Acta et Commentationes, Sciences of Education, nr. 4 (18), 2019 ISSN 1857-0623
p. 104-108 E-ISSN 2587-3636

CZU: 004.02 DOI: 10.36120/2587-3636.v18i4.104-108

THE TRICK FROM ALIENS IN COMPETITIVE PROGRAMMING

Șerban CERCELESCU

International Computer High-School, Bucharest, Romania

Abstract. The scope of this article is presenting a very useful DP optimization technique, introduced in the competitive programming community with the problem Aliens at IOI 2016. The technique is used to reduce dimensions in particular DP configurations, by exploiting the convex nature of some cost functions. We will introduce the technique by starting with a simpler DP problem, show the optimization from $O(N^2)$ to $O(NlogVAL)$ then reveal the full solution of the original problem. Apparently, the official name of this optimization technique is "parameter search" and the Chinese community calls it "wqs binary search".

Keywords: dynamic programming, parameter search, competitive programming.

TRUCUL PROBLEMEI ALIENS ÎN PROGRAMAREA COMPETITIVĂ

Rezumat. Scopul acestui articol reprezintă ilustrarea unei tehnici de optimizare a DP foarte utilă, introdusă în comunitatea de programare competitivă odată cu problema Aliens la Olimpiada Internațională de Informatică din 2016. Tehnica este utilizată pentru a reduce dimensiunile în anumite configurații DP, prin exploatarea caracterului convex al unor funcții de cost. Vom introduce această tehnică începând cu o problemă DP mai simplă, vom arăta optimizarea de la $O(N^2)$ la $O(NlogVAL)$, apoi vom dezvălui soluția completă a problemei originale. Aparent, denumirea oficială a acestei tehnici de optimizare este „căutare de parametri", iar comunitatea chineză o numește „căutare binară wqs".

Cuvinte cheie: programare dinamică, căutare de parametri, programare competitivă.

A problem example: You are given an array *v* of integers (possibly negative) of length *N* ($\leq 10^5$) and a number *K* ($\leq N$). Select at most *K* disjoint sub-arrays of the initial sequence such that the sum of the elements included in the sub-arrays is maximized. The standard approach to such a problem would be a DP of the form:

$dp[n][k]$= ["*the solution for an array with the first n elements of the given array and k sub-arrays to be taken*"] where

$$dp[n][k] = max\{dp[n-1][k], max_{i=k}^{n-1}\{dp[i-1][k-1] + \sum_{k=i}^{n} v_k\}\}$$

Implementing this recurrence directly would be $O(N^4)$, supposing that *K* is comparable in size to *N*. It is left as an exercise to the reader to find a way of optimize this recurrence to $O(N^2)$. The trick behind the "aliens optimization" is that we can add a cost (penalty) which we will denote by λ for each taken sub-array. If λ=0, then the solution would be taking a sub-array for each positive element, but by increasing the value of λ, the optimum solution shifts to taking fewer sub-arrays. Now we just have to find a λ that allows us to take as many sub-arrays as possible, but still fewer that *K*. To do a small recap, λ is the cost we assign to adding a new sub-array, and increasing λ will decrease the number of sub-arrays in an optimal solution or keep it the same, but never increase it. That suggests that we could just binary search the smallest value of λ that yields an optimal solution with less than *K* elements.

$dp_\lambda[n]$=[“*The solution for the prefix of length n of our initial array v, where adding a sub-array comes with cost* λ”]

$$dp_\lambda[n] = max\{dp_\lambda[n-1], max_{i=1}^{n-1}\{\sum_{k=i} v_k + dp_\lambda[i-1] - \lambda\}\}$$

Besides just the dp, we will store another auxiliary array:

$cnt_\lambda[i]$ = [“how many sub-arrays does $dp_\lambda[n]$employ in its solution”]

These recurrences are easily implementable in linear time using partial sums and maxima. The pseudocode behind all of it would go something like this:

```
minbound = 0, maxbound = 1e18
while maxbound - minbound > ε:
   λ = (maxbound + minbound) / 2
   #compute dp and aux for λ
   if cnt[n] <= k:
      minbound = λ
   else:
      maxbound = λ
#compute dp and cnt for the final λ (= final minbound)
return dp[n] + cnt[n] * λ #note that if there are less than k positive values, then cnt[n] < k
```

Proof and Formal Requirements: In the case of our initial problem, the fact that increasing λ never increases the number of sub-arrays taken was probably a very intuitive fact, but we'd like to find an actual proof that this works and find a general criterion for using the peak setting optimization in reducing DP dimensions. This criterion is in a way concavity (or convexity). Let's denote by *ans*[*k*] the answer for the problem, but using exactly *k* sub-arrays. The key observation in proving that our solving method is correct is that the *ans*[*k*] sequence is concave, that is $ans[k] - ans[k-1] \leq ans[k-1] - ans[k-2]$. A more natural way of thinking about this and the actual way most people “feel” the concavity/convexity is by interpreting it as *if I have k sub-arrays and add another one, it will help me more than if I had k* + 1 *sub-arrays and added another one*.

Now let's see how this concavity helps us prove the correctness of our solution. Suppose λ=0. Our solution will just find the global maximum of our concave sequence, be it *ans*[*k*]. Notice that no matter the value of λ, the fact that our sequence is concave won't change. Let's shift our attention for a bit from concave sequences to concave functions. $f(x) = \lambda x - x^2$ is a fine example. By changing λ, we can move the peak of the function wherever we want and the function will remain concave.

Now let's go back to our more “discrete” sequence. We have an algorithm that finds *p* and *ans*[*p*] such that *ans*[*p*] is the maximum of the sequence, but we don't want the maximum of the sequence, we want *ans*[*k*] for some given *k*. So... we can force *k* to be the maximum of the sequence, by adding a linear function to our sequence (*ans*[*k*] → *ans*[*k*] + λ*k*), just as we changed the peak of our continuous function, we can forcefully change the peak of our sequence, which is exactly what we did in our solution.

The algorithm will yield that the maximum of the sequence is at *k* with the value *ans*[*k*] + λ*k* and we just need to subtract λ*k* to obtain our desired value: *ans*[*k*]. As for the

general criterion, you might have already guessed it: if $(ans[k])_{1 \le k \le n}$ is the sequence of answers for given *k*s, the sequence must be convex or concave, that is:

$$\forall i \in (1 \ldots n), ans[i] - ans[i-1] \le ans[i+1] - ans[i]$$

or

$$\forall i \in (1 \ldots n), ans[i] - ans[i-1] \ge ans[i+1] - ans[i]$$

Reconstruction Issues: Let's get back to our initial problem (there are *N* integers, you have to choose *K* sub-arrays etc...) and let's change the statement, instead of selecting at most *K*, you have to select exactly *K* sub-arrays. The difference is quite subtle, and the actual result is different iff there are less than *K* non-negative integers in the sequence. In this case, we just have to replace return $dp[n] - \lambda * aux[n]$ with return $dp[n] - \lambda * k$. This may seem weird and quite unintuitive as for why it works. Let's look at a few proprieties of our algorithm. First of all, it may not be the case that for each *k* we have a corresponding set of lambdas, that is: if for a given *p*, the maximum λ for which taking *p* objects is optimal, then the solution for $\lambda \leftarrow \lambda + \varepsilon$ where ε is an arbitrarily small value, may use more than *p*+1 objects, i.e. there may be no choice of λ for which the optimal solution employs a fixed number of elements. This may seem as a small game-killer for our technique, but let's look at the cause of this issue. Looking back at the Proof paragraphs, we are given the condition:

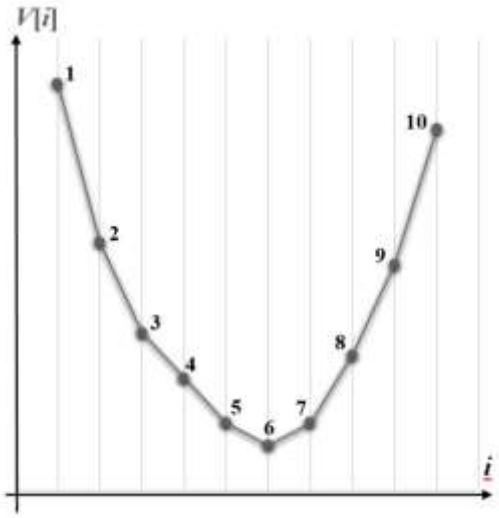
$$\forall i \in (1 \ldots n), ans[i] - ans[i-1] \le ans[i+1] - ans[i]$$

In case of equality, we may have the following situation: $ans[i+1] = ans[i] + t$, $ans[i+2] = ans[i] + 2t$etc. If the λ we choose equals *t*, then all of these solutions will seem equivalently "good". In fact, if a sub-array of solutions $ans[a], ans[a+1], \ldots, ans[b]$ for an arithmetic progression, there is no choice of λ that finds any other optimal solution other than using *a* or *b* objects. However, the fact that our solution fails only on possible arithmetic progressions from our sequence (i.e. if the sequence is not *strictly convex*) is the very thing that will help us solve this issue. Suppose we find the smallest lambda that makes the solution employs $\le k$ objects (let's say it uses *a* objects). This means the answer using exactly *p* objects is $dp[p] - \lambda p$, but this basically implies that between *p* and *k* (the fixed number of objects we want to use) there is an arithmetic progression (i.e. $ans[k] = ans[p] + t*(k-p)$). So if the answer for *p* would be $dp[n] - p\lambda$, then the answer for *k* must be $dp[n] - p\lambda - (k-p)\lambda$ (which equals $dp[n] - \lambda k$). This is quite weird as by finding a solution for *p*, we also find the answer for *k*, even if $(aux[n] = p) \ne k$. The downside of this workaround, is that even if we can find the value of the answer, a general method of reconstructing the solution (finding out what sub-arrays should we select) may not always exist.

Integral Lambda Search: You might have noticed that we are binary searching a floating point λ, not an integral valued one. The reason is that if the prerequisites of applying the optimization are satisfied, then we have proved that a λ exists, not that it

would have an integral value. The thing is, in most DP problems, the optimization works just as well with integers. It's just not that obvious to prove why.

Let's consider a convex sequence of N elements, call it $V_{1...N}$. Now let's consider a set of points $P=\{(i, p_i) \mid i \in [1...N]\}$, once drawn, together with the line segments between consecutive points (which will bear great importance in the following steps), you will see a convex/concave lower/upper hull.



A key observation now is that when looking at our convex sequence geometrically, the "peak" of the sequence will be the unique point that has segments with different signs of the slope to its left and to its right (with the exception the edge cases where the optimum is the first or last element of the sequence). In our drawn example, that peak is the 6th point, with a segment with negative slope value on its left and positive on its right. Another useful observation is that if we have two lines$y = a_1x + b_1$and $y = a_2x + b_2$, adding a constant value λ to both their slopes doesn't change the x coordinate of their intersection abscise.

In our context, this means that if we add a constant value to all the slopes of the segments, the intersection points will remain the same, so the peak x coordinate will still be integral. Now all that is left to do is "say" this: if we want to force a position t to be the global optimum of this sequence and the slope of the segment to the left of the point is l and the slope of the segment right of the point is r and $\{l,r\} \in Z$, then there exists at least one value $\lambda \in Z$ such that $l + \lambda$ and $r + \lambda$ have different signs. Translating this directly into terms of our convex sequence of answers, where our "slopes" are just the differences between two adjacent answers (i.e. $slope[k] = ans[k + 1] - ans[k]$), if the values of ans are integral, then obviously the differences (slopes) will also be integral, so if the answers to our problem are integral, then we can always binary search λ as an integral value.

Aliens – IOI 2016: I find it quite amusing that when the IOI introduces some totally new technique for 99% of its contestants (like the convex hull trick with the problem *batch scheduling* at IOI 2002), the technique is usually merely a sub-problem of a task that is

quite difficult on its on own, even without the fact that the contestant is required to rediscover some then obscure technique.

So is the case with Aliens (IOI 2016), even if you know the optimization, it's still quite a tricky convex hull trick problem.

The solution goes something like this:

First of all, notice that if a point lies below the main diagonal, we can just replace it with its transpose (i.e. $(x, y) \rightarrow (y, x)$), as any photo that captures (x, y) will also capture (y, x). After this transformation, notice the fact that we might be left off with a lot of useless points, that if removed will not change our answer. That is because if we have two points $p_2 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$ such that $x_1 \leq x_2$ and $y_1 \geq y_2$, any photo that captures p_2 will also capture $p1$, but not all photos capturing p_2 will capture p_2 and given that we must capture all points (so p_2 must be captured), we might as well remove $p1$ because it would have been captured by the photo containing $p2$ anyway. We can remove these useless points using a stack, and we'll be left with a sequence of points (x_1, y_1), (x_2, y_2),…, (x_b, y_b) that if sorted in increasing order by the x coordinate, the sequence will also be sorted in decreasing order by the y coordinate. From now on, we will consider the sequence of points in this order.

Let's define $dp[n][k]$ = {["*minimum area of a square that covers all points from t + 1 to n*"] +

$dp_\lambda[t]$ - ["*the area of the intersection between the square and the ones used in* $dp_\lambda[t]$"] + λ}, which is $dp_\lambda[n] = min_{t=1}^{n-1}\{(x_n - y_n + 1)^2 + dp_\lambda[t] - max(x_t - y_{t+1}, 0)^2 + \lambda\}$, which can be computed in linear time using the convex hull optimization.

Conclusions

"The aliens trick" is a very powerful optimization technique that most probably will soon be widespread in the competitive programming world, exploiting the convex nature of the solution space of some problems. I see this optimization, as a vital technique in the toolkit of all future and current competitive programmers.

Bibliography

1. Meenakshi K. R. Dynamic Programming for Coding Interviews: A Bottom-Up approach to problem solving. 1st Edition. Kindle Edition, 2017. 144 p.
2. Lew A., Mauch H. Dynamic Programming. A Computational Tool. Springer, 2007. 377 p.
3. https://ioinformatics.org/files/ioi2016problem6.pdf
4. https://www.hackerearth.com/practice/algorithms/dynamic-programming/introduction-to-dynamic-programming-1/tutorial/
5. https://www.tutorialspoint.com/data_structures_algorithms/dynamic_programming.htm