CSE 100: GRAPH ALGORITHMS
Dijkstra’s Algorithm: Questions

- Initialize the graph: Give all vertices a dist of INFINITY, set all “done” flags to false
- Start at s; give s dist = 0 and set prev field to -1
- Enqueue (s, 0) into a priority queue. This queue contain pairs (v, cost) where cost is the best cost path found so far from s to v. It will be ordered by cost, with smallest cost at the head.
- While the priority queue is not empty:
  - Dequeue the pair (v, c) from the head of the queue.
  - If v’s “done” is true, continue
  - Else set v’s “done” to true. We have found the shortest path to v. (It’s prev and dist field are already correct).
  - For each of v’s adjacent nodes, w:
    - Calculate the best path cost, c, to w via v by adding the edge cost for (v, w) to v’s “dist”.
    - If c is less than w’s “dist”, replace w’s “dist” with c, replace its prev by v and enqueue (w, c)

When a node comes out of the priority queue, how do you know you’ve found the shortest path to the node?
Spanning trees

- We will consider spanning trees for undirected graphs
- A spanning tree of an undirected graph \( G \) is an undirected graph that...
  - contains all the vertices of \( G \)
  - contains only edges of \( G \)
  - has no cycles
  - is connected

- A spanning tree is called “spanning” because it connects all the graph’s vertices

- A spanning tree is called a “tree” because it has no cycles (recall the definition of *cycle* for undirected graphs)

- What is the root of the spanning tree?
  - you could pick any vertex as the root; the vertices adjacent to that one are then the children of the root; etc.
Spanning trees: examples

- Consider this undirected graph $G$: 

```
  V0 --- V1
   \   /   \
    V2 --- V3
         |     |
         V5   V6
   \     /   \
    V4   V1
```
Spanning tree? Ex. 1

Is this graph a spanning tree of G?

A. Yes
B. No
Is this graph a spanning tree of $G$?

A. Yes
B. No
Spanning tree? Ex. 3

Is this graph a spanning tree of G?

A. Yes
B. No
Minimum Spanning tree: Spanning tree with minimum total cost

Is this graph a minimum spanning tree of G?

A. Yes  
B. No
Minimum spanning trees in a weighted graph

- A single graph can have many different spanning trees.

- They all must have the same number of edges, but if it is a weighted graph, they may differ in the total weight of their edges.

- Of all spanning trees in a weighted graph, one with the least total weight is a minimum spanning tree (MST).

- It can be useful to find a minimum spanning tree for a graph: this is the least-cost version of the graph that is still connected, i.e. that has a path between every pair of vertices.

- How to do it?
Prim’s MST Algorithm

- Start with any vertex and grow like a mold, one edge at a time
- Each iteration choose the cheapest crossing edge
Fast implementation of Prim’s MST Algorithm

- Iteration 1
Finding a minimum spanning tree: Prim’s algorithm

• As we know, minimum weight paths from a start vertex can be found using Djikstra’s algorithm

• At each stage, Djikstra’s algorithm extends the best path from the start vertex (priority queue ordered by total path cost) by adding edges to it

• To build a minimum spanning tree, you can modify Djikstra’s algorithm slightly to get Prim’s algorithm

• At each stage, Prim’s algorithm adds the edge that has the least cost from any vertex in the spanning tree being built so far (priority queue ordered by single edge cost)

• Like Djikstra’s algorithm, Prim’s algorithm has worst-case time cost $O(|E| \log |V|)$

• We will look at another algorithm: Kruskal’s algorithm, which also is a simple greedy algorithm

• Kruskal’s has the same big-O worst case time cost as Prim’s, but in practice it can be made to run faster than Prim’s, if efficient supporting data structures are used
Fast Implementation of Prim’s MST Algorithm

1. Initialize the vertex vector for the graph. Set all “done” fields to false. Pick an arbitrary start vertex \( s \). Set its “prev” field to -1, and its “done” field to true. Iterate through the adjacency list of \( s \), and put those edges in the priority queue.

2. Is the priority queue empty? Done!

3. Remove from the priority queue the edge \((v, w, \text{cost})\) with the smallest cost.

4. Is the “done” field of the vertex \( w \) marked true?
   - If Yes: this edge connects two vertices already connected in the spanning tree, and we cannot use it. Go to 1.

5. Accept the edge: Mark the “done” field of vertex \( w \) true, and set the “prev” field of \( w \) to indicate \( v \).

6. Iterate through \( w \)’s adjacency list, putting each edge in the priority queue.

7. Go to 1.

- The resulting spanning tree is then implicit in the values of “prev” fields in the vertex objects.
**Weighted minimum spanning tree: Kruskal’s algorithm**

- Prim’s algorithm starts with a single vertex, and grows it by adding edges until the MST is built:

- Kruskal’s algorithm starts with a forest of single-node trees (one for each vertex in the graph) and joins them together by adding edges until the MST is built;

Why Kruskal?
Kruskal’s algorithm:
Kruskal’s algorithm: output

- Show the result here:

- What is the total cost of this spanning tree?

- Is there another spanning tree with lower cost? With equal cost?
Naïve Implementation of Kruskal’s algorithm:

1. Sort edges in increasing order of cost

2. Set of edges in MST, $T=\{\}$

3. For $i= 1$ to $|E|$
   
   If $T \cup \{e_i\}$ has no cycles
   
   Add $e_i$ to $T$

Ref: Tim Roughgarden (stanford)
Running Time of Naïve Implementation of Kruskal’s algorithm:

1. Sort edges in increasing order of cost

2. Set of edges in MST, $T=\{\}$

3. For $i=1$ to $|E|$
   
   If $T \cup \{e_i\}$ has no cycles
   
   Add $e_i$ to $T$

Ref: Tim Roughgarden (stanford)
Towards a fast implementation for Kruskal’s Algorithm

• What is the work that we are repeatedly doing with Kruskal’s Algo?
Towards a fast implementation for Kruskal’s Algorithm

- What is the work that we are repeatedly doing with Kruskal’s Algo?
  - Checking for cycles: Linear with BFS, DFS
  - Union-Find Data structure allows us to do this in almost constant time!
Graph algorithm time costs: a summary

- The graph algorithms we have studied have fast algorithms:
  - Find shortest path in unweighted graphs
    - Solved by basic breadth-first search: $O(|V|+|E|)$ worst case
  - Find shortest path in weighted graphs
    - Solved by Dijkstra’s algorithm: $O(|E| \log|V|)$ worst case
  - Find minimum-cost spanning tree in weighted graphs
    - Solved by Prim’s or Kruskal’s algorithm: $O(|E| \log|V|)$ worst case

- The “greedy” algorithms used for solving these problems have polynomial time cost functions in the worst case
  - since $|E| \leq |V|^2$, Dijkstra’s, Prim’s and Kruskal’s algorithms are $O(|V|^3)$

- As a result, these problems can be solved in a reasonable amount of time, even for large graphs; they are considered to be ‘tractable’ problems

- However, many graph problems do not have any known polynomial time solutions...!
Towards a fast implementation of Kruskal’s Algorithm

Q: Which of the following algorithms can be used to check if adding an edge \((v, w)\) to an existing Graph creates a cycle?

A. DFS
B. BFS
C. Either A or B
D. None of the above
BFS: Running Time

The basic idea is a breadth-first search of the graph, starting at source vertex \( s \)

- Initially, give all vertices in the graph a distance of \text{INFINITY}.
- Start at \( s \); give \( s \) distance = 0
- Enqueue \( s \) into a queue
- While the queue is not empty:
  - Dequeue the vertex \( v \) from the head of the queue
  - For each of \( v \)'s adjacent nodes that has not yet been visited:
    - Mark its distance as \( 1 + \) the distance to \( v \)
    - Enqueue it in the queue

What is the time complexity (in terms of \(|V|\) and \(|E|\)) of this algorithm?

A. \( O(|V|) \)
B. \( O(|V||E|) \)
C. \( O(|V|+|E|) \)
D. \( O(|V|^2) \)
E. Other
Running Time of Naïve Implementation of Kruskal’s algorithm using BFS for cycle checks:

1. Sort edges in increasing order of cost
2. Set of edges in MST, $T = \{ \}$
3. For $i = 1$ to $|E|$
   
   If $T \cup \{e_i\}$ has no cycles
   
   Add $e_i$ to $T$

Ref: Tim Roughgarden (stanford)
Towards a fast implementation for Kruskal’s Algorithm

• What is the work that we are repeatedly doing with Kruskal’s Algo?
Towards a fast implementation for Kruskal’s Algorithm

- What is the work that we are repeatedly doing in Kruskal’s Algo?
  - Checking for cycles: Linear with BFS, DFS
  - Union-Find Data structure allows us to do this in nearly constant time!
The Union-Find Data Structure

• Efficient way of maintaining partitions
• Supports only two operations
  • Union
  • Find
Equivalence Relations

- An equivalence relation $E(x,y)$ over a domain $S$ is a boolean function that satisfies these properties for every $x, y, z$ in $S$
  - $E(x,x)$ is true \textbf{(reflexivity)}
  - If $E(x,y)$ is true, then $E(y,x)$ is true \textbf{(symmetry)}
  - If $E(x,y)$ and $E(y,z)$ are true, then $E(x,z)$ is true \textbf{(transitivity)}

- Example 1:
  - $E(x,y)$: Are the integers $x$ and $y$ equal?
  - Then $E()$ is an equivalence relation over integers

- Example 2: Given vertices $x$ and $y$ in a Graph $G$
  - $E(x,y)$: Are $x$ and $y$ connected?
Equivalence Classes

- An equivalence relation $E()$ over a set $S$ defines a system of *equivalence classes* within $S$
- The equivalence class of some element $x \in S$ is that set of all $y \in S$ such that $E(x,y)$ is true
- Note that every equivalence class defined this way is a subset of $S$
- The equivalence classes are disjoint subsets: no element of $S$ is in two different equivalence classes
- The equivalence classes are exhaustive: every element of $S$ is in some equivalence class

Example 1:
- $E(x,y)$: Are the integers $x$ and $y$ equal?
- Then $E()$ is an equivalence relation over integers
- The equivalence classes in this case is:
Equivalence Classes for Kruskal’s

For Kruskal’s algo we will partition all vertices of the graph into disjoint sets, based on the equivalence relation: Are two vertices connected?

Q: The above equivalence relation partitions the graph into which of the following equivalence classes?

A. Connected subgraphs
B. Fully-connected (Complete) subgraphs
Application of Union-Find to Kruskal’s MST

- Vertices that form a connected subgraph will be in the same group.
- Connected subgraphs that are disconnected from each other will be in different groups.

The graph data structure

Q1: How can we check if adding an edge \((v, w)\) to the graph creates a cycle using the operations supported by union-find?

Q2: In Kruskal’s algo what would we like to do if adding the edge does not create a cycle?
Perform these operations:

Find(4) =
Find(3) =
Union(1,0) =
Find(4) =

- Each subtree represents a disjoint set
- The root node represents the set that any node belongs to
Running Time of Kruskal’s algorithm with union find data structure:

1. Sort edges in increasing order of cost

2. Set of edges in MST, T={}

3. For $i = 1$ to $|E|

   if (find(u) != find(v)){ //If $T \cup \{e_i=u,v\}$ has no cycles

   Add $e_i$ to $T$

   union(find(u), find(v))

}
Cycle Detection in a Graph

- DFS
- BFS
- Union-Find
Perform these operations:

Find(4) = 
Find(3) = 
Union(1,0)= 
Find(4)=
Array representation of Up-trees

- A compact and elegant implementation
- Each entry is the up index
- -1 for the roots
- Write the forest of trees, showing parent pointers and node labels, represented by this array

Find(4)
Performing a union operation

Union(6,7)

Fill in 6, 7 and 8 in the array

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>0</td>
<td>-1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>-1</td>
<td>-1</td>
<td>7</td>
</tr>
<tr>
<td>-1</td>
<td>0</td>
<td>-1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Disjoint subsets using up-trees
(Simple Union and Find)

Start with 4 items: 0, 1, 2, 3

Suppose $\text{Union}(i,j)$ makes the node $\text{Find}(i)$ the parent of the node $\text{Find}(j)$

Perform these operations:

- $\text{Union}(2,3)$
- $\text{Union}(1,2)$
- $\text{Find}(0) =$
- $\text{Find}(3) =$
- $\text{Union}(0,1)$
- $\text{Find}(1) =$

\begin{array}{cccc}
0 & 1 & 2 & 3 \\
-1 & -1 & -1 & -1 \\
\end{array}
Run time of Simple union and find (using up-trees)

If we have no rule in place for performing the union of two up trees, what is the worst case run time of find and union operations in terms of the number of elements N?

A. find: O(N), union: O(1)
B. find: O(1), union: O(N)
C. find: O(log₂N), union: O(1)
D. find: O(1), union: O(log₂N)
Improving Union operations

- In the Union operation, a tree becomes like a linked list if, when joining two trees, the root of the smaller tree (e.g. a single node) always becomes the root of the new tree.
- We can avoid this by making sure that in a Union operation, the *larger* of the two trees’ roots becomes the root of the new tree (ties are broken arbitrarily).

Henry Kautz, U. Washington
Smarter Union operations

- Avoid this problem by having the *larger* of the 2 trees become the root
- **Union-by-size**
  - Each root stores the size (# nodes) of its respective tree
  - The root with the larger size becomes the parent
  - Update its size = sum of its former size and the size of its new child
  - Break ties arbitrarily
Disjoint subsets using trees
(Union-by-size and simple Find)

Start with 4 items: 0, 1, 2, 3

Assume that $\text{Union}(i,j)$ makes the root of the smaller tree a child of the root of the larger tree.

Perform these operations:

- $\text{Union}(2,3)$
- $\text{Union}(1,2)$
- $\text{Find}(0) =$
- $\text{Find}(3) =$
- $\text{Union}(0,2)$
- $\text{Find}(1) =$
- $\text{Find}(0) =$
Smarter Union operations

- Avoid this problem by having the *larger* of the 2 trees become the root
  - **Union-by-size**
  - **Union-by-height (also called union by rank)**
    - Each root stores the height of its respective tree
    - If one root has a greater height than the other, it becomes the parent. Its stored height doesn’t need to be updated
    - If the roots show equal height, pick either one as the parent
    - Its stored height should be increased by one
    - Break ties arbitrarily
Disjoint subsets using trees
(Union-by-height and simple Find)

Start with 4 items: 0, 1, 2, 3

Assume that $\text{Union}(i,j)$ makes the root of the shorter tree a child of the root of the taller tree.

Perform these operations:

- Union(2,3)
- Union(1,2)
- Find(0) =
- Find(3) =
- Find(1) =
- Union(0,2)
- Find(0) =
Either union-by-size or union-by-height will guarantee that the height of any tree is no more than $\log_2 N$, where $N$ is the total number of nodes in all trees.

Q: What is the worst case run time of find and union operations in terms of the number of elements $N$?

A. find: $O(N)$, union: $O(1)$  
B. find: $O(1)$, union: $O(N)$  
C. find: $O(\log_2 N)$, union: $O(1)$  
D. find: $O(1)$, union: $O(\log_2 N)$

Solo vote!
Bounding the height of the up-tree using union by size.

- Initially all the nodes are in singleton trees (with height 1)
- Take the perspective of a single node.
- The only time the height of the tree which the node is part of increases by 1 is when the node joins a larger group i.e. if the height of the node’s up-tree increases by 1, the number of nodes in that tree would have at least doubled
- The maximum number of nodes in any tree is N, so the height of the resulting tree can be at most \( \log N \)
Cost of disjoint subsets operations with smarter Union and simple Find

Q: What is the worst case run time of find and union operations in terms of the number of elements N?

   A. find: \(O(N)\), union: \(O(1)\)
   B. find: \(O(1)\), union: \(O(N)\)
   C. find: \(O(\log_2 N)\), union: \(O(1)\)
   D. find: \(O(1)\), union: \(O(\log_2 N)\)

Discuss and revote

• Therefore, doing \(N-1\) union operations (the maximum possible) and \(M\) find operations takes time \(O(N + M \log_2 N)\) worst case

• With simple unions the complexity was:

• This is a big improvement; but we can do still better, by a slight change to the Find operation:
   adding *path compression*
Kruskal’s algorithm run time with smart union path compression find:

1. Sort edges in increasing order of cost

2. Set of edges in MST, T={}

3. For i= 1 to |E|
   
   if (find(u)!=find(v)) { //If T U {e_i=u,v} has no cycles
     Add e_i to T
     union(find(u), find(v))
   }

What is the improvement from simple union find?

Ref: Tim Roughgarden (stanford)
• Using the array representation for disjoint subsets, the code for implementing the Disjoint Subset ADT’s methods is compact

```cpp
class DisjSets{
    int *array;

    /**
     * Construct the disjoint sets object
     * numElements is the initial number of disjoint sets
     */
    DisjSets( int numElements ) {
        array = new int [ numElements ];
        for( int i = 0; i < numElements; i++ )
            array[ i ] = -1;
    }
}
```
/**
 * Union two disjoint sets using the height heuristic.
 * For simplicity, we assume root1 and root2 are distinct
 * and represent set labels.
 * root1 is the root of set 1
 * root2 is root of set 2
 * returns the root of the union
 */
int union ( int root1, int root2 ){

    if( array[ root2 ] < array[ root1 ] ) {
        // root2 is higher
        array[ root1 ] = root2;                // Make root2 new root
        return root2;
    } else {
        if( array[ root1 ] == array[ root2 ] )
            array[ root1 ]--;                // Update height if same
        array[ root2 ] = root1;                // Make root1 new root
        return root1;
    }
}
Simple Find

• In a disjoint subsets structure using parent-pointer trees, the basic Find operation is implemented as:

  • Go to the node corresponding to the item you want to Find the equivalence class for
  • Traverse parent pointers from that node to the root of its tree
  • Return the label of the root
  • This has worst-case time cost $O(\log_2 N)$
  • The time cost of doing another Find operation on the same item is the same
Find with Path-compression

• The path-compression Find operation is implemented as:
  • Go to the node corresponding to the item you want to Find the equivalence class for
  • Traverse parent pointers from that node to the root of its tree
  • Return the label of the root
  • But ….

… as part of the traversal to the root, change the parent pointers of every node visited, to point to the root of the tree (all become children of the root)

worst-case time cost is still $O(\log N)$
Example of path compression

As part of the traversal to the root, change the parent pointers of every node visited, to point to the root of the tree (all become children of the root). The worst-case time cost is still $O(\log N)$.

Henry Kautz
Cost of find with path compression

• What is the time cost of doing another Find operation on the same item, or on any item that was on the path to the root?

A. $O(\log N)$
B. $O(N)$
C. $O(1)$
Disjoint subsets using trees
(Union-by-height and path-compression Find)

• Start with 4 items: 0, 1, 2, 3
• \textbf{Union}(i,j) makes the root of the shorter tree a child of the root of the taller tree
• We perform path compression
• If an array element contains a negative \textit{int}, then that element represents a tree root, and the value stored there is -1 times (the height of the tree plus 1)

Perform these operations:
  \texttt{Union}(2,3)
  \texttt{Union}(0,1)
  \texttt{Find}(0) =
  \texttt{Find}(3) =
  \texttt{Find}(3) =
  \texttt{Union}(0,2)
  \texttt{Find}(1) =
  \texttt{Find}(3) =
  \texttt{Find}(3) =
Find with path compression

/**
 * Perform a find with path compression
 * Error checks omitted again for simplicity
 * @param x the label of the element being searched
 * @return the label of the set containing x
 */
int find( int x ) {

    if( array[ x ] < 0 )
        return x;
    else
        return array[ x ] = find( array[ x ] );
}

• Note that this path-compression find method does not update the disjoint subset tree heights; so the stored heights (called “ranks”) will overestimate of the true height

• A problem for the cost analysis of the union-by-height method (which now is properly called union-by-rank)
Self-adjusting data structures

- Path-compression Find for disjoint subset structures is an example of a *self-adjusting* structure

- Other examples of self-adjusting data structures are splay trees, self-adjusting lists, skew heaps, etc

- In a self-adjusting structure, a find operation occasionally incurs high cost because it does extra work to modify (adjust) the data structure, with the hope of making subsequent operations much more efficient

- Does this strategy pay off? *Amortized cost analysis* is the key to the answering that question...
Amortized cost analysis

- Amortization corresponds to spreading the cost of an infrequent expensive item (car, house) over the use period of the item.
- The amortized cost should be comparable to alternatives such as renting the item or taking a loan.
- Amortized analysis of a data structure considers the average cost over many actions.
Amortized cost analysis results for path compression

Find

- It can be shown (Tarjan, 1984) that with Union-by-size or Union-by-height, using path-compression, Find makes any combination of up to $N-1$ Union operations and $M$ Find operations have a worst-case time cost of $O(N + M \log^* N)$.

- This is very good: it is almost constant time per operation, when amortized over the $N-1 + M$ operations!
Amortized cost analysis results for path compression

- \log^* N = “log star of N” = smallest \( k \) such that \( \log^{(k)} n \leq 1 \) or 
  \# times you can take the log base-2 of \( N \), before we get a number \( \leq 1 \)
- Also known as the “single variable inverse Ackerman function”

- \( \log^* 2 = 1 \)
- \( \log^* 4 = 2 \)
- \( \log^* 16 = 3 \)
- \( \log^* 65536 = 4 \)
- \( \log^* 2^{65536} = 5 \)

- \log^{(k)} n = \log (\log (\log \ldots (\log n)))

- \log^* N grows extremely slowly as a function of \( N \)
- It is not constant, but for all practical purposes, \log^*N is never more than 5
Running Time of Kruskal’s algorithm with union find data structure:

1. Sort edges in increasing order of cost

2. Set of edges in MST, T={}  

3. For i= 1 to |E|  
   S1=find(u); S2=find(v);  
   if (S1!=S2){ //If T U {e_i=u,v} has no cycles  
   Add e_i to T  
   union(S1, S2)  
   }  

Ref: Tim Roughgarden (stanford)
Dijkstra’s Algorithm: Run time

- Initialize the graph: Give all vertices a dist of INFINITY, set all “done” flags to false
- Start at s; give s dist = 0 and set prev field to -1
- Enqueue (s, 0) into a priority queue. This queue contain pairs (v, cost) where cost is the best cost path found so far from s to v. It will be ordered by cost, with smallest cost at the head.
- While the priority queue is not empty:
  - Dequeue the pair (v, c) from the head of the queue.
  - If v’s “done” is true, continue
  - Else set v’s “done” to true. We have found the shortest path to v. (It’s prev and dist field are already correct).
  - For each of v’s adjacent nodes, w:
    - Calculate the best path cost, c, to w via v by adding the edge cost for (v, w) to v’s “dist”.
    - If c is less than w’s “dist”, replace w’s “dist” with c, replace its prev by v and enqueue (w, c)

What is the running time of this algorithm in terms of |V| and |E|? (More than one might be correct—which is tighter?)

A. O(|V|^2)
B. O(|E| + |V|)
C. O(|E| log(|V|))
D. O(|E| log(|E|) + |V|)
E. O(|E|*|V|)
Prim’s MST Algorithm: Run Time

1. Create an empty graph T. Initialize the vertex vector for the graph. Set all “done” fields to false. Pick an arbitrary start vertex s. Set its “done” field to true. Iterate through the adjacency list of s, and put those edges in the priority queue.

2. While the priority queue is not empty:
   • Remove from the priority queue the edge (v, w, cost) with the smallest cost.
   • Is the “done” field of the vertex w marked true?
     • If Yes: this edge connects two vertices already connected in the spanning tree, and we cannot use it. Go to 2.
     • Else accept the edge:
       • Mark the “done” field of vertex w true, and add the edge (v, w) to the spanning tree T.
       • Iterate through w’s adjacency list, putting each edge in the priority queue.

What is the running time of this algorithm in terms of |V| and |E|? (More than one might be correct—which is tighter?)

A. O(|V|^2)
B. O(|E| + |V|)
C. O(|E| log(|V|))
D. O(|E| log(|E|) + |V|)
E. O(|E|*|V|)