CSC2710 Analysis of Algorithms — Fall 2022
Unit 4 - Brute Force Algorithms

Book: §22.2 – §22.3; §32.1; §33.3 (intro) – 33.4 (intro)

4.1 Brute Force Algorithms

- Our first algorithm design strategy.
- By definition

  Definition 1 (Brute Force)**.** *A straight forward approach to solving a problem, usually directly based on the problem statement and definitions of the concepts involved.*

- You have often times used brute force algorithms in your travels.
- **Practice**: How would you compute a^n using a brute force algorithm?
- While not the most clever approach to algorithm design it is useful.
- Brute force is also the most *general* approach to algorithm design.
- Lastly, brute force provides a baseline for algorithms that solve the problem in question.
- The main characteristic of any brute force algorithm is:
  - Simplicity
  - Poor efficiency.

4.2 Brute Force Sorting

- Recall the sorting problem.

  Definition 2 (Sorting Problem)**.**
  Input*: A sequence of n numbers* $A = \langle a_1, a_2, \ldots, a_n \rangle$
  Output*: A permutation* $A' = \langle a'_1, a'_2, \ldots, a'_n \rangle$ *of the input such that* $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

- Ignore all the "fancy" sorts you may have learned in data structures, what is the simplest algorithm you can come up with to sort a sequence of numbers?
  - Describe the process, I'm not looking for an algorithm name!
- Here is one simple method starting with $i = 0$.
  1. Find the i-th smallest element in the sequence $a_1, a_2, \ldots, a_n$.
  2. Swap the smallest element in the sequence with the first element of the sequence to obtain the permutation:
     $$a'_1, a'_2, \ldots, a'_n.$$
  3. increment i and go to step 1.

- Let's look at an example of the idea in action. Let the original sequence be $3, 7, 5, 2, 1, 4$. In what follows bold numbers will denote the most recently placed number.

$3, 7, 5, 2, 1, 4$
$\mathbf{1}, 7, 5, 2, 3, 4$
$1, \mathbf{2}, 5, 7, 3, 4$
$1, 2, \mathbf{3}, 7, 5, 4$
$1, 2, 3, \mathbf{4}, 5, 7$
$1, 2, 3, 4, \mathbf{5}, 7$ No further swaps are done.
$1, 2, 3, 4, 5, 7$

- This method of sorting is called the *Selection Sort*.
- The selection sort algorithm is given as:

```
# Input: An array of comparable objects A, two indices x and y
# Output: The elements of A at indices x and y have swapped positions
def Swap(A, x, y):
    n = A[x]
    A[x] = A[y]
    A[y] = n

    # Input: An array A of integers
# Output: A is in sorted order
def SelectionSort( A ):
    for i in range( len(A) - 1 ):                                    # Line 1
        m = i                                                        # Line 2
        # Find the smallest element from element i to the end
        for j in range( i+1, len(A) ):                               # Line 3
            if( A[j] < A[m] ):                                       # Line 4
                m = j                                                # Line 5
        Swap( A, i, m )                                              # Line 6
```

- **Practice**: What is the worst case input for this algorithm?
- Lets build a chart to compute the running time of this sort in the worst case.

Line	Cost	Count	
1	c_1	$n-1$	
2	c_2	$n-1$	
3	c_3	$\sum_{i=0}^{n-1} t_i$	t_i is the number of iterations the loop performs on interation i.
4	c_4	$\sum_{i=0}^{n-1} t_i$	
5	c_5	$\sum_{i=0}^{n-1} t_i$	In the worst case.
6	c_6	$n-1$	In the worst case.

- What is the value of t_i?
  - Claim: $n - i - 1$
  - Derivation:

$$\begin{aligned} t_i &= (n-1) - (i+1) + 1 \\ &= n - 1 - i - 1 + 1 \\ &= n - i - 1 \end{aligned}$$

- You will notice that the running time of our algorithm is dominated by the sum $\sum_{i=0}^{n-2}(n-i-1)$. What is the closed form of this series?

$$\begin{aligned}
\sum_{i=0}^{n-2} t_i &= \sum_{i=0}^{n-1}(n-i-1) \\
&= \sum_{i=0}^{n-1} n - \sum_{i=0}^{n-1} i - \sum_{i=0}^{n-1} 1 && \text{(Appendix A Sum Manipulation Rule 2)} \\
&= n(n-1) - \sum_{i=0}^{n-1} i - (n-1) \\
&= n^2 - n - \sum_{i=0}^{n-1} i - n + 1 \\
&= n^2 - n - \left(\frac{(n-1)(n)}{2}\right) - n + 1 && \text{(Gaussian Sum – Appendix A)} \\
&= n^2 - n - \frac{1}{2}n^2 + \frac{3}{2}n - \frac{3}{2} - n + 1 \\
&= \frac{1}{2}n^2 - \frac{1}{2}n - \frac{1}{2} \\
&\leq n^2 && \text{(By upper bounding)}
\end{aligned}$$

- Since our dominating factor is bounded above by n^2 we can say that our running time is $\Theta\left(n^2\right)$.
- Interestingly enough the number of swaps needed is only $\Theta(n)$.
- Another sort that people think about when talking about brute force sorts is *bubble sort*
- The intuition behind the bubble sort is:
  - Compare adjacent elements in the list and exchange them if they are out of order. This will bubble up the largest element to the last element in the list.
  - Pass i bubbles up the i-th largest element in the same way.
    - Pass i only ends with elements $a_0, a_1, \ldots, a_{n-i-1}$ being unprocessed and elemetns $a_{n-i}, \ldots, a_{n-1}$ are in their correct spots.

- Let's see how this works for the list of numbers: $3, 7, 5, 2, 1, 4$. In what follows, the bold numbers represent the left most finalized number.

$3, 7, 5, 2, 1, 4$	Compare 3 and 7
$3, 5, 7, 2, 1, 4$	Compare 7 and 5 (swap)
$3, 5, 2, 7, 1, 4$	Compare 7 and 2 (swap)
$3, 5, 2, 1, 7, 4$	Compare 7 and 1 (swap)
$3, 5, 2, 1, 4, \mathbf{7}$	Compare 7 and 4 (swap)
$3, 5, 2, 1, 4, \mathbf{7}$	Compare 3 and 5
$3, 2, 5, 1, 4, \mathbf{7}$	Compare 5 and 2 (swap)
$3, 2, 1, 5, 4, \mathbf{7}$	Compare 5 and 1 (swap)
$3, 2, 1, 4, \mathbf{5}, 7$	Compare 5 and 4 (swap)
$2, 3, 1, 4, \mathbf{5}, 7$	Compare 3 and 2 (swap)
$2, 1, 3, 4, \mathbf{5}, 7$	Compare 3 and 1 (swap)
$2, 1, 3, 4, \mathbf{5}, 7$	Compare 3 and 4
$2, 1, 3, \mathbf{4}, 5, 7$	
$1, 2, 3, \mathbf{4}, 5, 7$	Compare 2 and 1 (swap)
$1, 2, 3, \mathbf{4}, 5, 7$	Compare 3 and 3
$\vdots$	
$1, 2, 3, 4, 5, 7$	Sequence is sorted

- The algorithm is given by

```
# Input: An array A of integers
# Output: A is in sorted order
def BubbleSort( A ):
    for i in range( len(A) - 1 ):                    # Line 1
        for j in range( len(A)- 1 - i ):             # Line 2
            if( A[j+1] < A[j] ):                     # Line 3
                Swap( A, j, j+1 )                    # Line 4
```

- **Practice**: What do you think the running time of this algorithm is? Why?

4.3 Sequential Search and Brute-Force String Matching

- We have already talked about sequential search. However, it is worthwhile to mention that it is a brute force algorithm. Please see lecture 3 for the analysis.
- We will focus on the string matching problem.

  Definition 3 (String Matching Problem)**.**

  Input*: An n symbol string*

  $$\sigma = \sigma_1\sigma_2\cdots\sigma_n$$

  and an $m \leq n$ symbol search string

  $$s = s_1 s_2 \ldots s_m.$$

  Output*: If there exists an index i such that*

  $$\sigma_i = s_1, \sigma_{i+1} = s_2, \ldots, \sigma_{i+m} = s_m,$$

  output i; otherwise, output $\perp$.

- How might you go about solving this problem using brute force?
  - Here is one way:
    - continually shift the patter one character right until it lines up. Once you have less than m characters to the right of the start of the match attempt you can safely stop.
- In pseudo-code we have the following string match algorithm:

```
# Input: An array A of n symbols, and an array S of m-n symbols
# Output: The index i of the start of the pattern S in A, or False
def StringMatch(A, S):
    for i in range(len(A)-len(S)+1):                # Line 1
        j = 0                                       # Line 2
        while( j < len(S) and S[j] == A[i+j] ):     # Line 3
            j = j + 1                               # Line 4
        if( j == len(S) ):                          # Line 5
            return i                                # Line 6
    return False                                    # Line 7
```

- Let's trace the algorithm on

  $$\sigma = \text{“racecar”}$$

  with pattern

  $$s = \text{“car”}$$

r	a	c	e	c	a	r
c	a	r				
	c	a	r			
		c	a	r		
			c	a	r	
				c	**a**	**r**

- The output is $i = 4$.
- Let's determine the worst case runnning time of this algorithm.

- **Practice**: What does the worst case input look like?
- Lets count the operations in our normal way

Line	Cost	Count
1	c_1	$n-m+1$
2	c_2	$n-m+1$
3	c_3	$(n-m+1)\,m$
4	c_4	$(n-m+1)\,m$
5	c_5	$n-m+1$
6	c_6	0
7	c_7	1

- $T(n,m) = 3\,(n-m+1) + 2m\,(n-m+1) + 1$
- Thus we have that running time is $O\,(mn)$. (**Practice:** Why?)
  - Note: We didn't say Θ, this bound is not necessarily tight. We would have to work through the rest of the parts of the equation to check.
- We will revist better solutions to the string matching problem during the course of the semester.

4.4 Closest-Pair and Convex-Hull Problems

- We will consider two very fundamental problems in computational geometry:
  1. The closest-pair problem
  2. The Convex-hull problem
- We define the closest-pair problem as follows:

  Definition 4 (Closest Pair Problem)**.**
  Input*: A set $P = \{p_1, p_2, \ldots, p_n\}$ of n points in d-dimensional space and a metric $\Delta : P \times P \to \mathbb{R}$.*
  Output*: The distance between the two closest points p_i and p_j according to the metric Δ.*

- For our purposes we will only consider the 2-D Euclidean plane and with the metric

$$\Delta(p_i, p_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}.$$

- What is the brute force approach here?
  - Compute the distance between each pair of distinct points and find the pair with the smallest distance.
- Formally, we have the algorithm

```
# Input: Two 2D points
# Output: The euclidean distance between the points
def distance( A, B ):
    return math.sqrt( (A[0] - B[0])**2 + (A[1] - B[1])**2 )

# Input: P is the list of n >= 2 points in XY space
# Output: The distance between the two closest points
def ClosestPair( P ):
    d = 10000000 # Basically, infinity                    # Line 1
    for i in range( len(P) - 1 ):                         # Line 2
        for j in range( i + 1, len(P) ):                  # Line 3
            d = min(d, distance( P[i], P[j] ) )           # Line 4
    return d                                              # Line 5
```

- What is the running time of this algorithm?
- Let's construct the the table

Line	Cost	Count	
1	c_1	1	
2	c_2	$n-1$	
3	c_3	$\sum_{i=1}^{n-1} t_i$	Where t_i is the number of iterations that occur for iteration i.
4	c_4	$\sum_{i=1}^{n-1} t_i$	Technically the square root is not necessarily a constant time operation. There do exists many constant time implementations so we will assume it is constant.
5	c_5	1	

- What's t_i?
  - $t_i = n - (i+1) + 1 = n - i - 1 + 1 = n - i$.

- Let's compute the closed form of our dominating factor $\sum_{i=1}^{n-1} t_i$.

$$\begin{aligned}
\sum_{i=1}^{n-1} t_i &= \sum_{i=1}^{n-1} (n-i) \\
&= \sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} i && \text{(Split the summation over subtraction)} \\
&= (n-1)\, n - \sum_{i=1}^{n-1} i \\
&= (n-1)\, n - \frac{(n-1)\, n}{2} && \text{(Closed form for Gaussian summation)} \\
&= n^2 - n - \frac{1}{2}n^2 + \frac{1}{2}n \\
&= \frac{1}{2}n^2 - \frac{1}{2}n \\
&\leq n^2
\end{aligned}$$

- The running time of the algorithm is therefore, $\Theta\left(n^2\right)$.
- We now turn our attention to what is considered the most important problem in computational geometry. *The convex hull problem.* Applications include:
  - Collision detection in video games.
  - Path planing
  - Processing satellite maps for accessibility
  - Detecting statistical outliers
  - Computing the diameter of a set of points (i.e. the distance between the two farthest points).
- First lets define what it means for a set to be convex.

  Definition 5 (Convex Set)**.** *A set of points (finite or infinite) in the plane is called* convex *if for any two points p and q in the set, the entire line segment $\overline{pq}$ belongs to the set.*

  - An example of a of a convex set is the set of points that make up a square, rectangle, parallelpiped, etc.
  - In fact, the vertices of any convex polygon form a convex set.

    Definition 6 (Convex Polygon)**.** *A polygon is* convex *if given any two points on its boundary or in its interior, all points on the line segment draw between them are contained in the polygon's interior or boundary.*

    * Intuitively, a convex polygon is any polygon with out "dents" it it.
- Given a set of n points in the plane we can think of the convex hull as the smallest convex polygon such that the points in the set are either inside or on the boundary of the polygon.
- Formally,

  Definition 7 (Convex Hull)**.** *The* convex hull *of a set S of points is the smallest convex set containing S.*

  - The smallest requirement means that the convex hull of S must be a subset of any convex set containing S.

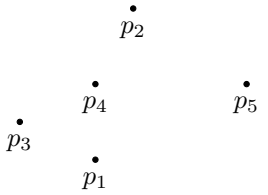
- Exams of convex hulls are:
  - S of S is a convex set.
  - If S is 2 points the convex hull is the line segment.
  - If S is three points not on the same line then the convex hull is a triangle.
  - etc.
- There is a useful theorem relating convex polygons to convex hulls

  Theorem 1. *The convex hull of any set S of $n > 2$ points not all on the same line is a convex polygon with the vertices as some of the points of S.*

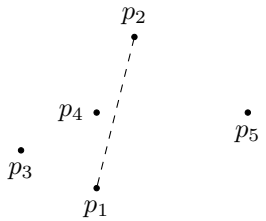
- We are now equipped to define the convex hull problem.

  Definition 8 (Convex Hull Problem)**.**
  Input*: A set S of n points*
  Output*: The list of extreme points that form the vertices of the convex polygon*

- We call a point an *extreme point* of a set S if it is a point that is not a middle point of any line segment with end points in S.
  - Example: extreme points for a triangle are its vertices.
  - Example: extreme points for a circle are the points on its circumference.
- There is a brute force method but, it is not obvious.
  - **Observation**: A line segment connecting two points p_i and p_j of a set of n points is part of the convex hull's boundary iff all other points of the set lie on the same side of as the straight line through these two points.
- Our brute force algorithm is nothing more then just repeating the test in our observation for *every* pair of points in the set S.
  - We collect these line segments so that we can return the list of extreme points when we are done.
- Let's work through a few steps on the following example set of points below:

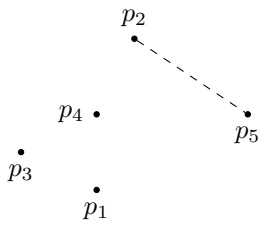


  - **Bad Segment Example**: We test the line segment between p_1 and p_2



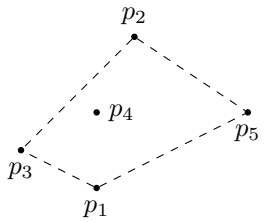
Notice that this line segment splits the plain in such a way that their are points on both sides of the line segment. Therefore this segment can not be part of the convex hull.

- **Good Segment Example**: We test the line segment between p_2 and p_5



Notice that every point in the plane is all on the same side of the line segment this geometric construct is called a *half-plane*. This segment is part of the convex hull.

- The convex hull for the entire set is:



- This is a *very* inefficient way of solving the problem. It can be shown to take $O\left(n^3\right)$ time. Why?
  - We need to check $\frac{n(n-1)}{2}$ line segments.
    - **Practice**: Why? Hint: think about definition of $\binom{n}{k}$
  - Each line segment requires us to check $(n-2)$ points.
  - Taking this together we have

$$(n-2)\frac{n\left(n-1\right)}{2} = \frac{(n-2)\left(n^2-n\right)}{2} = \frac{n^3-3n^2+2n}{2} \in O\left(n^3\right)$$

4.5 Depth-First Search (DFS) and Breadth-First Search (BFS)

- We can use exhaustive search to process all vertices and edges in a graph.
- There are two such algorithms we will investigate to perform this type of search
  1. Depth-First Search (DFS)
  2. Breadth-First Search (BFS)
- Both types of graph searches are *extremely* frequent in Computer Science.

4.5.1 DFS

- **Intuition**: You can think of a DFS as a walk of a graph that starts at a chosen source vertex and with each step visits a vertex one hop further away. Once it has visited all the vertices the maximum of n-hops away, it visits vertices $n-1$ hops away and so on.
  - We may occasionally refer to the distance between nodes in a graph as *hops*.
- High level algorithm
  1. Start at an arbitrary vertex (the source) and mark it as visited,
  2. proceed to visit an adjacent vertex.
     - Ties are broken arbitrarily.
     - As a matter of implementation ties are broken efficient per the data structure used to represent the graph.
  3. Process continues until a vertex with no adjacent unvisited vertices (a *dead vertex*) is encountered, go to the parent (predecessor) and try to continue.
  4. Once we backup to the starting vertex we stop.
  5. If there are still unvisited vertices, we must restart our search at one of the unvisited vertices.
     - If we must do this, we know that the graph is *unconnected*.
- When managing the vertices we are exploring what data structure makes sense?
  - Answer: A stack. We push a vertex on the stack when it is first encountered and pop it off the stack when it becomes a dead vertex.
- When talking about a DFS we will often talk of the *Depth-First Search Forest*.
  - A depth-first search forest is a tree constructed by a DFS.
    1. The first vertex visited by the DFS is called the root of the first tree in the forest.
    2. Each time an unvisited vertex is encountered it is attached as a child of it's predecessor.
       - This edge is called a *tree edge*.
       - Given and edge (u, v) traversed by the DFS algorithm, we call u the *predecessor* of v and v the descendant of u.
    3. If a visited vertex is discovered from a vertex already in the tree, then a special edge is used to connect the two this edge is called a *back edge*.
       - It's called a back edge as it connects a vertex to its ancestor.
- The basic algorithm for DFS occurs in two parts a DFS algorithm which calls a recursive algorithm DFS-Visit.
  - The DFS-Visit algorithm provides the stack for us through recursion.

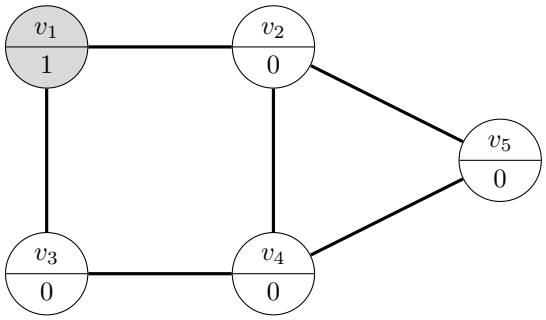
```
# Input: A Graph G=(E,V) and a count (global)
# Output: A graph G‘=(E‘,V‘) where E‘ = E and V‘=V with its
#   verticies marked with consecutive integers in the order that
#   the vertices were first encountered
def DFS( G ):
    global count
    count = 0                                   # Line 1
    # Mark every vertex in V with a 0
    for v in G:                                 # Line 2
        v.visited = 0                           # Line 3
    for v in G:                                 # line 4
        if( v.visited == 0 ):                   # Line 5
            DFS_Visit(v)                        # Line 6
```

- The DFS-Visit algorithm is as follows

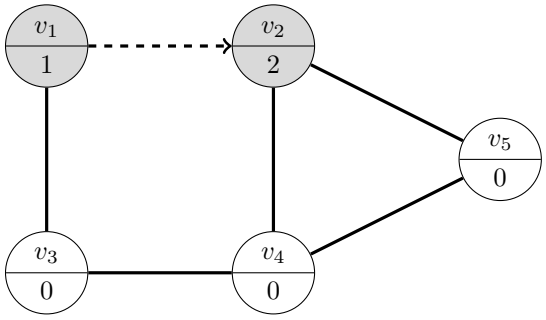
```
# Input: v is an unvisited vertex
# Output: All descendants of v are visited
def DFS_Visit( v ):
    global count
    count = count + 1                           # line 1
    v.visited = count                           # Line 2
    for u in v.get_connections():               # Line 3
        if( u.visited == 0 ):                   # Line 4
            DFS_Visit( u )                      # Line 5
```

- Let's look at an example trace. In what follows, the numbers represent the visit counter value, the tree edges are denoted by dashed lines with arrows, and the back edges are denoted by dotted lines with arrows.

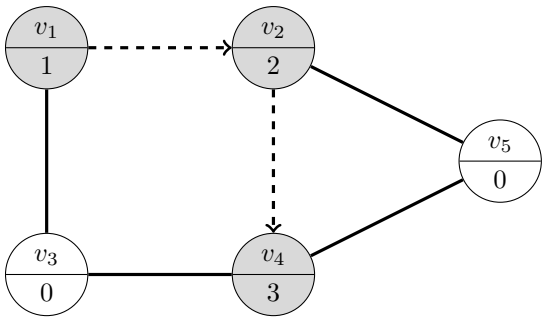
  1. Given the initial DFS graph, with the source vertex chosen to be v_1



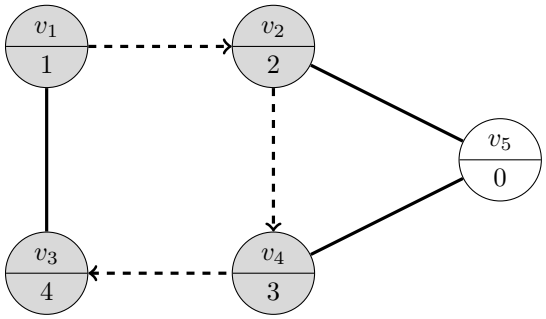
2. Visit vertex v_2 as it is adjacent to v_1



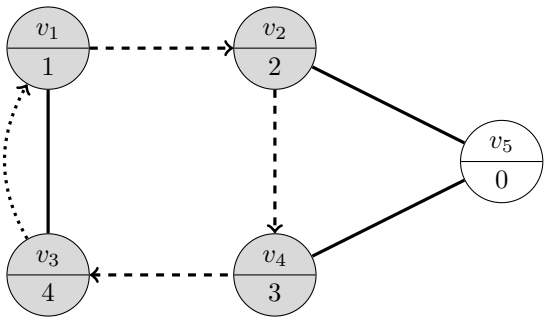
3. Visit vertex v_4 as it is adjacent to v_2



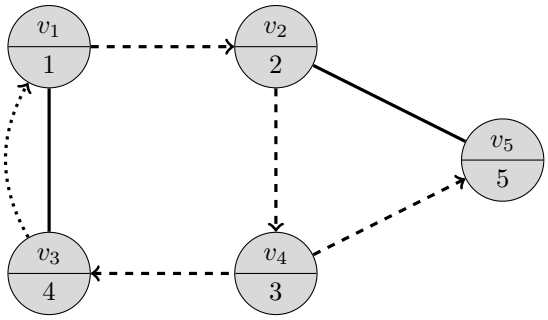
4. Visit vertex v_3 as it is adjacent to v_4



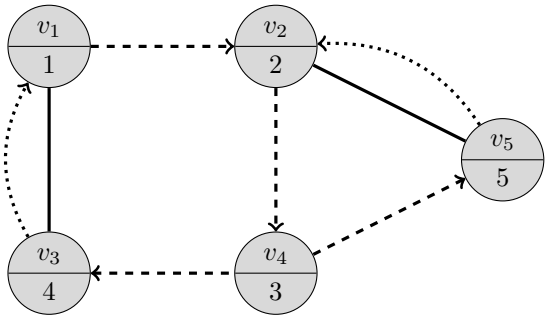
5. v_3's descendants have been visited so $v_3 \rightarrow v_1$ is a *back edge*



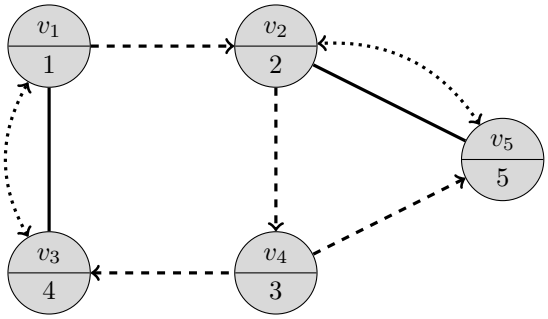
6. Back up to v_4 and visit its next adjacent vertex which in this case is v_5.



7. v_5 will not visit v_2 as it has already been visited so, $v_5 \to v_2$ is a back edge.



8. As our recursion unwinds we gain two new back-edges. Namely, $v_1 \to v_3$ and $v_2 \to v_5$.



- We can analyze the DFS algorithm as follows:
  - We assume we will use an adjacency list as it is most efficient.
  - Line 5 and 7-9 both take $\Theta(|V|)$ time in the worst case.
  - The DFS-Visit function is called *exactly* once for each vertex (*why*?). Lines 5-7 in DFS-Visit run $|\mathsf{Adj}_v|$ where Adj_v denotes the adjacency list for vertex v.
  - Since, every vertex is visited exactly once, we have that the loop in DFS-Visit is in

$$\sum_{v \in V} |\mathsf{Adj}_v| = \Theta(|E|)$$

  - Since it costs $\Theta(|V|)$ to run lines in DFS we have the total time in $\Theta(|V| + |E|)$.

- There are many interesting properties we can derive from the DFS forest.
  - **Graph Connectivity**: This question asks if the graph is connected. A graph is connected if there is a path from any vertex in the graph to any other vertex in the graph.

    - Once DFS-Visit is done running check to see if all vertices have been visited if not, the graph is not connected.
  - **Cycle Detection**: This questions asks if a graph has a cycle. In other words is their a path p through the graph such that $v_i \overset{p}{\rightsquigarrow} v_i$.
    - Inspecting the DFS forest one can determine if the graph is cyclic or acyclic.
    - The graph is cyclic if there are back edges in the DFS forest.
    - The graph is acyclic if there are no back edges in the DFS forest.

4.5.2 BFS

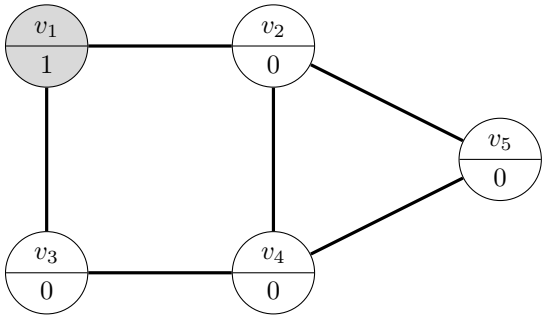
- **Intuition**: Starting at a source vertex we want to visit all vertices 1 hop away, visit all vertices 2 hops away, ..., visit all vertices $n-1$ hops away, and visit all vertices n hops away.
- We need to use a queue to maintain the list of discovered vertices.
  - Queue initially contains the source vertex.
  - At each iteration of the BFS, the algorithm:
    1. enqueue all vertices adjacent to the vertex that is at the front of the queue.
    2. Mark each newly inserted vertex visited.
    3. Dequeue the vertex at the front of the queue.
- If we finish our search from the source vertex, and not all nodes have been visited we must start BFS again from an arbitrary unvisited vertex.
- Similar to the DFS a BFS results in the generation of a BFS forest.
  - The source vertex becomes the root of the tree
  - When a vertex is reached for the first time, it is attached to its predecessor using a *tree edge*.
  - If vertex has already been visited and it is *not* the direct predecessor, then the edge connecting the two is a *cross edge*.
    - Note: Cross edges never connect vertices at different levels of the tree. They only join siblings or the parents siblings (uncles or aunts) in the tree.

- Like DFS the pseudo-code for BFS is divided into two algorithms BFS and BFS-Visit:

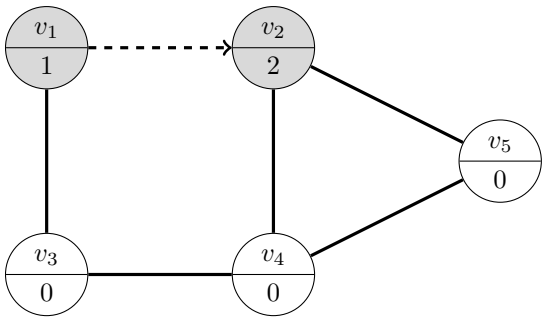
```
# Input: A Graph G=(E,V) and a count (global)
# Output: A graph G‘=(E‘,V‘) where E‘ = E and V‘=V with its
#   verticies marked with consecutive integers in the order that
#   the vertices were first encountered
def BFS( G ):
    global count
    count = 0                                       # Line 1
    # Mark every vertex in V with a 0
    for v in G:                                     # Line 2
        v.visited = 0                               # Line 3
    # Visit each vertex
    for v in G:                                     # Line 4
        if(v.visited == 0 ):                        # Line 5
            BFS_Visit( v )                          # Line 6

# Input: v is an unvisited vertex
# Output: All descendants of v are visited
def BFS_Visit( v ):
    global count
    count = count + 1                               # Line 1
    v.visited = count                               # Line 2
    # Create a Queue, enqueueing the vertex
    Q = [v]                                         # Line 3
    # While there are still elements in the queue,
    while( len(Q) > 0 ):                            # Line 4
        for u in v.get_connections():               # Line 5
            if( u.visited == 0 ):                   # Line 6
                count = count + 1                   # Line 7
                u.visited = count                   # Line 8
                Q.append(u)                         # Line 9
        Q.pop(0) # Dequeue the first element        # Line 10
```

- Let's look at an example trace of BFS on the same graph we used for the DFS. In what follows, the numbers represent the visit counter value, the tree edges are denoted by dashed lines with arrows, and the cross edges are denoted by dotted lines with arrows.

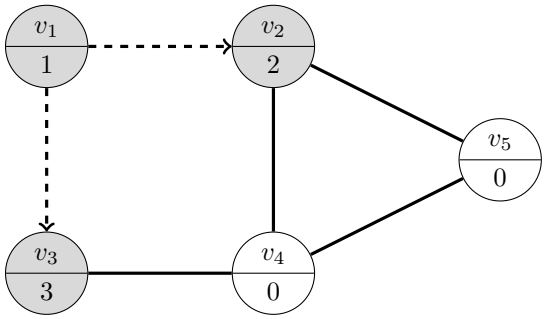
1. The source vertex is v_1 and $Q = v_1$.



2. We visit the first descendant of v_1 and $Q = v_1, v_2$

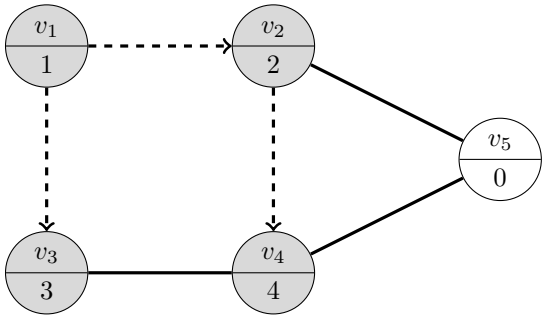


3. We visit the next descendant of v_1 and $Q = v_1, v_2, v_3$

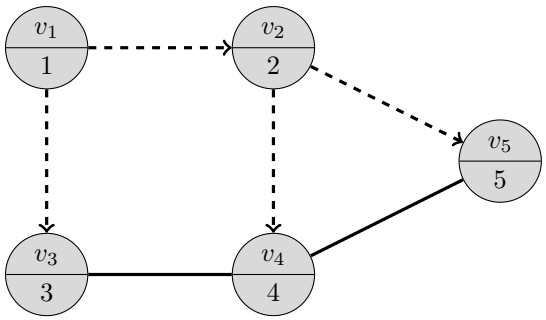


4. v_1 has no more descendants, it is removed from the queue. The queue is now $Q = v_2, v_3$.

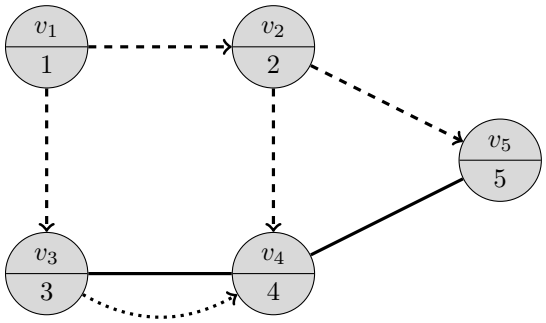
5. We visit the first descendant of v_2, $Q = v_2, v_3, v_4$



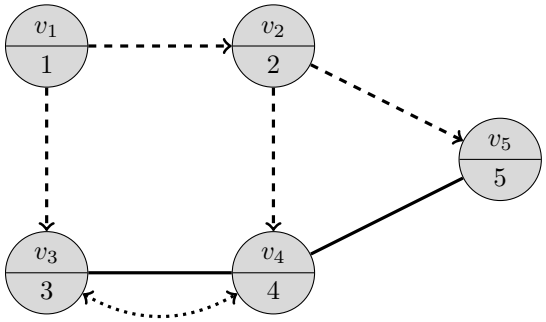
6. We visit the second descendant of v_2, $Q = v_2, v_3, v_4, v_5$



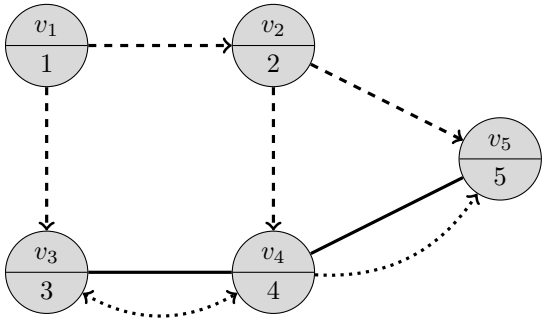
7. v_2 has no more descendants, it is removed from the queue. The queue is now $Q = v_3, v_4, v_5$.
8. v_3's only descendant is v_4 which has already been visited this means we have a cross edge. v_3 is removed from the queue so $Q = v_4, v_5$



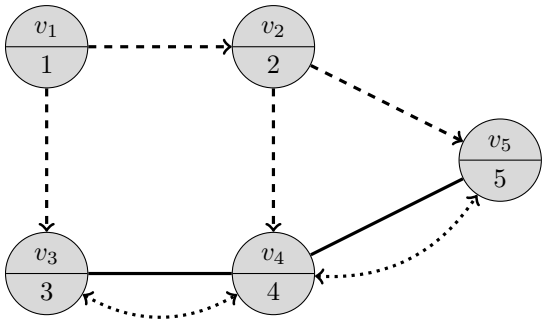
9. v_4's first descendant is v_3 which has already been visited this means we have a cross edge.



10. v_4's second descendant is v_5 which has already been visited this means we have a cross edge. Since, there are no additional descendants v_4 is removed from the queue. This results in $Q = v_5$.



11. v_5's only descendant is v_4 which has already been visited this means we have a cross edge. v_5 is removed from the queue so $Q = \emptyset$ and our BFS has finished since there are no unvisited vertices in G.



- For similar reasons to the DFS, our running time of BFS on a graph represented using an adjacency list has running time $\Theta(|V| + |E|)$.
  - We only enqueue (and dequeue) a vertex once so this leads to $\Theta(|V|)$ queue operations.
  - Each vertex u we encounter we process it's complete adjacency list Adj_u.
  - The total amount of work to scan the adjacency lists is

$$\sum_{u \in V} |\text{Adj}_u| \in \Theta(|E|).$$

- Useful properties of a BFS forest

- Easy to perform cycle detection by looking for cross-edges.
  - Giving the cycle is substantially more work with BFS, it is almost trivial with DFS.
- Graph connectivity is checked the exact same way as it is in the case of a DFS.
- In an unweighted graph, the BFS tree provides us with the shortest distance between any two vertices (in terms of hops)

Challenge Problems

1. Given an array of integers `A`,

   - Write a brute-force algorithm in Python to determine if the array of integers is in sorted order. Determine its running time and space requirements.
   - Fun with averages (be sure to show your work when finding the running times). You may *not* assume the input array is sorted, nor should you sort the array, as that is not a brute-force approach.
     - Develop a brute-force algorithm to compute the mean value of an array of integers. What is its asymptotic running time?
     - Develop a brute-force algorithm to compute the median value of an array of integers. What is its asymptotic running time?
     - Develop a brute-force algorithm to compute the mode of an array of integers. What is its asymptotic running time?

2. BFS finds the shortest distance to any destination node from a source node. Now imagine you are tasked with finding the shortest path in a *weighted* graph using a variation of BFS. How might you adjust the algorithm? What data structure(s) would you use?