The previous solution counts holes as surface area since they neighbor cubes while not being inside the cube set itself.

Here, we count only the exterior surface area of the lava drop, without considering any holes within. To do this, we construct a boundary around the entire grid and then use *flood-filling* around the lava drop without crossing into the drop itself – thus ignoring all holes.

Still, for every visited cube the six neighbors have to be iterated. If a neighbor is part of a cube set, the global counter needs to be incremented. Our flood-filling algorithm therefore needs a set of *visited* nodes such that we don't count any borders twice.

In the following minimized two-dimensional example we start from the top right of the boundary and fill the grid in a breadth-first order. The first 7 steps are marked, with the additional queries for the neighbors marked in green. Note how holes inside the cube set will never be considered by the algorithm.



To implement this in pseudocode, we use a breadth-first-search (BFS) algorithm:

```
count = 0
visited = set(cubes)
Q = [max_coord + (1, 1, 1)]
while Q:
    cube = Q.dequeue()
    for neighbor in neighbors(cube):
        if not in_boundary(neighbor):
            continue
        if neighbor in cubes:
            count += 1
        if neighbor in visited:
            continue
        visited.add(neighbor)
        Q.enqueue(neighbor)
print(count)
```

The `visited` set is initialized with the cubes such that the flood-fill never enters the lava drop itself. The starting value of the queue is some point outside of the cube set, here the incremented maximum coordinates.

Translating the pseudocode to APL requires implementations of the assumed definitions:

```
mins←(↑⌊/cubes)-1 1 1
maxs←(↑⌈/cubes)+1 1 1
in_boundary←{0≡+/((ω<mins),(ω>maxs))}
count←0
```

The maximums/minimums are calculated by folding the cubes over the dyadic maximum/minimum function and then adding/subtracting 1.

The boundary checker creates a one-dimensional array of 6 ones or zeroes depending on whether any value of the given coordinate is smaller or larger than the minimum and maximum coordinates. These values are then summed using the dyadic fold function and compared to zero – ultimately yielding a one/zero if the coordinate is within the boundary or not.

Our BFS implementation is derived from a snippet within Dyalog's utility functions database[8]:

```
θ{                                 ⍝ no vertices visited.
    ω≡θ:α                          ⍝ no vertices left: done.
    head←1↑ω ⋄ tail←1↓ω            ⍝ next and remaining vertices in queue.
    next←head⊃graph                ⍝ vertices adjacent to head.
    (α,head)∇(tail∪next)~α         ⍝ accumulate visited vertices.
}start                             ⍝ from starting vertex.
```

This uses a trick that only works with APL's dyadic functions: The set of visited vertices as well as the queue are encoded as left and right arguments of the function (see first and last line). In the recursive dyadic call ∇, the visited set is extended with the lastly dequeued element head of the queue, and the queue is set to the union of the `tail` and `next` elements (with the already visited vertices α removed).

Our solution applies several modifications:

(1) Return the global counting variable if the queue is empty. Also use a different ending condition since our queue is multi-dimensional because of the coordinate arrays.
(2) Next elements are chosen from the neighbors within the boundary.
(3) The global counting variable is incremented by all neighbors that are not inside the cube set.

```
cubes{
    0=≢ω:count                                   ⍝ (1)
    head←1↑ω ⋄ tail←1↓ω
    neighbors←movements+¨head                    ⍝ (2)
    next←{(in_boundary¨ω)/ω}neighbors            ⍝ (2)
    count⊢←count+(+/{(↓ω)∊cubes}¨neighbors)      ⍝ (3)
    (α,head)∇((tail∪next)~α)
}↓maxs
```

Similar to the pseudocode implementation, the visited set is initially set to `cubes` such that the algorithm doesn't enter the drop. The queue is initialized to the starting point, here an array of the incremented maximum coordinates `maxs` defined before.

6.1 Complexity & Worst-Case

The breadth-first-search algorithm has a time complexity of $O(V+E)$, V being the number of vertices and E the number of edges in a graph[3].

In the worst case, the number of accessible vertices is (approximately) the entire area within the boundary. In our implementation this scenario can be achieved by setting the input to two coordinates with the first one being at the front-bottom-left of a large hypothetical cube and the second one being at the back-top-right.

In a realistic setting this worst-case scenario doesn't make sense. We would not expect a lava drop to have only two parts with a lot of air between them. However, even if we had six lava cubes in a three-dimensional grid connected by single-cube diagonals, the limiting behavior with arbitrarily large coordinates would still be in the same range.

In general, we can expect the boundary dimensions to be proportional to the input size n. Then, $V = n^3$.

The number of edges is $6V = 6n^3$, since every vertex has six neighbors. Therefore the time complexity is $O(7n^3) = O(n^3)$.

7 VISUALIZATIONS

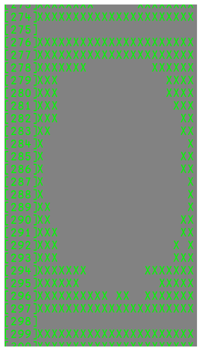
Some APL editors support a live view of variables. For our visualizations, we used Dyalog's Ride IDE. Changing the variables at runtime will directly show the updated data to the user. Inspired by the `life` snippet from the beginning, we use this feature to repeatedly update a grid of characters to get an animation of the data.

In the first animation we want to visualize the "movement" of the BFS algorithm in part 2. To do this, we extend the implementation with an instruction to write 'X' at every dequeued coordinate. Since the lava drop itself is never entered, we expect an *inverted* view of the drop. The grid is three-dimensional so the data is printed as a concatenation of its two-dimensional layers.

```
_←⎕ED 'grid' ⍝ open variable preview
grid⊢←(maxs+1 1 1)⍴' ' ⍝ matrix of spaces
cubes{
    0=≢ω:count
    head←1↑ω ⋄ tail←1↓ω
    grid[|head]⊢←'X' ⍝ draw 'X' at current coord
    neighbors←movements+¨head
    next←{(in_boundary¨ω)/ω}neighbors
    count⊢←count+(+/{(↓ω)∊cubes}¨neighbors)
    (α,head)∇((tail∪next)~α)
}↓maxs
```

For simplicity, the implementation forces the minimum coordinate to be $(0, 0, 0)$ using the monadic absolute function |. This is also in accordance with our input data.

The animation shows the progress of the BFS and ends with the final state of the algorithm. For example, the final state around the middle of the generated output (notice the start of the layers above and below):



```
[274]XXXXXXXXXXXXXXXXXXXXXX
[275]
[276]XXXXXXXXXXXXXXXXXXXXXX
[277]XXXXXXXXXXXXXXXXXXXXXX
[278]XXXXXXX         XXXXXX
[279]XXX               XXXX
[280]XXX               XXXX
[281]XXX                XXX
[282]XXX                 XX
[283]XX                  XX
[284]X                    X
[285]X                   XX
[286]X                   XX
[287]X                    X
[288]X                    X
[289]XX                   X
[290]XX                  XX
[291]XXX                 XX
[292]XXX                X X
[293]XXX                XXX
[294]XXXXXXX        XXXXXXX
[295]XXXXXX           XXXXX
[296]XXXXXXXXXX XX  XXXXXXX
[297]XXXXXXXXXXXXXXXXXXXXXX
[298]
[299]XXXXXXXXXXXXXXXXXXXXXX
```

We then show the layers of the grid using a timeout and iteration:

```
_←⎕ED 'layergrid'
{
    layergrid⊢←⍵
    _←⎕DL 0.3 ⍝ wait for 300ms
}¨↓↓grid ⍝ for each layer
```

A recording of the animation can be found in the supplementary material or on GitHub.

8 DISCUSSION

Our solutions both run within a reasonable timeframe of less than one second. Aside from using explicit parallelization, we were not able to find more efficient solutions to the problem.

In part 2, there may be some benefits of running the BFS *within* the lava drop. This eliminates the need for a boundary and, depending on the shape of the drop, could require fewer steps.

However, it is not clear whether the lava drop is guaranteed to be one coherent shape – it could have smaller drops outside of its main part (see also Complexity & Worst-Case). Our technique does not ignore such cases.

8.1 APL

The APL language was chosen out of interest and is unusual for such programming problems[4]. We believe this is because of three main reasons:

(1) *Accessibility*:
  - Tutorials are not easy to find and often assume existing knowledge about array/stack-oriented programming.
  - Documentation may only work for specific APL dialects or versions.
  - Forums on specific problems are often broken or not existent.

(2) *Learning Curve*:
  - Memorizing unicode operators and their key combinations require a lot of time and dedication.
  - The syntax and coding style is unconventional and different from most other modernly used languages.

(3) *Attractiveness*:
  - Websites and documentations do not look modern or appealing to young programmers.
  - Slogans and users do not make convincing arguments for using APL.
  - APL is often represented as an esoteric “golfing” language.
  - Some design choices are implications from backward compatibility and its old age.[7]

We knew some of these arguments before deciding on using this language. In the process of solving the problems, however, we learned to appreciate the concise and logical way that the language is structured. Contrary to popular belief, the language can be very readable and enjoyable.

While our solutions themself are not the most concise and could be reduced further, one can clearly see a significant code reduction during the translation of pseudocode to APL. Furthermore, the final solutions even use more elegant algorithms than originally intended.

Most of the stated criticism is related to the language’s history and age and could be reduced with better documentations, tutorials, and more standardization. Modern alternatives such as “BQN” try to reduce these issues and are more appealing to younger programmers.[7]

Generally we believe that languages like APL should be known to more people, for example by being taught in more universities.

8.2 Problem Solving & Language Learning

Problems such as the Advent of Code challenges often involve the same techniques and can be approached using similar strategies, as shown in section 1. Having a structured approach to the problem not only helped us understand the underlying tasks, but also to learn the APL programming language itself.

To us, simplifying the problem and reducing its parameters – for example using two dimensions instead of three – helped finding an initial solving strategy. In combination with visualizations, we were able to directly derive pseudocode solutions. Problems arising during the translation of the pseudocode to the actual solution allowed us to research *concrete* aspects of APL.

We believe this technique to be generally useful for learning new programming languages.

9 CONCLUSION

We demonstrated solving a specific programming problem through a structured approach. By careful problem reduction and visualizations, we were able to derive succinct pseudocode involving only common data structures and algorithms. Step-by-step, our technique yielded a final fairly concise snippet of APL code.

We showed that solving these problems in a deliberate structure not only makes problem-solving and algorithmic understanding easier, but can also help learning new languages.

We want to thank the anonymous reviewers for their helpful notes.

REFERENCES

[1] Marvin Borner. 2022. *Advent of Code Solutions.* https://github.com/marvinborner/AdventOfCode/tree/main/2022

[2] Wai-Mee Ching and Da Zheng. 2012. Automatic parallelization of array-oriented programs for a multi-core machine. *International Journal of Parallel Programming* 40, 5 (2012), 514–531.

[3] Charles E.; Rivest Ronald L.; Stein Clifford Cormen, Thomas H.; Leiserson. 1990. 22.2 Breadth-first search. *Introduction to Algorithms* (1990), 531–539.

[4] Jeroen Heijmans. [n. d.]. *Advent of Code Survey.* https://jeroenheijmans.github.io/advent-of-code-surveys/

[5] Roger KW Hui and Morten J Kromberg. 2020. APL since 1978. *Proceedings of the ACM on Programming Languages* 4, HOPL (2020), 1–108.

[6] Kenneth E Iverson. 1962. A programming language. In *Proceedings of the May 1-3, 1962, spring joint computer conference.* 345–351.

[7] Mashall Lochbaum. [n. d.]. *Why BQN.* https://mlochbaum.github.io/BQN/commentary/why.html

[8] John Scholes. 2022. *Classic Breadth-First Search.* https://dfns.dyalog.com/n_bfs.htm

[9] John Scholes. 2022. *John Conway’s ”Game of Life”.* https://dfns.dyalog.com/n_life.htm

[10] Eric Wastl. 2022. *Advent of Code.* https://adventofcode.com/2022/

[11] Eric Wastl. 2022. *Advent of Code, About.* https://adventofcode.com/2022/about

[12] Eric Wastl. 2022. *Advent of Code, Day 18.* https://adventofcode.com/2022/day/18

[13] Daniel M Yellin. 1992. Representing sets with constant time equality testing. *Journal of Algorithms* 13, 3 (1992), 353–373.