himanshujaju's blog

0-1 BFS [Tutorial]

By **himanshujaju**, history, 2 years ago,

Link To PDF version (Latex Formatted)

Topic : 0-1 BFS

Pre Requisites : Basics of Graph Theory , BFS , Shortest Path

Problem :

You have a **graph G** with **V vertices** and **E edges**. The graph is a weighted graph but the weights have a contraint that they can only be 0 or 1. Write an efficient code to calculate shortest path from a given source.

Solution :

Naive Solution — Dijkstra's Algorithm.

This has a complexity of O(E + VlogV) in its best implementation. You might try heuristics , but the worst case remains the same. At this point you maybe thinking about how you could optimise Dijkstra or why do I write such an efficient algorithm as the naive solution? Ok , so firstly the efficient solution isn't an optimisation of Dijkstra. Secondly , this is provided as the naive solution because almost everyone would code this up the first time they see such a question , assuming they know Dijkstra's algorithm.

Supposing Dijkstra's algorithm is your best code forward , I would like to present to you a very simple yet elegant trick to solve a question on this type of graph using Breadth First Search (BFS).

Before we dive into the algorithm, a lemma is required to get things crystal clear later on.

Lemma : "During the execution of BFS, the queue holding the vertices only contains elements from at max two successive levels of the BFS tree."

Explanation : The BFS tree is the tree built during the execution of BFS on any graph. This lemma is true since at every point in the execution of BFS , we only traverse to the adjacent vertices of a vertex and thus every vertex in the queue is at max one level away from all other vertices in the queue.

So let's get started with 0-1 BFS.

0-1 BFS :

This is so named , since it works on graphs with edge weights 0 and 1. Let's take a point of execution of BFS when you are at an arbitrary vertex "u" having edges with weight 0 and 1. Similar to Dijkstra , we only put a vertex in the queue if it has been relaxed by a previous vertex (distance is reduced by travelling on this edge) and we also keep the queue sorted by distance from source at every point of time.

Now , when we are at "u" , we know one thing for sure : Travelling an edge (u,v) would make sure that v is either in the same level as u or at the next successive level. This is because the edge weights are 0 and 1. An edge weight of 0 would mean that they lie on the same level , whereas an edge weight of 1 means they lie on the level below. We also know that during BFS our queue holds vertices of two successive levels at max. So, when we are at vertex "u" , our queue contains elements of level L[u] or L[u] + 1. And we also know that for an edge (u,v)

, L[v] is either L[u] or L[u] + 1. Thus , if the vertex "v" is relaxed and has the same level , we can push it to the front of our queue and if it has the very next level , we can push it to the end of the queue. This helps us keep the queue sorted by level for the BFS to work properly.

But, using a normal queue data structure , we cannot insert and keep it sorted in O(1). Using priority queue cost us O(logN) to keep it sorted. The problem with the normal queue is the absence of methods which helps us to perform all of these functions :

1. Remove Top Element (To get vertex for BFS)
2. Insert At the beginning (To push a vertex with same level)
3. Insert At the end (To push a vertex on next level)

Fortunately, all of these operations are supported by a double ended queue (or deque in C++ STL). Let's have a look at pseudocode for this trick :

```
for all v in vertices:
        dist[v] = inf
dist[source] = 0;
deque d
d.push_front(source)
while d.empty() == false:
        vertex = get front element and pop as in BFS.
        for all edges e of form (vertex , u):
                if travelling e relaxes distance to u:
                        relax dist[u]
                        if e.weight = 1:
                                d.push_back(u)
                        else:
                                d.push_front(u)
```

As you can see , this is quite similar to BFS + Dijkstra. But the time complexity of this code is O(E + V) , which is linear and more efficient than Dijkstra. The analysis and proof of correctness is also same as that of BFS.

Before moving into solving problems from online judges , try these exercises to make sure you completely understand why and how 0-1 BFS works :

1. Can we apply the same trick if our edge weights can only be 0 and x (x >= 0) ?
2. Can we apply the same trick if our edge weights are x and x+1 (x >= 0) ?
3. Can we apply the same trick if our edge weights are x and y (x,y >= 0) ?

This trick is actually quite a simple trick, but not many people know this. Here are some problems you can try this hack at :

1. http://www.spoj.com/problems/KATHTHI/ — My implementation
2. https://community.topcoder.com/stat?c=problem_statement&pm=10337
3. Problem J of Gym

Div1 — 500 on topcoder are tough to crack. So congrats on being able to solve one of them using such a simple trick :). I will add more problems as I find.

Happy Coding!

P.S. : My first attempt at a tutorial. Please suggest edits wherever required!

bfs, 0-1 bfs, shortest path, thistagwillhelpmefindyou

+178 himanshujaju 2 years ago 35

Nickolas → Codeforces Round #455 (Div. 2)

rng_58 → AtCoder World Tour Finals 2018: A new onsite contest with free flights/hotels, no age limit!

Mohtasim → ACM ICPC AMRITAPURI REGIONAL 2017

fushar → Asia-Pacific Informatics Olympiad (APIO) 2015 (online mirror available!)

PrinceOfPersia → Algorithm Gym :: Data structures

rajiv.vaidyanathan4 → HELP IN QUESTION: THE DELIVERY MAN

minimario → Problems that are Dijkstra with a Twist

gritukan → Codeforces Round 454 (and Technocup 2018 — Elimination Round 4)

kingofnumbers → OILCOMP spoj.

kadoban → Codeforces's Haskell setup is bad (with respect to "Safe Haskell")

ShowStopper728 → Graph Contest

Petr → A test-driven week

madhur4127 → SIGSEGV on Segment Tree for large sized array

Petr → A F5 week

Petr → An empty week

komendart → Codeforces Round #353 (Div. 2) Editorial

fcspartakm → Codeforces: Copying of Contests and Gyms

Al.Cash → Efficient and easy segment trees

veschii_nevstrui → Editorial Codeforces Round 454 (and Technocup 2018 — Elimination Round 4)

raj454raj → StopStalk: Tool to maintain your algorithmic progress

shmundhra → Runtime Error - 1073741819

violet_coder → Number of Common Subsequences of K String

yogahmad27 → References of Number Theory

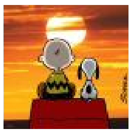
removed1 → Codeforces Beta Round #1 - Tutorial

fcspartakm → Codeforces Round #452 (Div.2) Editorial

Detailed →

Comments (35)

Write comment?



2 years ago, # | 0

*Auto comment: topic has been updated by **himanshujaju** (previous revision, new revision, compare).*

→ Reply

himanshujaju

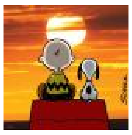
kshj78

2 years ago, # | ← Rev. 2 +6

Hi everyone!

You may solve problem J from here.

→ Reply


himanshujaju

2 years ago, # ^ | 0

Thanks , added to the list.

→ Reply

mogers

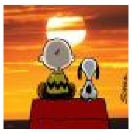
2 years ago, # | +3

Nice tutorial! Just a comment:

This has a complexity of O(ElogV) in its best implementation. You might try heuristics , but the worst case remains the same

The worst case time complexity of Dijkstra's algorithm is O(E + V log V), using Fibonacci heaps. In contest problems, this can actually perform worse than the typical O(E log V) implementation, but the theoretical complexity is still O(E + V log V).

→ Reply


himanshujaju

2 years ago, # ^ | ← Rev. 2 0

Thanks. I didnt actually know about the Fibonacci Heap implementation, will read up on that. I wrote it as O(ElogV) considering that everyone implements the priority_queue method in contests. Edited the post accordingly.

→ Reply

m.boniecki

2 years ago, # | 0

Really nice tutorial! I am waiting for more :)

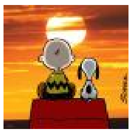
→ Reply

kingofnumbers

2 years ago, # | +19

for the first question it's clearly yes, but for second and third ones I think the answer is no, am I right?

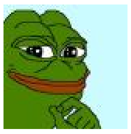
→ Reply


himanshujaju

2 years ago, # ^ | 0

Yes.

→ Reply


additya1998

2 years ago, # ^ | 0

Can you please explain why is it NO for the second question? Also, if it is NO, should it be x>0?

→ Reply

himanshujaju

2 years ago, # ^ | ← Rev. 3 0

Firstly, x >= 0 is given to specify the general case when x is non negative (Similar to dijkstra constraints).

You might think subtracting x will reduce it to a 0-1 BFS problem , but let's suppose the graph is like :

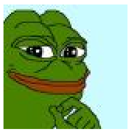
1 -> 2 (weight x)

2 -> 3 (weight x)

1 -> 3 (weight x + 1)

For source = 1 and destination = 3 :

0-1 BFS gives us minimum distance by traversing the two 0 edges , while we should traverse the 1 edge to get minimum distance. [0 edges means edges with weight x , 1 edges means edges with weight x + 1]
→ Reply


additya1998

2 years ago, # ^ | ☆ 0

Okay, got it. Thanks.
→ Reply

fofao_funk

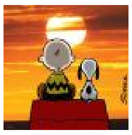
2 years ago, # | ☆ 0

I didn't get this part, specially the bolded one:

"Now , when we are at "u" , we know one thing for sure : Travelling an edge (u,v) would make sure that v is either in the same level as u or at the next successive level. **This is because the edge weights are 0 and 1. An edge weight of 0 would mean that they lie on the same level , whereas an edge weight of 1 means they lie on the level below.**"

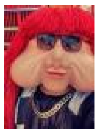
Could you elaborate a bit more?

Thanks.
→ Reply


himanshujaju

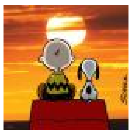
2 years ago, # ^ | ☆ 0

Consider the level here as distance from the root. For an edge weight 0 , the distance remains the same and for an edge weight 1, the distance increases by one.
→ Reply


fofao_funk

2 years ago, # ^ | ☆ 0

Got it! Thanks.
→ Reply


himanshujaju

2 years ago, # | ☆ ← Rev. 2 0

*Added Sample Implementation. Auto comment: topic has been updated by **himanshujaju** (previous revision, new revision, compare).*
→ Reply

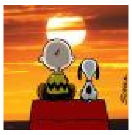
YouCantHandleMe

2 years ago, # | ☆ ← Rev. 5 +3

A similar but more general technique that can correctly deal with questions 2 and 3 is as follows.

Maintain a minimum priority queue of tuples with type (distance from source, queue of vertices having this distance). Proceed with a normal BFS, however, only pop from the queue with minimum distance until it is exhausted, then move to the next smallest. This is O(V+E) given a limited number of weights. In fact, I believe in the worst case its time complexity is bounded by O(V + E * lg(#distinct_edge_weights)). [nonsense]There should only be W(W+1)/2 elements in the priority queue at any given point in time, where W is the number of distinct edge weights.[/nonsense] Notice that this can have significantly larger constant factors as compared to running Dijkstra's algorithm, but for a small set of distinct edge weights, is should have low constant factors and superior asymptotic performance.

There's a good chance I have made a mistake as I derived this on the spot while typing. Please be nice.
→ Reply


himanshujaju

2 years ago, # ^ | ☆ 0

Is the priority queue tuple of the form pair<distance , queue > ? Could you explain why the log factor has only number of distinct edge weights?

→ Reply

YouCantHandleMe

2 years ago, # ^ | ☆ ← Rev. 3 0

Yes.

I've thought about it more (my first post was very hasty), and now I tentatively claim that you need only #distinct_edge_weights queues, and the next vertex we visit is the minimum element from the head of any queue. We insert elements onto the end of the queue corresponding to the edge weight we just traversed to get there.

→ Reply

himanshujaju

2 years ago, # ^ | ☆ ← Rev. 2 0

Suppose edge weights are 3 and 5. Initially we have two queues for distance 3 and 5. Now by iterating vertices at distance 3 , we can get two more queues for distance 6 and 8. Now, we have queues for distance (5,6,8). After iterating or 5, we have (6,8,10) and after iterating for 6, we have (8,9,10,11) which is greater than W*(W+1)/2. Or did I understand your claim incorrectly? Also, is sorting the priority queue O(logN) when we have a queue in the tuple or it increases?

→ Reply

YouCantHandleMe

2 years ago, # ^ | ☆ ← Rev. 2 0

I think you've misunderstood my response. My first post is 50% gibberish, please ignore it for the purposes of understand what I am talking about haha.

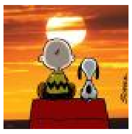
In that case there will only be two queues. One for weight 3, and one for weight 5. As I have said, the algorithm works like this. Create a queue for each distinct weight. Queue elements are a tuple of (distance traveled, current vertex). Now we run a modified BFS. While there is an element on any queue, find the queue with the minimum head element, pop it, then process it. When we wish to push a child to one of the queues, we will have just traversed an edge to get to the child we are pushing. Push the child to the queue corresponding to the weight of the edge we traversed.

I think this can be shown to always work. The reason I think it works is that all the queues will be in monotonically increasing order of distance. To prove this, let's postulate that, for some reason, they are not in increasing order. For this to be the case, there must be an element E of a queue such that its direct predecessor P in the same queue was greater, that is E < P. Let's assume that this is the first such inversion that our algorithm has produced for the sake of simplicity. Note that P and E are on the same queue, so they must both have ended with the same edge weight. Now, consider that when we popped off the prefix to E (the sub-path minus the last edge), it was on the front of some queue.

The same must have been true for the

The same must have been true for the prefix of P. They may have been on different queues. Now observe that prefix(E) < prefix(P). If both were on the front of queues, then we couldn't possibly have picked prefix(P) first, as we always pick the minimum element. So we must have picked some element that was earlier in the same queue as prefix(E). However, since we said that this is the first inversion that happened, all such queue elements must be <= prefix(E), so we would have had to have made the choice between prefix(P) and prefix(E) at some point. This leads to a contradiction: we couldn't possibly have picked prefix(P) before prefix(E), thus our queues are always in increasing order. (It's very likely I've made a mistake as I did in my first post, so definitely consider this and come to your own conclusions.)

→ Reply


himanshujaju

2 years ago, # ← Rev. 2 0

Oh yes, this thing works. Some people implement 0-1 BFS in this way!

→ Reply

retrograd

2 years ago, # | ☆ ← Rev. 2 +3

I would like to propose an alternative algorithm to what you said, that seems easier to me:

We keep a regular queue (just like regular bfs), and also two graphs, `G0` and `G1` (for edges of cost `0`, and `1`, respectively). Then the algorithm is as follows:

- Step 1: Add the source vertex to the graph. Mark it as visited
- Step 2: Take the front of the queue and do a DFS through `G0`, marking any unvisited nodes and updating the corresponding distance. Add the unvisited adjacent nodes (through `G1` this time) to the back of the queue
- Step 3: Repeat Step 2 while the queue is not empty.

The C++ code is as follows:

```
void dfs(int node) {
   //Traverse through the 0-connected component
   for(auto v : G0[node])
      if(!Seen[v]) {
         Seen[v] = 1;
         D[v] = D[node];
         dfs(v);
      }
   //Push the unvisited 1-cost neighbours
   for(auto v : G1[node])
      if(!Seen[v]) {
         Seen[v] = 1;
         D[v] = D[node] + 1;
         Q.push(v);
      }
}

void bfs1_0(int start) {
   Q.push(start);
   Seen[start] = 1;

   while(!Q.empty()) {
      int top = Q.front();
      Q.pop();

      dfs(top);
```

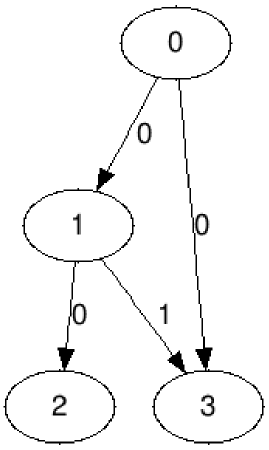
```
        dfs(top);
    }
}
```

You can easily prove that the queue Q will always be sorted by the distances, so, in essence, the algorithm cannot produce a different output than Dijkstra's.
→ Reply

Branimir

13 months ago, # ^ | ☆ ← Rev. 2 0

For the next graph:



Calling bfs1_0(0), your algorithm produce distance 1 for node 3, but it should be 0.

(Sorry for my poor(and horrible) english. xD)
→ Reply

bciobanu

2 years ago, # | ☆ ← Rev. 3 0

Another way to speed up the shortest path from a given source on a graph with small weights (not necessary to be 0 and 1) would be Dial's Algorithm (also known as Dijkstra with buckets), which runs in O(V * C + E) time and memory (considering the notations made by op and C the maximum weight of an edge).

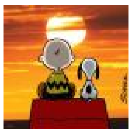
Here you have a presentation of the algorithm.

LE: I belive this is what YouCantHandleMe was talking about.
→ Reply

Valkata.a.k.a.TheHacker

2 years ago, # | ☆ +3

I think it can also be solved with FloydWarshall or BellmanFord.
→ Reply


himanshujaju

2 years ago, # ^ | ☆ 0

You can even solve with a recursive brute force!
→ Reply

Valkata.a.k.a.TheHacker

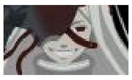
2 years ago, # ^ | ☆ 0

But it is too slow. The algorithms I wrote are known good algorithms.
→ Reply

himanshujaju

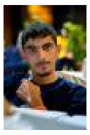
2 years ago, # ^ | ☆ 0

So dont you think this technique is faster than what you wrote? Sorry if I am being arrogant and idiotic but its all part of the handle :D
→ Reply


tieuchanlong

17 months ago, # | ☆ 0

Can anyone offer some more problems.I would be very appreciate.
→ Reply

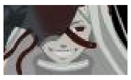

justHusam

17 months ago, # ^ | ☆ 0

11573 — Ocean Currents

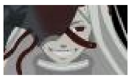
28. Error

B. Chamber of Secrets
→ Reply


tieuchanlong

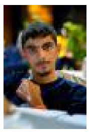
17 months ago, # ^ | ☆ 0

Thanks for your support.
→ Reply


tieuchanlong

8 months ago, # ^ | ☆ 0

I also found interesting one DIGJUMP codechef
→ Reply


justHusam

16 months ago, # ^ | ☆ 0

Another problem is 590C - Three States
→ Reply

Matjaz

5 months ago, # | ☆ 0

It should be pointed out that the algorithm described will sometimes push the same node on the queue twice which makes it somewhat unlike the usual BFS.
→ Reply

mahim007

new, 7 weeks ago, # ^ | ☆ 0

can you provide some details explanation about what u said and some problem related to it? thank u
→ Reply