Parallel Pointers: Graphs

Breadth-first Search (BFS) 101

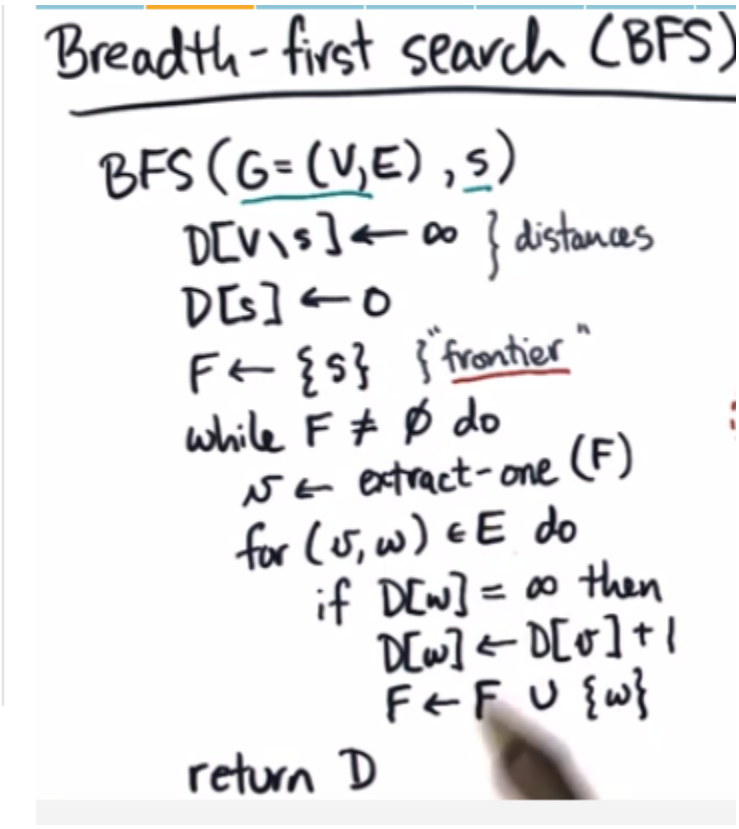
Given a vertex S, what is the distance of all the other vertices from S?

Why would this be useful? To find the fastest route through a network.

To do a BFS - keep a distance for each vertex.

1. S is the first vertex - it is 0 distance from itself. Set all the other vertices to infinity.
2. Begin at S. Look at its immediate neighbors. Their distance for S is 1.
3. Now move to these immediate neighbors and mark the distance between these neighbors as one.
4. Do this for each vertex in the graph.

Each successive advancement vertices being examined is called a Frontier.



Start with a graph G.
Set all the distances to infinity
Set the source distance to 0

Determine the frontier 'F' of S

Look at each vertex in the frontier
Examine each neighbor 'w' of each vertex

If the distance of that neighbor is infinity, it has not been visited.

If it has not been visited - the neighbors distance is the vertex's distance + 1

Do this for all vertices in the search.

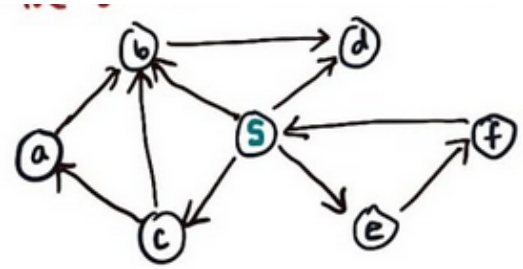
What does this algorithm cost? By cost we mean running time.

1. The frontier only grows when an unvisited vertices is inserted. Which means the frontier can only be as large as the number of vertices.
2. Each edge is visited at most once, if the graph is directed.

3. If the graph is undirected, each edge will be visited at most twice.

This means the runtime is linear and the size of the graph

BFS Quiz → Note this is a directed graph.

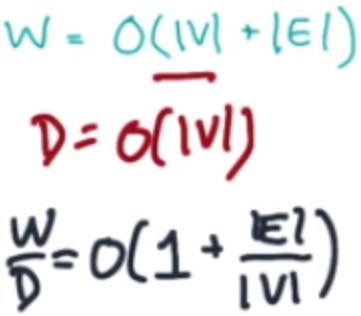


When looking for neighbors of 'S' - look only at those vertices that go INTO 'S', not out of it. For example, 'd' is a neighbor of 'S' and 'f' is NOT.

Analysis - Is BFS Inherently Sequential?

Can this algorithm be parallelized?

There is a bottleneck in the relationship between extracting v, inserting w.



The available parallelism is W/D → which is dependent upon |E|/|V|.

Is this a good thing or not?

In real life, graphs are sparse. So |E| = O(|V|). This means the average available parallelism will be a constant and that is bad.

This leads to the conclusion that the sequential algorithm is bad.

Intuition - Why we might do better

The BFS visits the graph in waves. There are two important implications from this fact:

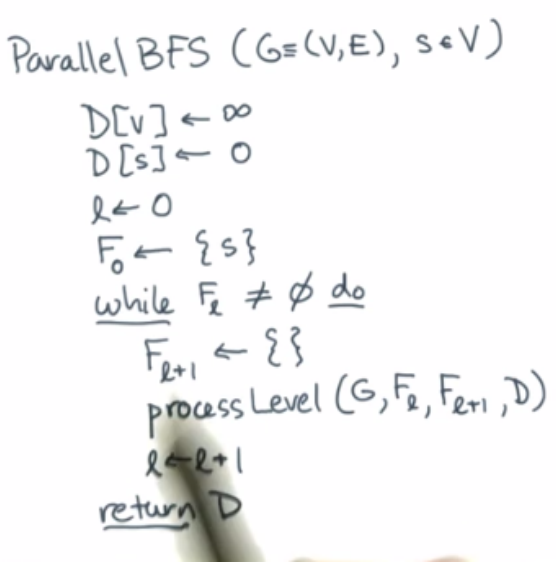
1. The upper bound on the span should be the number of waves, not the number of vertices. These waves are called levels. A level is all of the vertices equidistant to the source.
   The diameter of a graph is the maximum shortest distance between any pair of vertices. Diameter is a property of the whole graph, and is an upper bound on the number of levels of any starting vertex.
   Level-synchronous transversal is visiting the nodes level by level.

2. If you perform a level-synchronous BFS, then the order in which you visit each vertex of a given frontier is immaterial. This means we can visit them at the same time.

So this is an opportunity for parallelism - all the vertices of a frontier can be visited at the same time.

High Level Approach to Parallel BFS

1. Carry out the BFS level by level (not vertex by vertex)
2. Process the entire level in parallel - we can do this because the order in which the vertices are visited should not matter.



This is very similar to the serial version, with a few exceptions.

l ← 0 ….. the code is level synchronous.
This is the level counter set to 0

The frontiers referenced are also level specific (F_l)

l ← l + 1 …. increments the counter

The span (defined by the while loop) will be no larger than the diameter of the graph

Process level will: take the graph and the current frontier. It will produce a new frontier and update the distances (D)

Bag: Key Properties

The data structure to be used with the parallel BFS is a bag.
A bag has the following data properties:

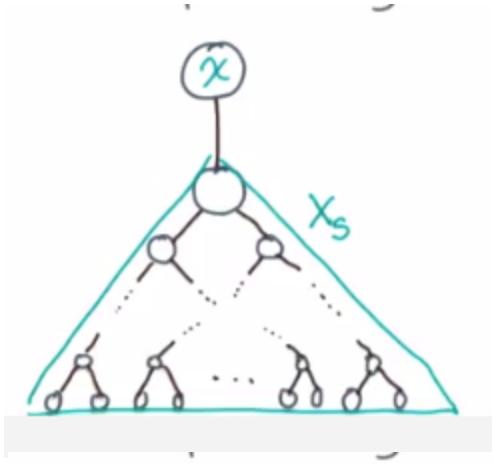
The data is an unordered collection
It will allow repetition
Allow redundant insertions of the same vertex, if necessary

A bag has the following operational properties:

Fast traversal
Fast and logically associative “union” and “split”

Logically associative means if A U B == B U A
This will also give the ability to apply Reducer Hyper Objects.

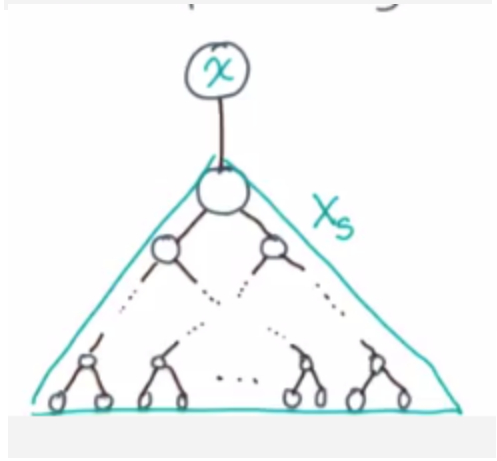
Pennant, Building Blocks for Bags

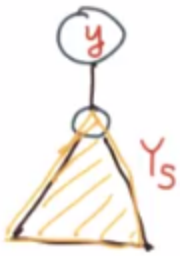


Pennant is: a tree with 2^k nodes and a unary root having a child. the child is the root of a complete binary tree.

X is the root
X_s is the complete binary tree

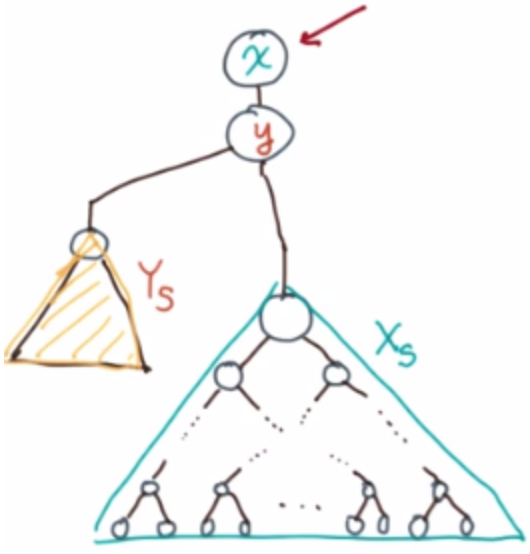
If there are two pennants, and they are EXACTLY THE SAME SIZE (|X| == |Y|)





Then the two pennants can be combined.

1. Choose one of the roots to be the root of the combined pennant.
2. The two pennants are now children of the root.



The combination is also a pennant.
There will now be 2^{k+1} nodes.

This was fairly easy, it involved rearranging pointers.

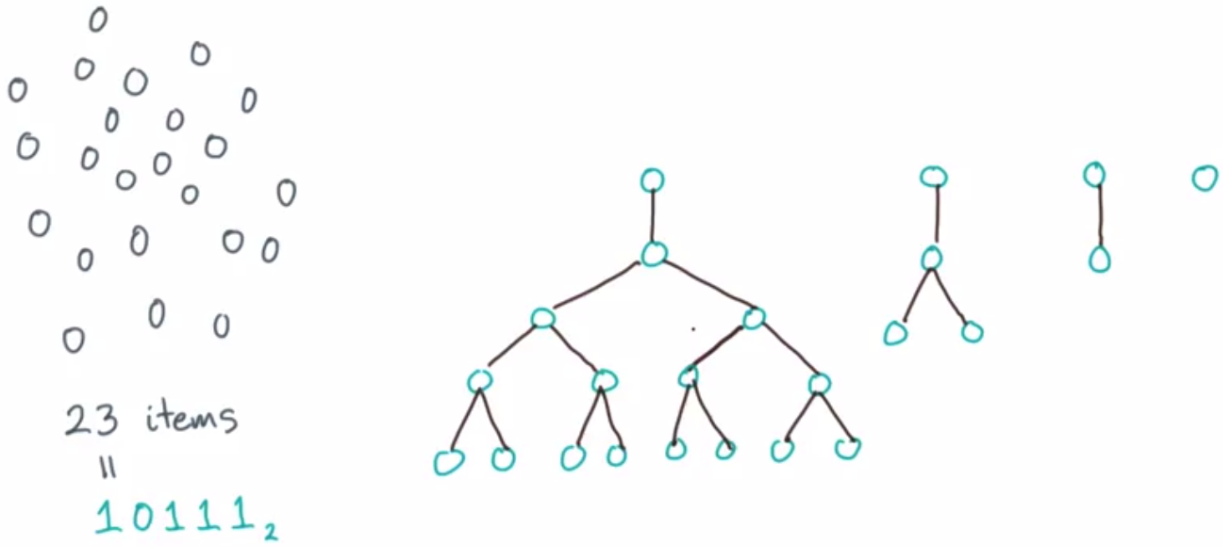
The reverse, splitting a pennant into two pennants can be done using the reverse of the given method.

Combining Pennants into Bags

Pennants are good for representing collections that have a size 2^N, but there needs to be something for other size collections.

A bag needs to contain any size collection. How can pennants be used to form bags?

An example of pennants used to put a collection in a bag.
23 items. Divided into groups of powers of 2. (23 = 10111)



To connect these groups, use an array of pointers, use the null pointer for any empty bits. This array is called a spine.

To insert a new element into the collection.

1. Try to insert it into the LSB
2. If the LSB is occupied, combine the LSB with the new element and carry it.
3. Move to the next LSB, repeat the process.

What is the Cost of Insertion?

Remember … when inserting an element, you may need to traverse the entire spine.
An integer of size N needs N bits to store its power.
Inserting an element is simply a matter of shuffling pointers.

It takes the same asymptotic time to insert N elements into a bag as it does to insert 1.
The amortized time to insert elements into a bag is constant.

Bag Splitting

A right bit shift is the same a division by two.

To split a bag - an empty bag is also necessary.
Split the pennant in half.
Store 1 of the halves in the spare bag and one in the next lower bit on the bag.

The cost of splitting a bag …..

$$\text{Cost} = [O(\log n) \text{ splits}] \times [O(1) \text{ per split}] = O(\log n)$$

Finishing the Parallel BFS with Bags

```
processLevel (G, F_ℓ, F_{ℓ+1}, D)
  if |F_ℓ| > φ_0 then
      (A, B) ← bagSplit (F_ℓ)
      spawn processLevel (G, A,
                          F_{ℓ+1}, D)
      processLevel (G, B, F_{ℓ+1}, D)
      sync
  else
      for v ∈ F_ℓ do
          par-for (v, w) ∈ E do
              if D[w] = ∞ then
                  D[w] ← ℓ+1
                  bagInsert (F_{ℓ+1}, w)
```

If the bag is big enough -- use divide and conquer otherwise use the sequential method.

The difference with this sequential method is the for loop - this uses a parallel for loop.

There is a data race here as each task tries to update the neighbors. This is perfectly safe here.

The costs are tricky for this experiment.
Span is affected by the number of levels.
Span of process level - has 3 parts: depth, cost of splitting, cost of the base case.