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*:

```
1:  function NORMALIZE(from, to, time):
2:      t = ∞
3:
4:      if start[v'] ≤ t < end[v'] then                ▷ No waiting time
5:          t = t + 1
6:
7:      else if t < start[v'] then                     ▷ Waiting time
8:          t = start[v'] + 1
9:
10:     else
11:         t = ∞                                      ▷ Available time already expired
12:     end if
13: return t
14: end function
```

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

The normalize function is then applied to a node traversal:

1.1.1 Pseudo-code

```
1: create vertex set Q of unvisited nodes
2: create vertexes set E' of edges weight
3: time = 0                                            ▷ Initial time for traversal
4: edges = stairs_of(0)          ▷ Get the incoming and outcoming edges of the
   starting node
5: function PROCESS(node, time)
```

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

```
6:
7:      if edge ∈ visited_edges then return
8:      end if
9:      traversal_time = ∞
10:     for each neighbor ∈ neighbors_of_node do
11:         traversal_time = traversal_time(node, neighbor, time)
12:         E'[0][node] = traversal_time        ▷ E'[i][j] holds the weight/traversal
13:                                             ▷ time for the stair between i and j
14:         for each new_neighbor ∈ neighbors_of_neighbor do
15:             normalize(neighbor, new_neighbor, traversal_time)
16:         end for
17:     end for
18:     dijkstra_graph = {V, E'}
19:     t = dijkstra(dijkstra_graph)
20:
21:     if t == ∞ then return -1;
22:     elsereturn t;
23:     end if
24: end function
```

Computational cost. See the previous section.

1.2 Solution 2: HogwartsDijkstra

```
1: function HogwartsDijkstra(G):
2:     create vertex set Q of unvisited nodes
3:     for each vertex v ∈ V do                                  ▷ initialization
4:         time[v] ← ∞                                ▷ unknown time from source to v
5:         add v to Q                                          ▷ all nodes initially in Q
6:     end for
7:     time[0] ← 0                                          ▷ time from source to source
8:     while Q ≠ ∅ do
9:         u ← x ∈ Q with min{time[x]}
10:        remove u from Q
11:        for each neighbor v of u do:
12:            if time[u] ≤ appear[v] then
13:                alt ← appear[v] + 1          ▷ wait the appearance of the stair
14:            else if time[u] < disappear[v] then
15:                alt ← time[u] + 1                                ▷ use the stair
16:            else
17:                alt ← ∞                    ▷ the stair has already disappeared
18:            end if
19:            if alt < time[v] then
20:                time[v] ← alt              ▷ a quicker path to v has been found
```

```
            end if
        end for
    end while
    return time[|N| − 1]
end function
```

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]))
    end for
    for i = 0 to N − 1 do
        done_[i] ← false
        distance_[i] ← ∞
    end for
    reached_[0].push_back(0)
    distance_[0] ← 0
    for t = 0 to MAX_TIME do
        for v ∈ reached_[t] do
            if not done_[v] then
                for edge ∈ edges_[v] do
                    staircase ← edge.first
                    neighbor ← edge.second
                    time ← max(distance_[v], appear[staircase]) + 1
                    if not done_[neighbor]
                                        and distance_[v] < disappear[staircase]
                                          and time < distance_[neighbor] then
                        distance_[neighbor] ← time
                        reached_[time].push_back(neighbor)
                    end if
                end for
                done_[v] ← true
            end if
        end for
    end for
    return (distance_[N − 1] == ∞)? − 1 : distance_[N − 1]
end function
```

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:

$$\{3,\ 2,\ 1\} \xrightarrow{paletta} \{1,\ 2,\ 3\} \tag{1}$$

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 like this:

$$\{3,\ ,\ 1\} \xrightarrow{paletta} \{1,\ 2,\ 3\} \tag{2}$$

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 respectevly 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 $|a| = N, \forall i \in [1, N-2], \neg\exists swap(i,j) : i \mod 2 \neq i \mod j$
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 $\log_2 n$, 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:

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

```
function MERGE_WITH_PALETTA(left, right, k):
    ...                                                  ▷ merge instructions

    if right > left then
        paletta_count = paletta_count + 1
        ...
    end if
end function
function SNAKE_CHECK:
    even, odd = 0
    for ;even, odd < N even = even + 1, odd = odd + 1 do

        if a[even] > a[odd] then
            return -1
        end if
    end for
     return paletta_count
end function
```

Computational cost: $\Omega(n \log_2(n)), \Theta(n \log_2(n)), O(n \log_2(n))$.

3 Range updates

Consider an array C of n integers, initially all equal to zero. We want to support the following operations:

- **update(i, j, c):** where $0 \leq i \leq j \leq n-1$ and c is an integer: it changes C such that $C[k] = C[k] + c$ for every $i \leq k \leq j$.
- **query(i)** where $0 \leq i \leq n-1$: it returns the value of $C[i]$.
- **sum(i,j)** where $0 \leq i \leq j \leq n-1$: it returns $\Sigma_{k=1}^{j}(C[k])$.

Design a data structure that uses $O(n)$ space and implements each operation above in $O(\log(n))$ time. Note that $query(i) = sum(i, i)$ but it helps to reason. [Hint to further save space: use an implicit tree such as the Fenwick tree (see wikipedia).]

3.1 Solution 1: Fenwick lazy a-b sums

Let T be a segmented binary tree over a continuous interval $I : [0, N-1]$ s.t. its leafs are the points in I, and the parent of two nodes comprises of their interval:

$$n' \cup n'' = n, n' \cap n'' = \emptyset \text{ s.t. n is the father of n', n"}$$

T will keep track of the prefix sums for every interval. We define a function

$$s' : [0, n-1] \to \mathbb{N} \tag{3}$$

that given a node in T returns the value associated with I, namely the cumulative sum of that interval.

In order to reduce the computational cost, we introduce a lazy algorithm that doesn't propagate sums over T as they are streamed in the input, which means $s'(i)$ might not be accurate at a given time t for any of the requested operation.

We'll instead either compute over T or update T as necessary. Let us define a function to do so:

$$l : \mathbb{N} \to (\mathbb{N} \cup \{\epsilon\}, \mathbb{N}) \tag{4}$$

to keep track of our lazy sums:

$$s(n) = \begin{cases} \epsilon, \_ & \text{if no lazy prefix sum is in that interval} \\ k, m & \text{if a lazy sum of k is to be propagated to m} \end{cases}$$

The *query* function is then trivial:

```
1: function QUERY(I, i, sum):
2:
3:     if I.size == 1 then                          ▷ Return found value
4:         return I.sum
5:     end if
```

```
6:
7:      if lazy(I), i ∈ I.left, i¬ ∈ I.right then                ▷ Lazy on left child
8:          lazy(I) = False
9:          query(I.left, i, sum + I.sum)
10:     end if
11:
12:     if lazy(I), i ∈ I.right, i¬ ∈ I.left then                ▷ Lazy on right child
13:         lazy(I) = False
14:         query(I.right, i, sum + I.sum)
15:     end if
16:
17:     if lazy(I), i ∈ I.right, i ∈ I.left then                 ▷ Lazy on both
18:         lazy(I) = False
19:         query(I.right, i, j, sum+I.sum) + query(I.left, i, j, sum+I.sum)
20:     end if
21:
22:     if !lazy(I), i ∈ I.left then                             ▷ Not lazy on left child
23:         query(I.left, i, sum)
24:     end if
25:
26:     if !lazy(I), i ∈ I.right then                            ▷ Not lazy on right child
27:         query(I.right, i, sum)
28:     end if
29:
30:     if !lazy(I), i ∈ I.right, i ∈ I.left then                ▷ Not lazy on both
31:         sum(I.right, i, sum)
32:     end if
33: end function
 1: function SUM(I, i, j, sum):
 2:
 3:     if I.size == 1 then                                      ▷ Return found value
 4:         return I.sum + sum
 5:     end if
 6:
 7:     if lazy(I), i ∈ I.left, i¬ ∈ I.right then                ▷ Lazy on left child
 8:         lazy(I) = False
 9:         sum(I.left, i, j, sum + I.sum)
10:     end if
11:
12:     if lazy(I), i ∈ I.right, i¬ ∈ I.left then                ▷ Lazy on right child
13:         lazy(I) = False
14:         sum(I.right, i, j, sum + I.sum)
15:     end if
16:
17:     if lazy(I), i ∈ I.right, i ∈ I.left then                 ▷ Lazy on both
```

```
        lazy(I) = False
        sum(I.right, i, j, sum + I.sum) + sum(I.left, i, j, sum + I.sum)
    end if

    if !lazy(I), i ∈ I.left then                          ▷ Not lazy on left child
        sum(I.left, i, sum)
    end if

    if !lazy(I), i ∈ I.right then                         ▷ Not lazy on right child
        sum(I.right, i, sum)
    end if

    if !lazy(I), i ∈ I.right, i ∈ I.left then             ▷ Not lazy on both
        sum(I.right, i, sum)
    end if
end function
function UPDATE(I, i, j, k):

    if I.size == 1 then                                   ▷ Return found value
        return I.val+ = update
    end if

    if lazy(I), i ∈ I.left, i¬ ∈ I.right then             ▷ Lazy on left child
        lazy(I.left) = True
        I.left.val = k
    end if

    if lazy(I), i ∈ I.right, i¬ ∈ I.left then             ▷ Lazy on right child
        lazy(I.right) = True
        I.right.val = k
    end if

    if lazy(I), i ∈ I.right, i ∈ I.left then              ▷ Lazy on both
        lazy(I) = True
        I.val = k
    end if

    if !lazy(I), i ∈ I.left then                          ▷ Not lazy on left child
        update(I.left, i, update)
    end if

    if !lazy(I), i ∈ I.right then                         ▷ Not lazy on right child
        update(I.right, i, update)
    end if
```

```
30:     if !lazy(I), i ∈ I.right, i ∈ I.left then          ▷ Not lazy on both
31:         update(I.right, i, update)
32:     end if
33: end function
```

4 Depth of a node in a random search tree

A random search tree for a set S can be defined as follows: if S is empty, then the null tree is a random search tree; otherwise, choose uniformly at random a key $k \in S$: the random search tree is obtained by picking k as root, and the random search trees on $L = x \in S : x < kandR = x \in S : x > k$ become, respectively, the left and right subtree of the root k. Consider the randomized QuickSort discussed in class and analyzed with indicator variables CLRS 7.3, and observe that the random selection of the pivots follows the above process, thus producing a random search tree of n nodes. Using a variation of the analysis with indicator variables, prove that the expected depth of a node (i.e. the random variable representing the distance of the node from the root) is nearly $2\log_2(n)$. Prove that the probability that the expected depth of a node exceeds $2\log_2(n)$ is small for any given constant $c > 1$. [Note: the latter point can be solved after we see Chernoff's bounds.]

4.1 Recursive balanced proof

Let n be the number of nodes in the input list l, $h = \log_2(n)$ the height of a balanced tree over l, $T(p)$ the tree built over the permutation p of pivots, $d(m)$ be the positional distance of a value m of a partition from the median value of the said partition. Then the following holds:

- $height(T) = h \iff |T.left| = |T.right| \pm 1$ Trivially, let r be the root of a 3-nodes partition: then, if the partition is unbalanced, the lesser one will comprise of 0 nodes, while the greater one of 2, which implies that $height(T.right) == 2$.

- P = pivot, $d(m) = \pm k \implies height(T.left) = height(T.right) \pm k$. Recursively from the previous statement, a partition unbalanced of one element generates subtrees whose levels differ on a factor of 1. By iterating recursively, their subtrees, if unbalanced by 1, will yield one more level difference. Over k unbalanced pivots on a single subtree, at most k levels will be added to h.

- By the previous statement, it follows that $\nexists T, T' : height(T) >= height(T')$, T balanced, T' unbalanced. As stated, let T', T be the unbalanced/balanced tree respectively; let us cheat with T and switch the root pivot with the first element in its subtree. Now, let us prove by contradiction that T can't stay balanced and that its height will increase. By shifting the tree to the left we have deprived $T.right$ of either 0 levels (in case $T.right$ is able to switch every pivot in its tree with its right subtree root, ending with the rightmost leaf in its subtree) or 1, in case no rightmost leaf is present. Therefore $height(T) <= height(T')$.

- The completely unbalanced tree is the tree with the most levels. By taking partitions of size 0 we costantly force, at each level, one subtree to

disappear. Therefore, its level(s) has to be necessarly transferred to its brother. We then have exactly one node per level, therefore n levels.

Behaviour on random permutations Now let us analyse how the tree depth varies according to random pivot selection. We start by applying the 4.1k-distance to a tree T with $n = 3$ nodes. Trivially, $height(T)$ with balanced tree is equal to two. Now, let us pick either the lowest or the greatest pivot possible: the tree is unbalanced towards either the left or the right, but $height(T) = 2$ in both cases. As the reader can see from 4.1, the distance works in absolute value; it is then clear how, at every permutation for a pivot p, out of the n, there are 2 that generate a tree of the same height: $p = d(P) + k, p = d(P) - k$. Given that at every iteration a node x in a completely unbalanced tree T' has a probability of $\frac{1}{n-i}$, we can define the probability of x being a pivot at level l as:

$$P(x_k) = \frac{1}{n-l} \tag{5}$$

Now, in order for x not to be chosen as pivot in the previous $l-1$ levels we have:

$$P(x_k) = \Sigma_{k=1}^{l-1}(\frac{1}{n-l+1}) \tag{6}$$

Given the height of T, the (harmonic) partial series converges to $\ln(n)+1$. Let us now add a root r s.t. $T'.right = T, T'.left = T$. We now have to consider the mirror case $\ln(n')+\ln(n')$, given by the previous $n' = n/2$ in the logarithm, since the number of nodes doubled, the $+1$ removed for both, since now neither of $T'.left, T.right$ is the root, and a $+1$ added since a new level has been added.

4.2 Upper bound

By hypothesis,

$$E[d(x) > 2c\ln(n)] <<< 1 \tag{7}$$

By definition the ancestor of a node i are indipendent random variables, and we can apply the Chernoff bounds over the set $x : d(x) >= 2\ln(n)$ of random variables determining the expected distance of nodes.

$$\mathsf{P}[X \geq c\mathsf{E}[X]] < e^{-c\ln(\frac{c}{e})\mathsf{E}[X]}$$

Let us consider $X = 1 \forall i == \ln(n)$, the expected depth of $\ln(n)$, then

$$\mathsf{P}[X \geq c\ln(n)] < e^{-c\ln(\frac{c}{e})\ln(n)}$$