Advanced Algorithms Problems and Solutions

Mattia Setzu Giorgio Vinciguerra

October 2016

Contents

1 Hogwarts

The Hogwarts School[1] is modeled as a graph $G = (V, E)$ where V is the set of castle's rooms and $E \subseteq V \times V$ is the set of the stairs. Each stair is labelled with the time of appearance and disappearance, and can be walked in both directions, therefore the graph is undirected. The goal is to find, if possible, the minimum amount of time required to go from the first to the last room.

1.1 Solution 1: Preprocessing-then-Dijkstra

Dijkstra is able to find the shortest path in a graph with non-negative weights on its edges. Our main idea is to create a Dijkstra compatible graph through a NORMALIZE function, then apply Dijkstra to it in order to find the shortest path. The core of the preprocessing is the NORMALIZE function which computes traversal times between nodes at a given time *time*:

```
function NORMALIZE($from$, $to$, $time$):
    $t \leftarrow \infty$
    if $start[v'] \leq t < end[v']$ then                ▷ No waiting time
        $t \leftarrow t + 1$
    else if $t < start[v']$ then                        ▷ Waiting time
        $t \leftarrow start[v'] + 1$
    else
        $t \leftarrow \infty$                            ▷ Available time already expired
    return $t$
```

The normalize function is then applied to a node traversal:

1.1.1 Pseudo-code

```
create vertex set $Q$ of unvisited nodes
create vertexes set $E'$ of edges weight
$time \leftarrow 0$                                      ▷ Initial time for traversal
$edges \leftarrow$ STAIRS_OF$(0)$   ▷ Get incoming/outgoing edges of the source node
function PROCESS($node$, $time$)
    if $edge \in visited\_edges$ then
        return
    $traversal\_time \leftarrow \infty$
    for all $neighbor \in neighbors\_of\_node$ do
        $traversal\_time \leftarrow$ TRAVERSAL_TIME($node$, $neighbor$, $time$)
        $E'[0][node] \leftarrow traversal\_time$   ▷ $E'[i][j]$ holds the weight/traversal
                                                 ▷ time for the stair between $i$ and $j$
        for all $new\_neighbor \in neighbors\_of\_neighbor$ do
            NORMALIZE($neighbor$, $new\_neighbor$, $traversal\_time$)
    if DIJKSTRA$(V, E') = \infty$ then
```

[1] `http://didawiki.cli.di.unipi.it/lib/exe/fetch.php/magistraleinformatica/alg2/algo2_16/hogwarts.pdf`

```
            return -1
        else
            return $t$
```

Computational cost: $\Theta(n^2)$ if the vertex set in DIJKSTRA is implemented as an array. $O(|E| + |V| \log |V|)$ with Fibonacci heap.

1.2 Solution 2: HogwartsDijkstra

```
function HOGWARTSDIJKSTRA($G$):
    create vertex set $Q$ of unvisited nodes
    for all vertex $v \in V$ do                              ▷ initialization
        $time[v] \leftarrow \infty$                          ▷ unknown time from source to v
        add $v$ to $Q$                                       ▷ all nodes initially in Q
    $time[0] \leftarrow 0$                                   ▷ time from source to source
    while $Q \neq \emptyset$ do
        $u \leftarrow x \in Q$ with $\min\{time[x]\}$
        remove $u$ from $Q$
        for all neighbor $v$ of $u$ do:
            if $time[u] \leq appear[v]$ then
                $alt \leftarrow appear[v] + 1$               ▷ wait the appearance of the stair
            else if $time[u] < disappear[v]$ then
                $alt \leftarrow time[u] + 1$                 ▷ use the stair
            else
                $alt \leftarrow \infty$                      ▷ the stair has already disappeared
            if $alt < time[v]$ then
                $time[v] \leftarrow alt$                     ▷ a quicker path to $v$ has been found
    return $time[|N| - 1]$
```

Computational cost. See the previous section.

1.3 Solution 3: BFS-like traversal

```
function REACH($N$, $M$, $A[]$, $B[]$, $appear[]$, $disappear[]$)
    for $i = 0$ to $M - 1$ do
        $edges\_[A[i]].push\_back(make\_pair(i, B[i]))$
        $edges\_[B[i]].push\_back(make\_pair(i, A[i]))$
    for $i = 0$ to $N - 1$ do
        $done\_[i] \leftarrow false$
        $distance\_[i] \leftarrow \infty$
    $reached\_[0].push\_back(0)$
    $distance\_[0] \leftarrow 0$
```

```
10:  for $t = 0$ to $MAX\_TIME$ do
11:      for all $v \in reached\_[t]$ do
12:          if not $done\_[v]$ then
13:              for all $edge \in edges\_[v]$ do
14:                  $staircase \leftarrow edge.first$
15:                  $neighbor \leftarrow edge.second$
16:                  $time \leftarrow \max(distance_{[}v], appear[staircase]) + 1$
17:                  if not $done\_[neighbor]$
18:                                      and $distance\_[v] < disappear[staircase]$
19:                                        and $time < distance\_[neighbor]$ then
20:                      $distance\_[neighbor] \leftarrow time$
21:                      $reached\_[time].push\_back(neighbor)$
22:              $done\_[v] \leftarrow true$
23:  return $(distance\_[N-1] = \infty)? -1 : distance\_[N-1]$
```

Computational cost: $O(m + MAX\_TIME)$.

2 Paletta

Paletta ordering[2] is a peculiar ordering technique: given a 3-tuple of elements, paletta takes the central element as pivot and swaps the two elements right before and next to it. To make an example:

$$(3, 2, 1) \xrightarrow{paletta} (1, 2, 3)$$

We now want to develop an algorithm to order any array through paletta ordering with the minimal number of swaps. You should see as not every array can be ordered (e.g. [1, 3, 2]).

2.1 Solution 1: Split and count-inversions

We should note that the following properties hold:

1. Every element can be a pivot, but the first and the last one, as they have respectively no elements before and after them.

2. Every element can be swapped as many times as necessary, but only with elements of the same 2-remainder (numbers in even positions can only be swapped with numbers in even positions, the same holds for odd indexes). More formally, if n is the size of the array A we want to sort, $i, j \in [1, n-2]$, $A[i]$ can be swapped with $A[j]$ if and only if $i \equiv j \pmod 2$.

[2]`http://didawiki.cli.di.unipi.it/lib/exe/fetch.php/magistraleinformatica/alg2/algo2_16/paletta.pdf`

3. The least number of swaps does not backtrack any element. Formally, let k be the minimal number of swaps applied to an array, backtracks included. By hypothesis, k is minimal, but at least m, $m > 0$ backtrack swaps have been operated, therefore we found a $k' = k - m : k' < k$, a new minimal number of swaps: contradiction.

Given item 2, we can split our array in two, even and odd numbers, and order them counting the swaps. In our example we'll use *mergesort*, as it runs in $O(n \log n)$, does backtrack elements, and is very well-known. Clearly, given an array, a swap happens when an element is pushed back, pulling the one between its new position and the old one ahead: we can map this behaviour in the merge routine of mergesort: the array merged is able to push back elements from its right pointer to the new array, moving them back of $(m-i)+(j-m)$ positions, where m is the dimension of the current two sub-arrays to merge. Provided that our edited version of mergesort ran successfully on both the even-index and odd-index, we now need to verify if by merging them we obtain an ordered array. Intuitively, the merged array will start with the first element of the even-index arrays, followed by the first of the odd-index array, followed by the second of the even-index array, and so on. To check for these elements is pretty trivial and can be done in linear time. Follows the pseudo-code for the edited version and SNAKE_CHECK function:

```
function MERGE_WITH_PALETTA($left$, $right$, $k$):
    ...                                              ▷ merge instructions
    if $right > left$ then
        $paletta\_count \leftarrow paletta\_count + 1$
        ...
function SNAKE_CHECK
    $even, odd \leftarrow 0$
    for ;$even, odd < N$; $even = even + 1, odd = odd + 1$ do
        if $a[even] > a[odd]$ then
            return $-1$
    return $paletta\_count$
```

Computational cost: $\Theta(n \log n)$.