Queues

Manoj Gupta

August 15, 2016

Consider the following problem in which you are given a grid of $n \times n$ cells. Some of the cells in the grids are obstacles, that is, one cannot pass through these cells. Additionally, we are given the starting location s (cell $(0, 0)$) and the ending location t (some cell (i, j)). The aim is to find the shortest distance from s to t avoiding obstacles. Note that from a location, one can only move to right, left, above or below.

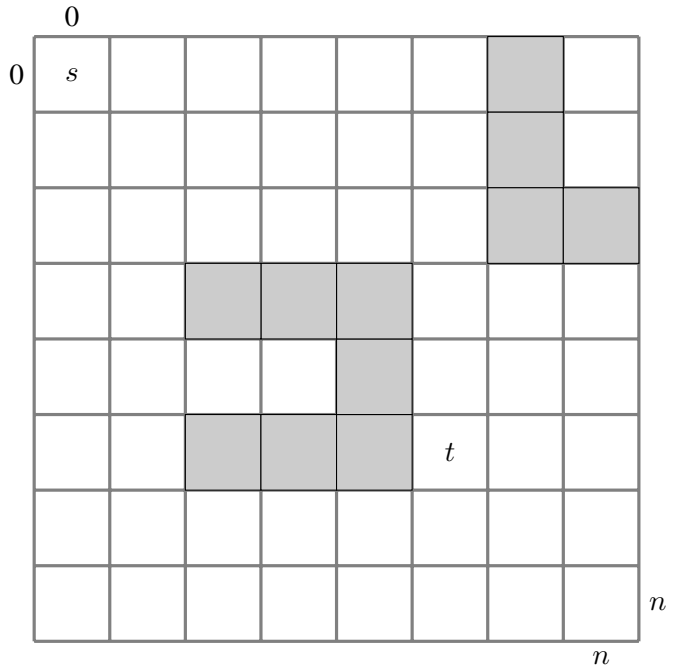

Figure 1: A grid with obstacles with a starting location s and an ending location t

A very naive algorithm for the above algorithm moves from the starting location one step at a time. So after the first step, we can reach two nodes adjacent to s. Thus we have found out all the cells that are at a distance 1 from s. After this iteration, we take all the cells that are at a distance 1 from s and repeat the process. Thus at the end of second iteration, we will reach all the nodes at distance 2 from s (if such a cell is not an obstacle). An algorithm based on the above observation can be found in Figure 2. One can prove that this procedure correctly finds the shortest distance from s to t avoiding obstacles. We will focus on the running time of our algorithm.

Consider the procedure FIND-NEXT-LAYER($\mathcal{L}_i$). This procedure processes all the cells in layer $\mathcal{L}_i$ and adds any one of its adjacent cell b to layer $i+1$ if (1) b is not an obstacle and (2) b has not been reached before (or $distance[b] = \infty$). Note that the inner for loop runs for 4 iterations as there are exactly 4 neighbors of each cell. The outer for loop runs for $|\mathcal{L}_i|$ iterations. So, the running time of FIND-NEXT-LAYER($\mathcal{L}_i$) is $O(|\mathcal{L}_i|)$.

The main programs initializes distances of all the cells to ∞. Note that distance variable is used in FIND-NEXT-LAYER($\cdot$) to deduce whether a cell is reached for the first time or not. The time taken to set the distance is $O(n^2)$. We then set $\mathcal{L}_0$ to $\{s\}$ as there is only one cell at a distance 0 from s, that is s itself. We then use FIND-NEXT-LAYER($\cdot$) iteratively to find the set $\{\mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_3 \dots\}$. The running time of the main depends on the number of times FIND-NEXT-LAYER($\cdot$) is called. In any case the running time can be bounded as follows: $O(n^2 + \sum_{i=0}^{\infty} |\mathcal{L}_i|)$, where the first term is for the initialization of distances and the second term is for summation of time taken by FIND-NEXT-LAYER($\cdot$).

```
1 L_{i+1} ← ∅;
2 foreach cell c in L_i do
3 |   foreach neighbor b of c such that b is not
  |   an obstacle do
4 |   |   if distance[b] = ∞ then
5 |   |   |   distance[b] ← i + 1;
6 |   |   |   L_{i+1} ← L_{i+1} ∪ {b};
7 |   |   end
8 |   end
9 end
```

(a) FIND-NEXT-LAYER($\mathcal{L}_i$)

```
1 foreach cell c in the grid do
2 |   distance[c] ← ∞
3 end
4 L_0 ← {s};
5 i ← 0;
6 while L_i is not empty or L_i does not contain
  cell t do
7 |   FIND-NEXT-LAYER(L_i);
8 end
```

(b) The main program

Figure 2: solving the grid problem

The main task then lies in finding the value of $\sum_{i=0}^{\infty} |\mathcal{L}_i|$. To this end, we use the following observation.

Observation 0.1. *Each cell can be a part of exactly one layer $\mathcal{L}_i$.*

Since each layer $\mathcal{L}_i$ can only contains cells, the above observation implies that $\sum_{i=0}^{\infty} |\mathcal{L}_i| = O(n^2)$. This implies that the running time of our main algorithm is $O(n^2)$.

We can ask whether there exists any other algorithm with better running time. However, that is not the main focus of this topic. We are bothered by another implementation issue: Can we design an algorithm that uses only one list (rather than many in our trivial algorithm)? To this end, we make the following important observation:

Observation 0.2. *In the main function, cells in layer $\mathcal{L}_{i+1}$ is processed only after all cells in layer $\mathcal{L}_i$ are processed.*

If we have to use just one list, then we have to add cells in $\mathcal{L}_{i+1}$ at the end of the list and process each cell from the start of the list. This is because the start of the list contains cell $\mathcal{L}_i$ and we must process them before processing cells in $\mathcal{L}_{i+1}$ which are added at the end of the list. In fact, the processing is done in FIFO - First In First Out - order. Such a data-structure is called a QUEUE.

A queue must support the following operations

1. CREATE-QUEUE(): Create an empty queue.
2. ENQUEUE(a) : Add a at the end of the queue.
3. DEQUEUE(): Remove an element from the front of the list.

We can implement queues using a linked list. We maintain two pointers: $front$ ad $rear$. Elements will always be added to the rear of the list and will be removed from the front of the list. One can check that the time taken for ENQUEUE(a) and DEQUEUE() takes $O(1)$ time.

```
1 front ← null;
2 rear ← null;
```

(a) CREATE-EMPTY()

```
1  allocate memory to a new node v;
   v.value ← a;
2  v.next ← null;
3  if rear = null then
   |  /* Queue is empty    */
4  |  front ← v;
5  |  rear ← v;
6  end
7  else
8  |  rear.next ← v;
9  |  rear ← v;
10 end
```

(b) ENQUEUE(a)

```
1  if front = null then
2  |  print "Queue Empty";
3  end
4  else
5  |  v ← front;
6  |  a ← v.value;
7  |  if front = rear then
   |  |  /* Queue contains
   |  |     onely one
   |  |     element          */
8  |  |  front ← null;
9  |  |  rear ← null;
10 |  end
11 |  else
12 |  |  front ← front.next;
13 |  end
14 end
15 deallocate the memory of v;
16 return a;
```

(c) DEQUEUE()

Figure 3: Implementation of Queues

Having implemented queues, we now proceed to implement a elegant solution to the shortest distance problem on the grid where we use queues to seamlessly move from layer i to layer $i + 1$. One can check that the running time of our algorithm is still $O(n^2)$ as each cell is inserted and removed from the queue at most once (similar to Observation 0.2)

```
1  CREATE-EMPTY();
2  foreach cell c in the grid do
3  |  distance[c] ← ∞;
4  end
5  distance[s] ← 0;
6  ENQUEUE(s);
7  while queue is not empty do
8  |  c ← DEQUEUE();
9  |  foreach neighbor b of c which is not an obstacle do
10 |  |  if distance[b] = ∞ then
11 |  |  |  distance[b] ← distance[c] + 1;
12 |  |  |  ENQUEUE(b);
13 |  |  end
14 |  end
15 end
```

Figure 4: Implementation of Queues