How to prove that BFS(s) actually finds shortest paths?

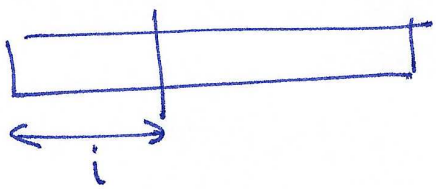
Discover loop invariant.

Statement parametrized by the number of iterations (of a loop), is typically trivially true at the beginning of the first loop, and is proven by induction over the number of iterations.

~~Statement after the last~~

Insertion sort: Input array A of length n

```
for i = 0 to n-1
{
    j = i
    while (j > 0 and A[j-1] > A[j])
    {
        swap A[j-1] and A[j]
        j--
    }
}
```



Loop invariant: $\forall\ 0 \le i \le n-1$:

After i iterations of the outer loop.

A[0..i] is in sorted order.

For $i=0$, $A[0]$ is in sorted order.

Base Case: Trivially true. Base case of induction.

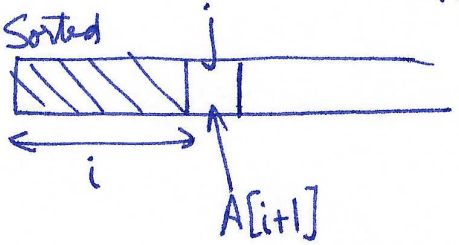
Induction: Assume statement is true after i iterations ($i \geq 0$).

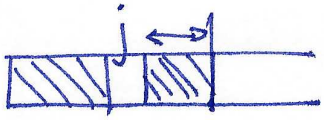
Let us prove statement after $(i+1)$ iterations. → Hypothesis

In the $(i+1)^{th}$ iteration,

j is initialized to $(i+1)$.

$A[j]$ is $A[i+1]$.





Throughout the running of the inner loop, $A[j]$ is always $A[i+1]$. We need to argue: when inner loop terminates $A[0..i+1]$ is in sorted order.

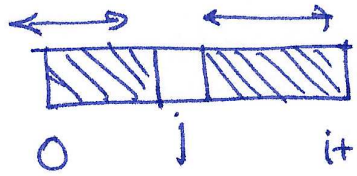
↳ When it terminates, $A[j-1] \leq A[j]$ or $j=0$.

Clm: Consider the $(i+1)^{th}$ outer loop. At all times, $A[j]$ is strictly less than $A[j']$ $\forall j < j' \leq i+1$.

Proof: Exercise

Case 1: $j=0$. By above claim $A[j]=A[0]$ is strictly less than $A[1], A[2] - - , A[i+1]$. The order among $A[1] -- A[i+1]$ is NOT changed by the swap, because the swap always involves $A[j]$. Therefore $A[0..i+1]$ is in sorted order.

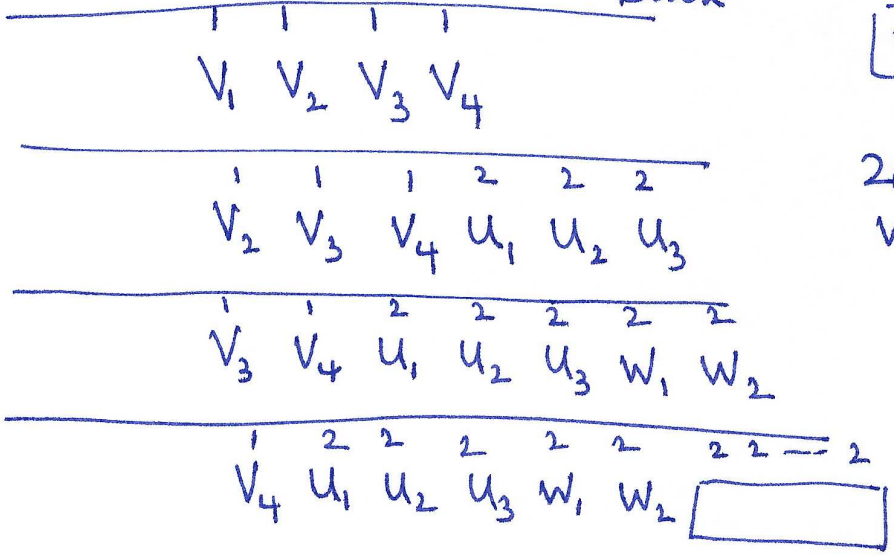
Case 2: $j \neq 0$, so for some $0 < j \leq i+1$ $A[j-1] \leq A[j]$.

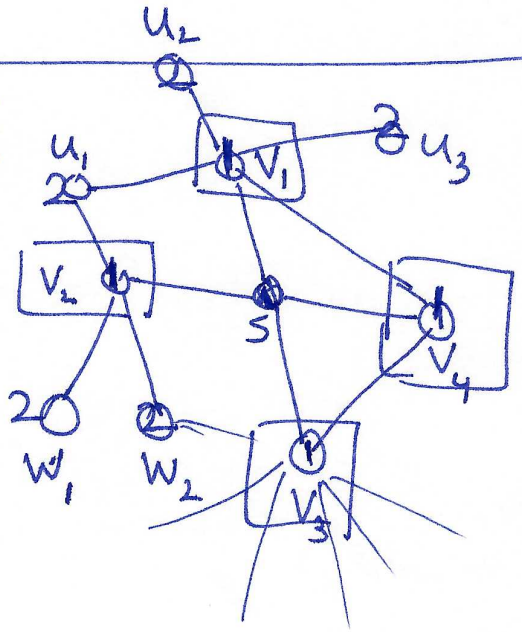


The swaps do not change $A[0..j-1]$, which by induction hypothesis, is already in sorted order. The swaps do not the order in $A[j+1..i+1]$, which is also already in sorted order (induction hypothesis). By claim above $A[j] < A[j+1]$. By case condition, $A[j-1] \leq A[j]$. Thus, $A[0..i+1]$ is in sorted order.

Loop invariant of BFS

Front ... Back



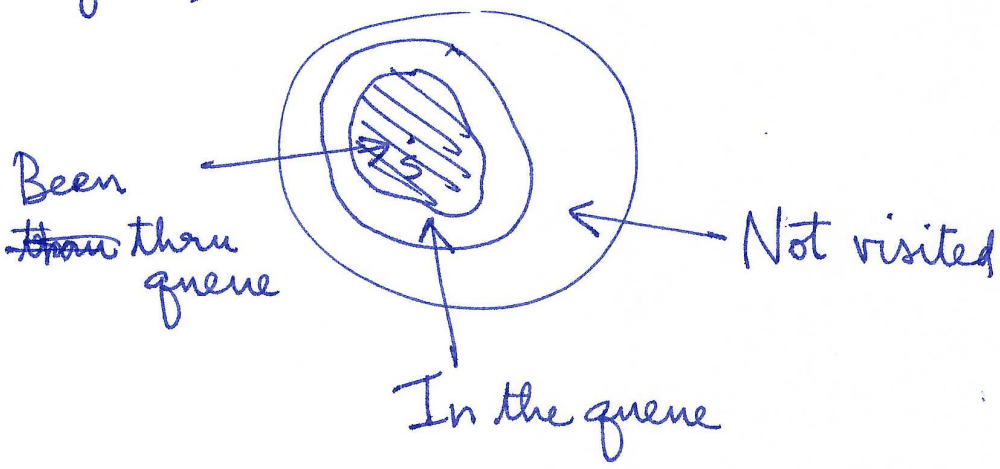


Thm: Loop invariant of BFS(s) After i (i vertices dequeued) runs of while loop in BFS(s), there exists a distance $l \in \mathbb{N}$ such that

(1) The queue contains vertices at distance l from s followed by (potentially zero) vertices at distance $(l+1)$.

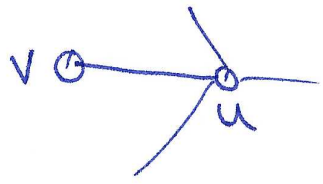
(2) All vertices at distance $\leq l$ from s that are NOT in the queue have already been dequeued (passed through queue).

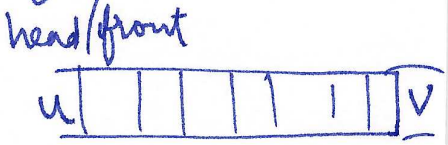
(3) No vertex at distance $> l$ from s has been dequeued (up to this point).



Let us now argue that BFS(s) returns ~~the~~ a shortest path tree.

Consider vertex v. When does v get into queue?





There is a neighbor u that is the head of the queue. Suppose u has distance l.

v cannot have distance $\leq l$. (By loop invariant)

Therefore, v has distance $> l$. v's neighbor u has distance l, hence v's distance is $(l+1)$. BFS(s) correctly determines a shortest path (technically induction).

BFS(s) BFS(G)

(1) $\forall$ v: set visited[v] to false, pred[v] to NULL.

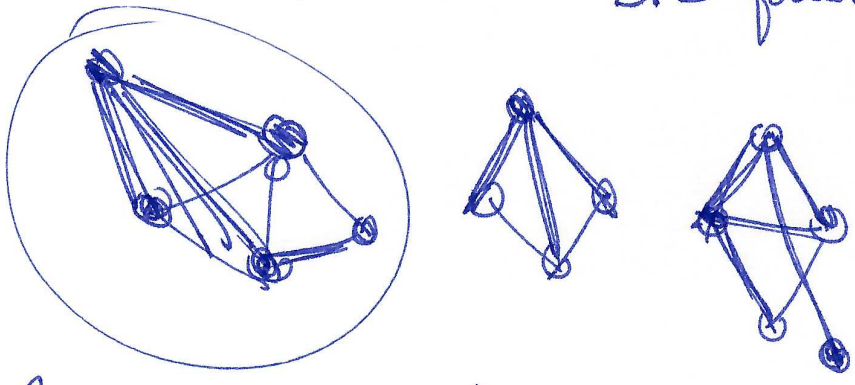
(2) for all s:

(i) If visited[s] is false, run BFS(s).

The output: ALL vertices get visited.

pred stores a collection of trees. Roots are vertices where pred is NULL

Forest ← BFS forest, each tree is a connected component.



Connected component

For simple undirected graph G, V can be partitioned into "maximal connected components"

$S \subseteq V$

S is a connected component if

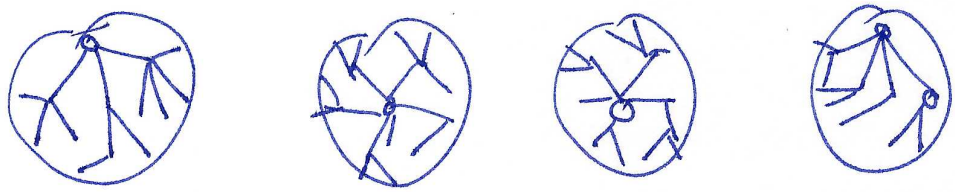
$\forall\ u,v \in S$ u is connected to v

S is maximal if $\forall\ u \in S, w \in V$: if u is connected to w, $w \in S$.

Running time of BFS(G) n vertices m edges

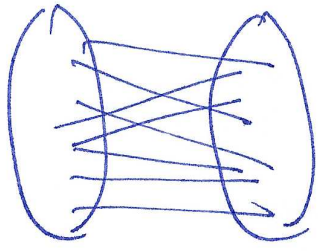
Every adjacency list is traversed exactly once (when vertex is dequeued.)

$\Theta(m+n)$ Linear time!



visited visited visited visited

Graph G is bipartite (two-colorable)
if V can be partitioned into $V_1 \cup V_2$
such that all edges go between V_1 and V_2



give an algorithm

Q. Given a graph G, determine if G is bipartite.

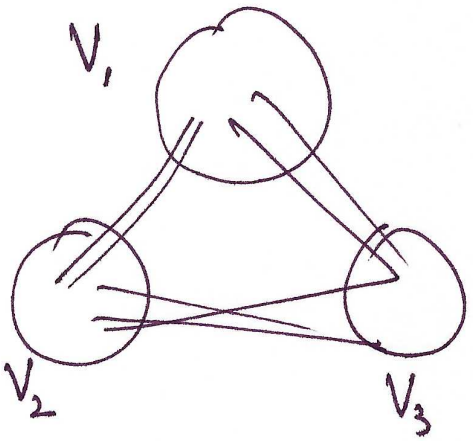
Brute force algorithm: Try all partitions

For all partition of $V = V_1 \cup V_2$, check if edges go between V_1 and V_2.

$\Theta((m+n)2^n)$ (2^n partitions)

BFS gives a $\Theta(m+n)$ algorithm for this problem.

G is tripartite (3-colorable) if V can be partitioned into $V_1 \cup V_2 \cup V_3$ s.t. all edges go between different sets of partition



Brute force:

$O(3^n)$ algorithm

Is there an algorithm for checking if graph is Tripartite whose running time is $O((m+n)^k)$ (k constant)

P vs NP

We believe NO.