Blind Snake Problem

Artem Abaturov

1. Spiral Algorithm

First, let's discuss the naive algorithm, which involves traversing our table in a spiral pattern: 1 step to the right, 1 step down, 2 steps to the left, 2 steps up, and so on. This algorithm works quite well for "square" tables, but in the case of a table with dimensions $1 \times S$, in the worst case, we will traverse $O(S^2)$ cells.

You can check out an example of how this algorithm works in the GitHub repository.

2. Reformulation of the problem

Let us introduce a coordinate system with the point $(0, 0)$ at the initial position of the snake. Since our table is cyclic, we only need to make fewer than B steps to the right and fewer than A steps upward to reach the cell with the apple. Let x represent the minimum number of steps to the right, and y the minimum number of steps upward required to reach the cell with the apple. As we have observed, the following inequalities hold: $x < A$ and $y < B$. Then:

$$x \cdot y < A \cdot B$$

$$(x + 1) \cdot (y + 1) \leq A \cdot B$$

$$(x + 1) \cdot (y + 1) \leq S$$

This implies that if the apple is located at coordinate (x, y), the area of our table is at least $(x + 1) \cdot (y + 1)$. Therefore, we only need to visit cells for which $(x + 1) \cdot (y + 1) \leq S_{\max}$.

In the worst case if the apple is located at cell (x, y), then the area of our table is exactly $(x + 1) \cdot (y + 1)$. If we want our algorithm to always execute no more than $k \cdot S$ commands, the cell (x, y) must be visited in no more than $(x + 1) \cdot (y + 1) \cdot k$ steps.

3. Special Case

Before moving on to the main algorithm, let's solve the problem where we only need to visit cells in which either $x = 0$ or $y = 0$ in time.

3.1. First Algorithm

The first algorithm I want to consider works as follows:

1. Initially, set $i = 0$.
2. Take 2^i steps to the right.
3. Take 2^i steps to the left.
4. Take 2^i steps upward.
5. Take 2^i steps downward.
6. Increment i by 1 and go back to the second step if not all possible cells have been visited.

Now, let's estimate the optimal k for such an algorithm. It is clear that if we manage to visit all the cells vertically, then, due to the fact that we first visit the symmetric cells horizontally, we will eventually visit all the cells.

Consider a specific cell $(0, y)$. To visit it, we need to spend

$$4 \cdot \left(1 + 2 + \ldots + 2^{b-1}\right) + 2 \cdot 2^b + y, \text{ where } b = \lceil \log_2 y \rceil$$

$$4 \cdot \left(1 + 2 + \ldots + 2^{b-1}\right) + 2 \cdot 2^b + y \leq 6 \cdot 2^b + y \leq 6 \cdot 2y + y = 13y$$

So, we get

$$13y \leq (y+1) \cdot k$$

Thus, if we take $k = 13$, we will manage to visit all the cells in time. In other words, this algorithm finds the apple in $O(13S)$ steps. Moreover, it does not matter what $S_{\max}$ is.

3.2. Greedy Approach

Unlike the previous algorithm, let's fix k, which we aim to achieve. Now, our task is to visit the cell (x, y) in no more than $(x+1) \cdot (y+1) \cdot k$ steps. Clearly, we want to minimize revisiting the same cells. Conceptually, the algorithm will work as follows: move as far as possible in one direction (while ensuring we can return to move in the opposite direction), then go back and traverse as far as possible in the other direction, and so on. You can check out my implementation for a better understanding.

```python
def get_deadline(x):
    # Calculate the deadline for reaching a cell based on its position and constant k
    return (abs(x) + 1) * k


# Constant k determines the maximum allowed steps per cell area
k = 9
# S_max defines the maximum coordinate range
S_max = 10 ** 6
# Initialize boundaries of visited cells. left is equivalent to the point (0, 1), right is equivalent to the point (1, 0)
left = -1
right = 1
# Our current position. 0 is equivalent to the point (0, 0)
current = 0
# The number of taken steps
steps = 0
while left >= -S_max or right <= S_max:
    if current + 1 == right:
        # Determine if we need to switch direction or expand further to the right
        if current + 1 + abs(left) + steps + 1 > get_deadline(left) or right == S_max + 1:
            steps += current + abs(left)
            current = left
            left -= 1
        else:
            steps += 1
            right += 1
            current += 1
    else:
        # Determine if we need to switch direction or expand further upwards
        if abs(current - 1) + right + steps + 1 > get_deadline(right) or left == -S_max - 1:
            steps += abs(current) + right
            current = right
            right += 1
        else:
            current -= 1
            steps += 1
            left -= 1
    # Check if the deadline is met
    if get_deadline(current) < steps:
        print("K is too small")
        exit(0)
```

Through experimentation, it was determined that the minimum integer k for which this greedy algorithm visits all cells in time, given that $S_{\max} = 10^6$, is 9.

4. Fast Algorithm

Let's return to the original problem. The algorithm under consideration will be a generalization of the approach used in the previous section. Conceptually, we will also try to traverse as many cells as possible in one direction (while ensuring we can visit cells on the opposite side in time) and then switch to the other side and traverse as many cells as possible there.

We face the following challenge: determining when to visit cells where $x > 0$ and $y > 0$. My solution to this problem is as follows: at the moment when we switch to the other side, let's visit all cells whose deadline is less than or equal to $(\max(x_{\max}, y_{\max}) + 1) \cdot k$ and which we haven't visited yet, where $x_{\max}$ is the maximum x coordinate we've visited, and $y_{\max}$ is the maximum y coordinate we've visited.

Another question is how to ensure that we can visit the cells on the opposite side in time. Without loss of generality, let's assume we are at the cell $(x_{\max}, 0)$. The number of steps required to move to the cell $(0, y_{\max} + 1)$ can be estimated as:

$$x_{\max} + y_{\max} + 2 \cdot n + 1$$

where n is the number of unvisited cells whose deadline is less than or equal to $(\max(x_{\max}, y_{\max}) + 1) \cdot k$.

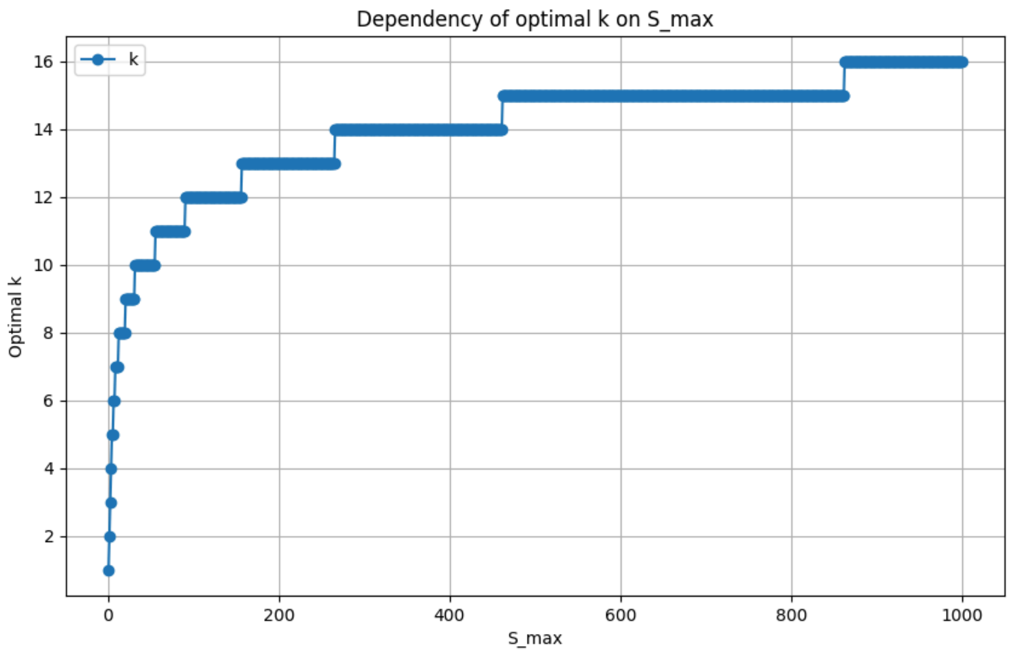
For better understanding, you can check out the implementation of the algorithm in the repository, as well as examples of how it works.

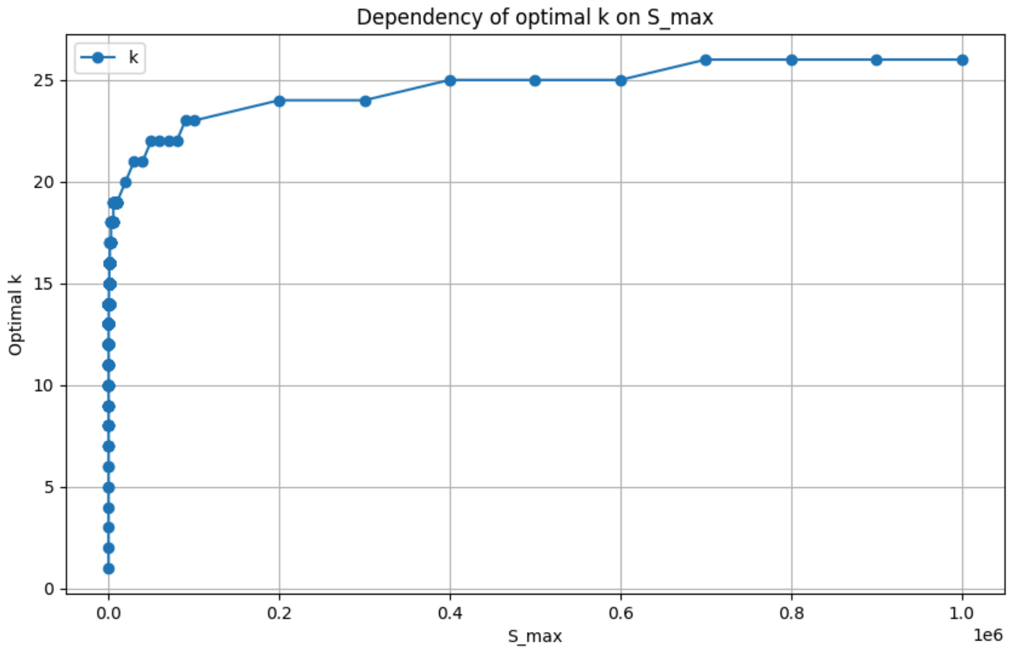
For $S_{\max} = 10^6$, this algorithm finds the apple in $O(26S)$ steps.

5. k-Optimizer

For a specific value of $S_{\max}$, it is not very difficult to find the optimal integer k for which our algorithm manages to visit all the cells in time. To do this, we can perform a binary search over k and check for each k whether we can visit all the required cells in time.

Here's how k depends on $S_{\max}$:


Dependency of optimal k on S_max
k
Optimal k
S_max
0
200
400
600
800
1000
2
4
6
8
10
12
14
16


Dependency of optimal k on S_max
k
Optimal k
S_max
0.0
0.2
0.4
0.6
0.8
1.0
1e6
0
5
10
15
20
25