NZPC 2023: 100-point problem editorials

version 0.1

Problem M: Molecules (Part 2)

This problem can be solved using maximum flow or modified bipartite matching.

To solve using max flow: Partition the atoms into two groups in a checkerboard pattern. Construct edges with capacity 1 between every pair of adjacent atoms (note that each edges will be between an atom in the first group and an atom in the second group). Create a "source" node and construct edges from the source node to each atom in the first group, with capacity equal to the valence of the atom. Likewise, create a "sink" node and construct edges from each atom in the second group to a sink node, with capacity equal to the valence of the atom. Then the grid is valid if and only if the maximum flow from the source to the sink is equal to expected number of bonds, which is half of the sum of the valences. (The edges between the two groups that have flow correspond to bonds; the capacities ensure the constraints of the problem are satisfied.) Using the Ford–Fulkerson algorithm, the time complexity is $O(E*f)$ where $E = O(r*c)$ is the number of edges and $f = O(r*c)$ is the maximum flow.

To solve using bipartite matching: Partition the atoms in a checkerboard pattern, and use a modified maximum bipartite matching algorithm where instead of allowing each node (atom) to be incident to at most one selected edge, the number of allowed edges is the valence of the atom. As above, the grid is valid iff the size of the maximum matching is equal to the expected number of bonds.

This problem was from the 2002 NZ Programming Contest.

Problem N: Faucet Flow

This is an *ad-hoc* problem that appears simple at first, but turns out to have some very tricky cases.

We start by determining whether the water will first spill over the left edge (i.e. the leftmost divider) or the right edge. Let max_L be the height of the highest divider to the left of the faucet, and let max_R be the height of the highest divider to the right. If $max_L < max_R$ then the water will spill over the left edge first, and if $max_L > max_R$ then the water will spill over the right edge.

The case where $max_L = max_R$ trickier. Let a be the location of the closest divider to the left of the faucet of height max_L, and let b be the location of the closest divider to the right of the faucet of height max_L (= max_R). The water will first fill up the "middle" section between a and b. Next, it will spill over the two dividers at a and b simultaneously (with half the flow rate over each), and continue until it spills over one of the edges. To determine which edge, we can compute the capacity of each side (the volume of water at the instant where water would spill over the edge). This turns out to be rather easy, because the water will form a simple series of steps: we can start from the edge and work our way inward until we reach a or b, keeping track of the highest divider seen so far; this will be the height of the water. Summing these heights (and multiplying by two because that's the gap between dividers) gives the capacity. The water will first spill out over the side that has the smaller capacity.

This also lets us compute the time for the case where $max_L = max_R$: it will be the time taken to fill up the middle section (which is simply the volume of the rectangular region), plus the time taken to fill the smaller of the two capacities (which is simply double the capacity, because the flow rate is halved).

The last part is to compute the time for the case where $max_L < max_R$ (the case where $max_L > max_R$ is symmetric and can be handled by reversing the input). Let *a* be as above and let *c* be the location of the closest divider to the right of the faucet that has height strictly greater than max_L. It is tempting to think that the water will fill the region between *a* and *c* up to height max_L, but that is not necessarily the case: if there is a divider of height max_L between the faucet and *c*, then the water will first fill up to that divider, and then it will spill over both sides simultaneously and may not fill all the way to *c*. So the algorithm is as follows:

A: If there is no divider of height max_L between the faucet and *c*, compute the volume by scanning from the left edge to *c*, keeping track of the highest divider seen so far and summing as above.

B: Otherwise, let *b* be the location of the closest divider to the right of the faucet of height max_L. The water will first fill up the region between *a* and *b* (the time taken for this part is trivial to compute), then it will spill over the two dividers at *a* and *b* simultaneously (with half the flow rate over each). Next, there are two sub-cases to consider:

B1: If the capacity to the left of *a* is smaller than the capacity of the rectangular region between *b* and *c*, then the water will spill over both *a* and *b* at half the flow rate until it spills over the left edge. The time taken is double the capacity to the left of *a*.

B2: Otherwise, the water will spill over both *a* and *b* at half the flow rate until it fills the rectangular region between *b* and *c*; then, it will fill the remaining capacity to the left of *a* (at the full flow rate) until it spills over the left edge. The time taken can be computed using careful arithmetic.

This problem was from the Waterloo ACM Programming Contest, 21 September, 2002.

Problem O: Lambda Lifting

This is a real-life problem that was first described and solved by Thomas Johnsson in 1985[1]; improvements were made over a series of papers ([2], [3], [4]).

This editorial focuses on the algorithmic problem of determining the minimal set of additional parameters. The full solution involves a considerable amount of additional work (parsing, resolving identifiers, building the call graph, identifying free (non-local) variables, and printing the output); but that work is fairly straightforward so there is not much to discussed.

We begin by describing some approaches that *don't* work (**A**, **B**). Then we describe a simple correct approach (**C**); it is slow, but fast enough for the bounds in this problem. We also point out an edge case relating to dead code (**D**). Then, as a bonus, we give a very brief overview of the algorithms found in the literature, which are rather complex (**E**), and finally we describe a simple and fast algorithm (**F**) which is not found in the literature.

A. If it weren't for the constraint that the set of additional parameters must be minimal, the simplest approach would be as follows: For each function, collect all the parameters of all its enclosing functions, and add them as the additional parameters. This would produce a syntactically valid program when the functions are lifted to global scope, however it may introduce unnecessary parameters. One might try to fix this by only introducing parameters if they are referenced in the function, however this can fail because a function may refer to a parameter *indirectly*: in Sample Input 2, the function `f` requires parameter `b` even though `f` never refers to `b` directly, because `f` calls `g` which refers to `b`. What we require is the transitive closure.

B. The next approach (which also doesn't quite work) uses a top-down recursive algorithm: to determine the additional parameters of a function *f*, take the union of

- the parameters referenced directly in the expression of *f*, and
- the additional parameters of all the function called from *f*, computed recursively;

then remove the parameters that are declared in *f* itself, and the resulting set gives the additional parameters for *f*. The issue with this approach is that fails on mutually recursive functions: if *f* calls *g* and *g* calls *f*, the algorithm would recurse infinitely. (Aside: It is possible to use a smarter implementation that stops when it detects a cycle. That would produce a correct result for the first function, but the additional parameters computed while processing other functions recursively may be incorrect – the parameters won't get propagated all the way around the cycle. Since the bounds in the problem are small, this could be fixed by invoking the algorithm separately on each function, without reusing the partial results across invocations. But approach **C** that follows is simpler.)

C. We now give a simple and correct approach as pseudocode. It uses the same propagation rules as **B**, but it propagates the parameters iteratively instead of using recursion.

```
for f in all_functions:
    additional_params[f] = the parameters referenced in the expression of f,
                           excluding the parameters of f
made_changes = true
while made_changes:
    made_changes = false
    for f in all_functions:
        for g in functions called from f:
            for v in additional_parameters[f]:
                if v is not in additional_parameters[f],
                        and v is not in the original parameters of f:
                    add v to additional_parameters[f]
                    made_changes = true
```

This algorithm is guaranteed to eventually terminate, and when it does all the sets of additional parameters will be correct. The time complexity is $O(V \times C \times F)$, where V is the number of parameters, C is the number of call expressions, and F is the number of functions. This is because the innermost loop will have at most V iterations each time; the loops over *f* and *g* iterate over all the call expressions, so together will have C iterations each time; and the outer **while** loop can have at most F iterations until all parameters are propagated to all functions. This time complexity is not optimal, but is fast enough for the bounds in this problem.

D. An important edge case is the presence of dead code. Consider the following input program:

```
func main ( a )
    func f ( )
        func g ( )
            a
        end
        1 + 1
    end
    f ( )
end
```

In this program, the function `g` refers to the parameter `a` and hence requires it as an additional parameter; however, the function `f` does *not* require `a` as an additional parameter: only parameters referenced in the *expression* of the function (directly or indirectly) are required, and in the case of `f` the expression is `1 + 1` (which does not reference `a`). The fact that `a` appears in a nested function definition within `f` is irrelevant.

E. Now for a very brief overview of the algorithms found in the literature: The original algorithm given in Johnsson's paper[1] is $O(n^3)$ where n is the size of the input program. It is based on setting up a system of recursive equations (equivalent to the rules in `B`) and then solving it using repeated substitution. Subsequently, [2] tried to improve the time complexity to $O(n^2$ by decomposing the call graph into strongly-connected components (SCCs); it collapses each SCC into a single node, and assigns all the functions in an SCC the same set of parameters; this eliminates cycles and allows the recursive algorithm (**B**) to be used. However, it was later discovered[3] that this approach does not produce the minimal set of parameters in some cases.

The final algorithm from the literature[4] fixes that issue using dominators while retaining the *$O(n^2)$* time complexity; it's fairly complicated.

F. It turns out that there is a simpler algorithm with time complexity *$O(n^2)$*. The key observation is that it's much easier to only consider one parameter at a time (and use multiple passes over the call graph; thankfully it turns out that this isn't any slower than handling all the parameters in a single pass over the call graph). The algorithm is as follows:

```
for v in all_parameters:
    Perform a depth-first traversal (DFS):
        Start from the functions that reference v in their expression,
        and propagate to the callers of functions,
        but do not propagate to the function in which v is declared.
    Add v as an additional parameter of the functions traversed by the DFS.
```

The worst-case time complexity is *$O(V \times C)$* where *V* is the number of parameters and *C* is the number of call expressions. This is because a single DFS is *$O(E)$* where *$E = C$* is the number of edges, and the number of DFS's is *V*. Since there exist cases where the size of the output is *$O(V \times C)$*, this worst-case time complexity is optimal.

It is possible to go further and show that the time complexity is *O(len(input) + len(output))*, plus the time taken to sort the additional parameters. This can be proven by matching each operation of the DFS's to a unique token in the output.

[1]: Johnsson, T. (1985). Lambda lifting: Transforming programs to recursive equations
[2]: Danvy, O., & Schultz, U. P. (2002, September). 🔒Lambda-lifting in quadratic time. In *International Symposium on Functional and Logic Programming* (pp. 134-151). Springer Berlin Heidelberg.
[3]: Morazán, M. T., & Mucha, B. (2006). Improved Graph-Based Lambda Lifting. In *Software Engineering Research and Practice, SERP 2006* (pp. 896-902).
[4]: Morazán, M. T., & Schultz, U. P. (2008). 🔒Optimal Lambda Lifting in Quadratic Time. In *Implementation and Application of Functional Languages, IFL 2007* (pp. 37-56). Springer Berlin Heidelberg.

Problem P: Losing Lakes

This is a computational geometry problem which does not require a fast algorithm but does require considerable attention to detail.

Firstly, note that the orientation of the polygons in the input is unspecified – the points may be given in clockwise or counterclockwise order. The orientation can be determined by computing the signed area of the polygon; a positive value means the points are in counterclockwise order, and negative means clockwise. It may be helpful to reverse the points to ensure a particular orientation.

We only consider a single lake, because multiple lakes can be handled independently. We can find all the intersections between the borders of the lava and the lake by using a line segment intersection test (e.g. simultaneous equations or cross product) on each line segment of the lava against each line segment of the lake. We can also identify the ordering of these intersections around the border of the lake, and likewise their order around the border of the lava (this requires a little care when there are multiple intersections on a single line segment).

Imagine walking along the border of the lake; notice how the parts between intersections with the lava alternate, being either covered by lava or outside the lava area. Likewise when walking around the border of the lava, the parts between intersections alternate, being either over the lake or outside the lake. To identify which part is which, we can inspect some pair of intersecting line segments: check whether they form a clockwise or counterclockwise turn (using the sign of their cross product), and carefully interpret the result.

Now consider the newly-formed lakes. The border of each new lake consists of a sequence of parts, alternating between a lake border outside the lava area and a lava border over the original lake. We can trace each of these new borders by starting from an arbitrary intersection and alternating between jumping to an adjacent intersection along the original lake border and jumping to an adjacent intersection along the lava border (being careful to go in the appropriate direction). This allows us to identify the new lakes and count them.

There is one special case: The border of the lake might not intersect with the border of the lava at all. This could be because the lava does not touch the lake, or because the lake is entirely covered by lava. It is possible to distinguish between them by picking an arbitrary point on the lake and checking if it's inside the lava area using a point-in-polygon algorithm.