CSC 6013 Algorithms and Discrete Structures

Michael Bradley

Class Notes for Week 5 Brute force algorithms

Read these sections from the textbook: §2.3-2.3 sorting (including brief mentions of selection sort (p29) and bubble sort (p40)), §22.1 - 22.4 (Depth-first and Breadth first searches), §32.1 (string matching), §33.3 (convex hull), §33.4 (closest pair)

In this unit we look at:

A. Brute-force algorithms:
- understanding brute force algorithms
- determining their asymptotic performance using summations

  examples:
  - selection sort
  - bubble sort
  - string matching
  - closest pair
  - convex hull (no python code)
  - depth first search (DFS)
  - breadth first search (BFS)

B. Notes on mathematical prerequisites:
- combinations
- review of a few summation formulas
- distance between two points

C. Notes on data structures prerequisites:
- linked lists
- representing graphs using adjacency matrices

A. Brute force algorithms

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

For example, a brute force algorithm to find the sum of the entries in an array would be to initialize a total to zero, then use a loop to add each entry to the total so far. The algorithms we discussed in the fourth class session to find the number of positive values in an array, to find the greatest common divisor of two positive integers, and the sequential search algorithm are brute force algorithms.

Brute force is the most general approach to algorithm design. While not the most clever approach to algorithm design, it is useful. Brute force provides a baseline for algorithms that solve the problem in question. The main characteristics of any brute force algorithm are simplicity, but poor efficiency.

Brute Force Sorting
Recall the sorting problem.
Definition (Sorting Problem).
Input: A sequence of n numbers A = <a1, a2,…, an>
Output: A permutation A’ = <a’1, a’2,…, a’n> of the input such that a’1 ≤ a’2 ≤ … ≤ a’n

Example 1: Selection Sort

Here is one simple method:
Find the smallest element in the sequence; swap it into slot 1.
Find the 2nd smallest element in the sequence; swap it into slot 2.
Find the 3rd smallest element in the sequence; swap it into slot 3.
etc.

Let's look at an example of the idea in action. Let the original sequence be
33; 77; 55; 22; 11; 44.
In what follows bold numbers will denote the most recently placed number.
33; 77; 55; 22; 11; 44 This is the original array A.

11; 77; 55; 22; 33; 44
11; ***22*;** 55; 77; 33; 44
11; 22**; 33;** 77; 55; 44
11; 22; 33; **44**; 55; 77
11; 22; 33; 44; **55**; 77 No further swaps are done. This is the result A’.

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

```
# Input: An array A and indices i and j.
# Output: An array where A[i] and A[j] have been swapped.
def Swap(A, i, j):
        temp = A[i]
        A[i] = A[j]
        A[j] = temp

# Input: An array A of integers.
# Output: Array A sorted in increasing order.
def SelectionSort(A):
        for i in range(len(A)-1):
                m = i

        # Find the smallest element in A[i .. n-1]
                for j in range(i+1, len(A)):
                        if A[j] < A[m]:
                                m = j
                Swap(A, i, m)
```

What is the worst-case input for this algorithm?
The comparison of keys in line 16 is the crucial operation we need to track.
In line 11, the for loop with index i ranges from i=0 to i = n-2.
In line 15, the for loop with index j ranges from i+1 to n-1.
For each value of i, the comparison of keys in this inner loop (line 16) will be executed (n-1) – (i+1) +1 times. This simplifies to

$$(n-1)-(i+1)+1 = n-1-i-1+1 = n-i-1$$

So, the maximum number of key comparisons in any run of this algorithm will be

$$\sum_{i=0}^{n-2}(n-i-1)$$

We can separate this into three separate sums

$$= \sum_{i=0}^{n-2} n - \sum_{i=0}^{n-2} i - \sum_{i=0}^{n-2} 1$$

The first sum is n-1 copies of n.
The second sum is 0 + 1 + 2 + … + (n-2).
The third sum is n-1 copies of 1.

$$= (n-1)n - \frac{(n-2)(n-1)}{2} - (n-1)1$$
$$= n^2 - n - \frac{n^2}{2} + \frac{3n}{2} - \frac{2}{2} - n + 1$$
$$= \frac{1}{2}n^2 - \frac{1}{2}n$$
$$< n^2$$

So, this selection sort algorithm in the asymptotic class $O(n^2)$.

A simpler analysis would be to notice that in the worst case, the number of times the comparison in line 16 is executed is (n-1) + (n-2) + (n-3) + …. + 3 + 2 + 1 = $\frac{(n-1)n}{2}$ which is $O(n^2)$.

It is interesting that the number of swaps (the number of calls to the swap function) is O(n).

example 2 - Bubble Sort

Another sort that people think about when talking about brute force sorts is bubble sort.
The intuition behind the bubble sort is:

Walk through the array comparing adjacent elements in the list,
exchanging them if they are out of order.

The first pass will cause the largest element to “bubble up” and end up in the last slot in the list.
Pass i bubbles up the i-th largest element in the same way.
After pass i, at least the i largest elements will be located in the last i slots of the array, so do not do any comparisons in this sorted tail portion of the array.

Let's look at an example of the bubble sort idea in action. Let the original sequence be
33; 77; 55; 22; 11; 44.
In what follows bold numbers will denote the most recently compared numbers.
33; 77; 55; 22; 11; 44 This is the original array A.

33; 77; 55; 22; 11; 44 compare 33 and 77
33; **77; 55;** 22; 11; 44 compare 77 and 55, and swap them
33; 55; **77; 22**; 11; 44 compare 77 and 22, and swap them
33; 55; 22; **77; 11**; 44 compare 77 and 11, and swap them
33; 55; 22; 11; **77; 44** compare 77 and 44, and swap them
33; 55; 22; 11; 44; **77**
finished 1st pass through array, the last item is in its correct place
33; 55; 22; 11; 44; 77 compare 33 and 55
33; **55; 22**; 11; 44; 77 compare 55 and 22, and swap them
33; 22; **55; 11;** 44; 77 compare 55 and 11, and swap them
33; 22; 11; **55; 44**; 77 compare 55 and 44, and swap them
33; 22; 11; 44; 55; 77
finished 2nd pass through array, the last 2 items are in their correct places
33; 22; 11; 44; 55; 77 compare 33 and 22, and swap them
22; **33; 11**; 44; 55; 77 compare 33 and 11, and swap them
22; 11; **33; 44**; 55; 77 compare 33 and 44
22; 11; 33; 44; 55; 77
finished 3rd pass through array, the last 3 items are in their correct places
22; 11; 33; 44; 55; 77 compare 22 and 11, and swap them
11; **22; 33**; 44; 55; 77 compare 22 and 33
11; 22; 33; 44; 55; 77
finished 4th pass through array, the last 4 items are in their correct places
11; 22; 33; 44; 55; 77 compare 11 and 22
11; 22; 33; 44; 55; 77
finished 5th pass through array, the last 5 items are in their correct places
No further swaps are done. This is the result A’.

The Bubble sort algorithm is given as:

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

Note: We did not include the code to define the Swap function since we already did this in the previous example. However, these seven lines of code will not work as a stand-alone block of code without the code for the Swap function.

What is the worst-case input for this algorithm?
The comparison of keys in line 6 is the crucial operation we need to track.
In line 4, the for loop with index i ranges from 0 to n-2.
In line 5, the for loop with index j ranges from 0 to n-2-i.
For each value of i, the comparison of keys in this inner loop (line 6) will be executed n-i-1 times.
So, the maximum number of key comparisons in any run of this algorithm will be

$$\sum_{i=0}^{n-2}(n-i-1)$$

This is the same summation as the selection sort algorithm, but we will evaluate it differently. Notice that the summation = (n-1) + (n-2) + (n-3) + … + 3 + 2 + 1 which, if we read this list of terms from right to left, can be written as

$$=\sum_{k=1}^{n-1}k$$

and we know a formula for this type of summation

$$=\frac{(n-1)n}{2}=\frac{1}{2}n^2-\frac{1}{2}n<n^2$$

So, this bubble sort algorithm in the asymptotic class $O(n^2)$.

As with selection sort, a simpler analysis of the worst case work in bubble sort would be to notice that the number of times the comparison in line 6 is executed is (n-1) + (n-2) + (n-3) + …. + 3 + 2 + 1 = $\frac{(n-1)n}{2}$ which is $O(n^2)$.

example 3 - String Matching

Another classic problem that has a brute force solution is the string matching problem.
Definition (String Matching Problem).
Input: An n symbol string $\sigma = \sigma_1\sigma_2\sigma_3 \dots \sigma_n$ and an m≤n symbol search string $s = s_1s_2s_3 \dots s_m$
Output: If there exists an index i such that $\sigma_i = s_1, \sigma_{i+1} = s_2, \sigma_{i+2} = s_3, \sigma_{i+m-1} = s_m$ then output i; otherwise, output None.

How might you go about solving this problem using brute force?
Here is one way:

Continually shift the pattern one character right until it lines up. Once you have less than m-1 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:

```
1    # Input: An array A of n symbols and an array S of m < n
2    #     symbols.
3    # Output: The index i of the start of the pattern in A or
4    #     None, otherwise.
5    def StringMatch(A, S):
6        for i in range(len(A)-1):
7            j = 0
8            while j < len(S) and A[i+j] == S[j]:
9                j = j+1
10           if j == len(S):
11               return i
12       return None
```

Let's trace the algorithm on array of symbols σ= "racecar" with pattern s = "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 (indicating that the pattern was found starting at position 4 in the array.)

The comparison of symbols A[i+j] == S[j] in line 8 is the crucial operation we need to track.
In line 6, the for loop with index i ranges from 0 to n-2.

In line 8, the while loop with index j ranges from 0 to m-1 in the worst case.

So, the maximum number of key comparisons in any run of this algorithm will be

$$\sum_{i=0}^{n-2} m$$

Since m is constant, the sum is simply

$$= (n-2)m = nm - 2m < nm$$

So, the running time of the string-matching algorithm is *O(nm)*.

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

example 4 - Closest Pair

We define the closest-pair problem as follows:
Definition (Closest Pair Problem).
Input: A set $P = \{p_1, p_2, p_3, \ldots, p_n\}$ of n points in d-dimensional space and a metric
$d: P \times P \to R$
Output: The distance between the two closest points pi and pj according to the metric d.

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

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

What is the brute force approach here?

For every pair of points, compute the distance between the two points and find the pair with the smallest distance.

Formally, we have the algorithm:

```
1     # Input: Two points A and B in the XY plane.
2     # Output: The Euclidean distance between the two points.
3     def Distance(A, B) :
4         return math.sqrt ((A[0]-B[0])**2 + (A[1]-B[1])**2)
5
6     # Input: P a list of n>2 points in the XY plans.
7     # Output: The distance between the closest pair of points.
8     def ClosestPair(P) :
9         d = 1e12  # a large number, might as well be infinity
10        for i in range(len(P)-1) :
11            for j in range(i+1, len(P)) :
12                d = min (d, Distance(P[i], P[j]))
13        return d
```

Note: math.sqrt() might require you to import math.

What is the running time of this algorithm?

The calculation of the distance between two points in line 12 is the crucial operation we need to track.
In line 10, the for loop with index i ranges from 0 to n-2.

In line 11, the while loop with index j ranges from i+1 to n-1 in the worst case. The body of this loop will be executed (n-1) – (i+1) +1 = n – i -1 times.

So, the maximum number of key comparisons in any run of this algorithm will be

$$\sum_{i=0}^{n-2}(n-i-1)$$

This is the same summation as the selection sort and bubble sort algorithms, so the work in the worst case is

$$=\frac{(n-1)n}{2}=\frac{1}{2}n^2-\frac{1}{2}n$$

So, this closest pair algorithm in the asymptotic class $O(n^2)$.

example 5 - Convex Hull

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 planning
- 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 let's define what it means for a set to be convex.
Definition (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 pq belongs to the set.
Examples of convex sets are the set of points that make up a square, rectangle, parallelepiped, etc. In fact, the vertices of any convex polygon form a convex set.

The convex hull of a set of points is the boundary of the region in the plane that includes the line segments joining all pairs of points in the set.

"Same Side Test": A line segment connecting two points pi and pj of a set of n points is part of the convex hull's boundary if all other points of the set lie on the same side of the straight line through these two points.

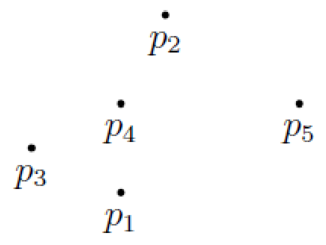
Given a set of n points in the plane we can think of the convex hull as the smallest convex polygon such that the points in the set are either inside or on the boundary of the polygon.

Exams of convex hulls are:
If S is a single point, S is its own convex hull.
If S is 2 points the convex hull is the line segment joining the two points.
If S is three points not on the same line then the convex hull is a triangle.

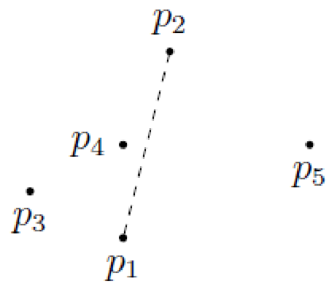
We are now equipped to define the convex hull problem.
Definition 8 (Convex Hull Problem).
Input: A set S of n points
Output: The list of points that form the vertices of the convex polygon

Our brute force algorithm involves repeating the "Same Side Test" for every pair of points in the set S, and compiling a list of points whose line segments pass the test.

We will not provide pseudocode for this algorithm, but we will work through a few steps on the following example set of points:

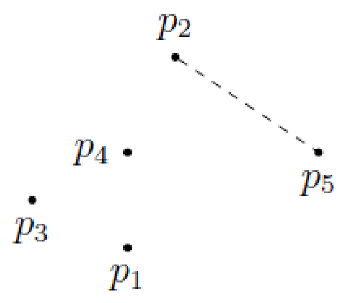


Bad segment example: Test the line between points p1 and p2.



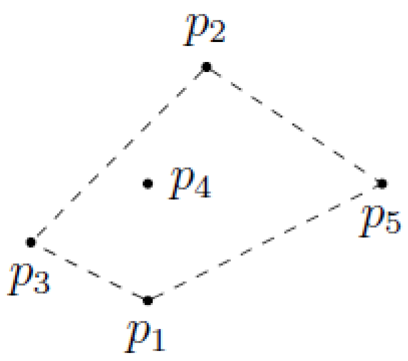
This line segment “splits the plane” in such a way that there are points on both sides of the line segment. Therefore, this segment cannot be part of the convex hull.

Good segment example: Test the line between points p2 and p5.



This line segment “splits the plane” in such a way that all remaining points lie on the same side of the line segment. Therefore, this segment is part of the convex hull.

The convex hull for the entire set is



so, our algorithm would return the subset of points {p1, p2, p3, p5} that lie on the convex hull.

This is a very inefficient way of solving the problem (as most brute force algorithms are). For every pair of points, we need to consider the line segment joining these two points and check to see if all remaining points lie on the same side of the line segment.

For a set of n points, how many line segments are there? There is one line segment for each pair of points, so there are "n choose 2" = $nC2 = \binom{n}{2} = \frac{n(n-1)}{2}$ segments. For each line segment, we need to determine on which side of the line does each of the remaining n-2 points lie. How we do this is quite complicated. But clearly, this is the essential unit of work that we need to count. The counting is simple in the worst case:

$$\frac{n(n-1)}{2} \cdot (n-2) = \frac{(n^2-n)(n-2)}{2} = \frac{n^3-3n^2+2n}{2} < n^3$$

so, this brute force algorithm for the convex hull of a set of n points is $O(n^3)$.

Searching in Graphs

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. We will implement both algorithms using the adjacency list representation of a graph.

BFS is a brute-force algorithm that uses a queue to organize the work to be done next. Algorithm DFS is a recursive algorithm, we will cover it here because it is logically related to BFS. We could have implemented DFS as a brute-force algorithm using a stack, but that is exactly what is taking place behind the scenes in recursion.

example 6 - Depth First Search (DFS)

The basic idea of Depth First Search is to start at a chosen vertex, visits one of its not-yet-visited neighbors, then one of that vertex's not-yet-visited neighbors, ... If this process reaches a vertex having no unvisited neighbors, it "backs up" to the most recent vertex that does have an unvisited neighbor, and goes forward from there. This forward and backward concept continues until all vertices in the graph have been visited. The algorithm gets its name from its attempt to go as deep into the graph as possible before backing up and trying a different direction.
In more detail with some technical terms, the algorithm for DFS is:
High level algorithm
1. Start at an arbitrary vertex (the source) and mark it as "visited".
2. Proceed to visit an "adjacent" vertex. (This edge is called a "tree edge")
   Ties are broken arbitrarily.
   As a matter of implementation, ties are broken efficiently as 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. (We insert a new "back edge" that goes from the vertex to its previously visited neighbor.) 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.
The algorithm for DFS occurs in two parts - a DFS algorithm which calls a recursive

algorithm DFSVisit. The DFS-Visit algorithm provides the stack for us through recursion.

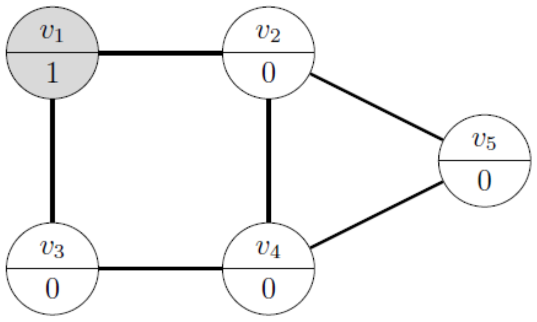
```
# Input: A graph G = (V, E) and a global variable, count.
# Output: A graph G = (V,' E') where E' = E and V' = V
#         except that the nodes of V' are marked with integers
#         that indicate in what order they were visited.
def DFS(G):
        global count
        count = 0

# Mark every vertex in V with 0.
for v in G:
        v.visited = 0

for v in G:
        if v.visited == 0:
                DFSVisit (v)
```

The DFSVisit algorithm is as follows:

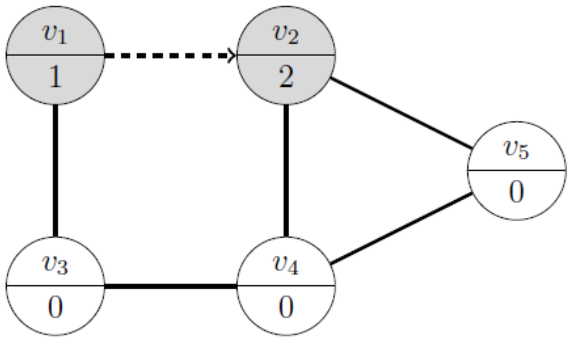
```
# Input: v is an unvisited vertex.
# Output: All descendants of v have been visited.
def DFSVisit(v):
        count = count + 1
        v.visited = count
        for u in v.getAdjacent():
                if u.visited == 0:
                        DFSVisit(u)
```

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.

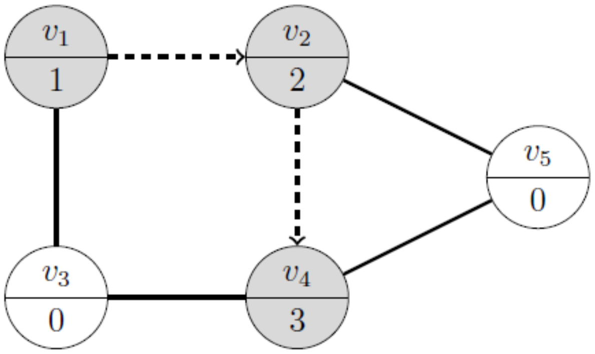
1. Given the initial DFS graph, with the source vertex chosen to be vertex v1.



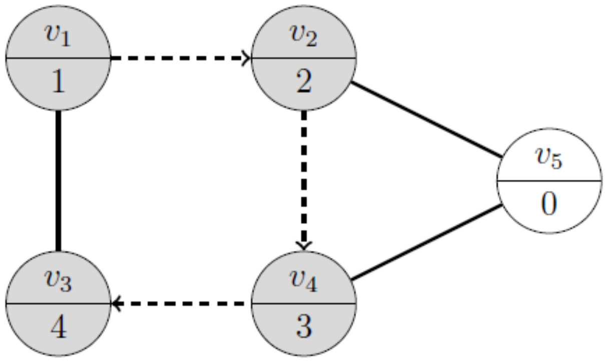
2. Vertex v1's neighbors are v2 and v3 (in that order). Visit vertex v2. Edge (v1, v2) becomes a tree edge. Vertex v2 becomes the 2nd vertex to be visited.



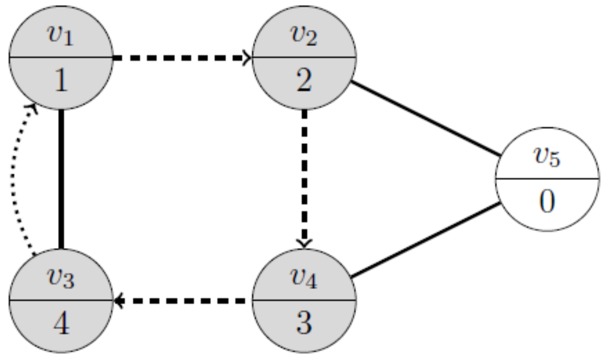
3. Vertex v2's neighbors are v4 and v5 (in that order). Visit vertex v4. Edge (v2, v4) becomes a tree edge. Vertex v4 becomes the 3rd vertex to be visited.



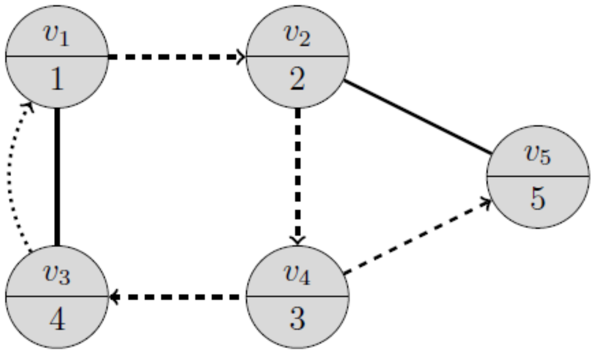
4. Vertex v4's neighbors are v3 and v5 (in that order). Visit vertex v3. Edge (v4, v3) becomes a tree edge. Vertex v3 becomes the 4th vertex to be visited.



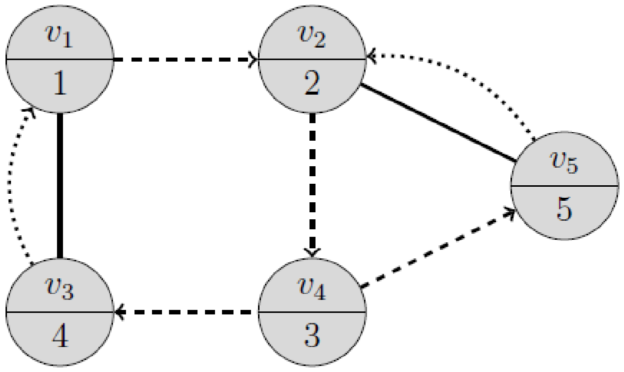
5. All of vertex v3's descendants (v1 and v4) have been visited so edge (v3, v1) becomes a back edge.



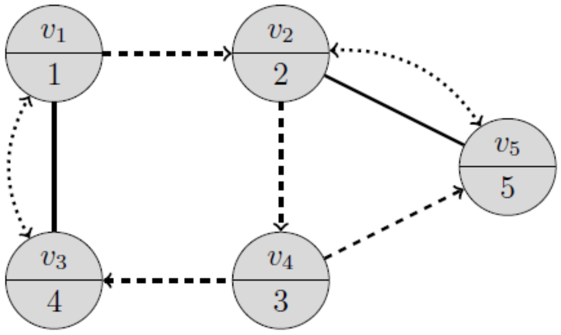
6. Back up to v4 and visit its next adjacent vertex which in this case is v5. Edge (v4, v5) becomes a tree edge. Vertex v5 becomes the 5th vertex visited.



7. Vertex v5 will not visit v2 as it has already been visited so, edge (v5, v2) becomes a back edge.



8. As our recursion unwinds we gain two new back-edges, (v1, v3) and (v2, v5). We draw these as a single edge with arrows on both ends rather than drawing two edges, one in each direction.



What is the work done by the DFS algorithm in the worst case? We can identify the crucial unit of work to be the call to DFSVisit, either from DFS() or recursively from DFSVisit().
In DFS(), after marking each vertex as unvisited, the algorithm calls DFSVisit() at most once for each vertex. So, this bound is $|V|$, the number of vertices in the graph.
In DFSVisit(), the algorithm recursively calls itself at most once for each edge in the adjacency list of the "parameter vertex". Since each edge appears twice in the graph's adjacency structure, this bound is $2|E|$ or twice the number of edges in the graph.

Therefore, the number of calls to DFSVisit is $\leq |V| + 2|E| < 2(|V| + |E|)$, so the DFS algorithm's asymptotic class is $O(|V| + |E|)$.

If the DFS() algorithm finishes, without visiting all vertices in the graph, we restart it from one of the unvisited vertices. This forms a DFS forest of separate subgraphs.

There are many interesting properties we can derive from the run of the DFS algorithm on a graph. Two of these are connectivity and cycles.

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 DFSVisit is done running check to see if all vertices have been visited. If not, the graph is not connected. The number of times it needs to be restarted is the number of connected components.

Cycle Detection: This question asks if a graph has a cycle. In other words, is there a path p through the graph from a vertex back to itself?
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.

example 7 - Breadth First Search (BFS)

The basic idea of Breadth First Search is to start at a chosen vertex, visits all of its not-yet-visited neighbors, then all of their not-yet-visited neighbors, ... If the graph is connected (if there is a path from the source vertex to every other vertex), all vertices will eventually be visited. The algorithm gets its name from its attempt to go as wide as possible before proceeding to vertices that are another level away from the source.

A high-level algorithm using a queue is as follows:

We use a queue to maintain the list of discovered vertices.
Queue initially contains the source vertex.
At each iteration of the BFS 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 what we did in DFS, when a visited vertex considers one of its visited neighbors, we insert a new "cross edge" that goes from the vertex to its previously visited neighbor.
Like DFS the pseudo-code for BFS is divided into two algorithms BFS and BFSVisit.

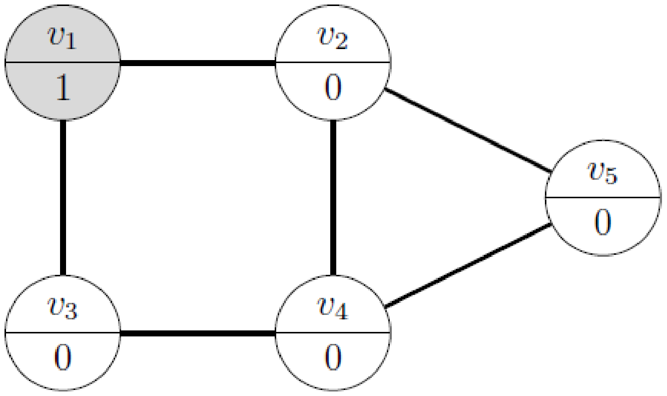
```
# Input: A graph G = (V, E) and a global variable, count.
# Output: A graph G = (V,’ E’) where E’ = E and V’ = V
#       except that the nodes of V’ are marked with integers
#       that indicate in what order they were visited.
def BFS(G):
    global count
    count = 0

# Mark every vertex in V with 0.
for v in G:
    v.visited = 0

for v in G:
    if v.visited == 0:
        BFSVisit(v)
```

The BFSVisit algorithm is as follows:

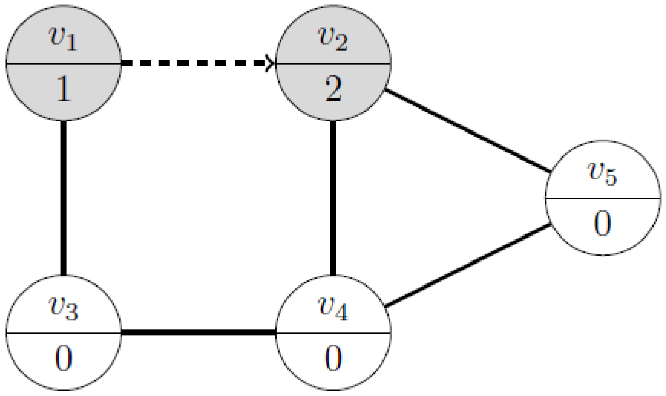
```
# Input: v is an unvisited vertex.
# Output: All descendants of v have been visited.
def BFSVisit(v):
    count = count + 1
    v.visited = count

    # Create a queue, enqueueing the vertex.
    Q = [v]
    for u in v.getAdjacent():
        count = count + 1
        u.visited = count
        Q.append(u)
    Q.pop()
```

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.

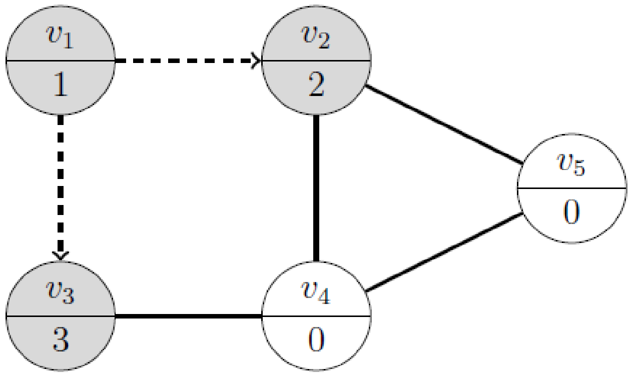
1. The source vertex is v1 and Q = [v1]. Mark v1 as the 1st visited vertex.



2. We visit the first descendant of v1 and Q = [v1, v2]. Mark v2 as the 2nd visited vertex. Make edge (v1, v2) a tree edge.

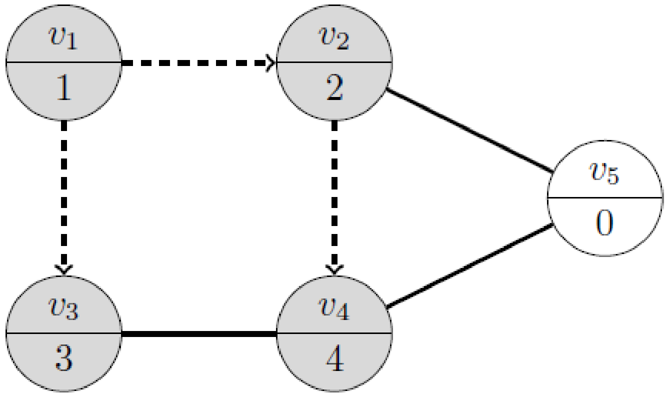


3. We visit the next descendant of v1 and Q = [v1, v2, v3]. Mark v3 as the 3rd visited vertex. Make edge (v1, v3) a tree edge.

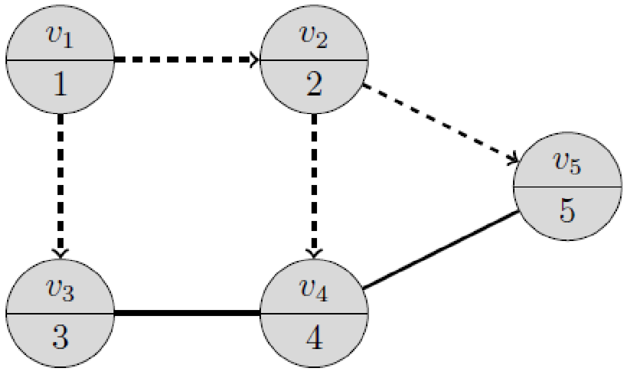


4. Vertex v1 has no more descendants, it is removed from the queue. The queue is now Q = [v2, v3].

5. We visit the first descendant of v2. Q = [v2, v3, v4]. Mark v4 as the 4th visited vertex. Make edge (v2, v4) a tree edge.

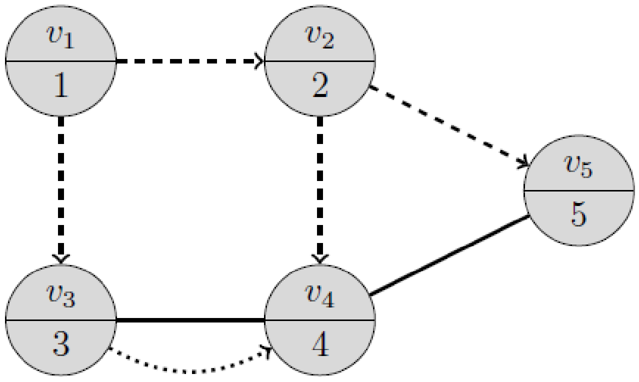


6. We visit the second descendant of v2. Q = [v2, v3, v4, v5]. Mark v5 as the 5th visited vertex. Make edge (v2, v5) a tree edge.

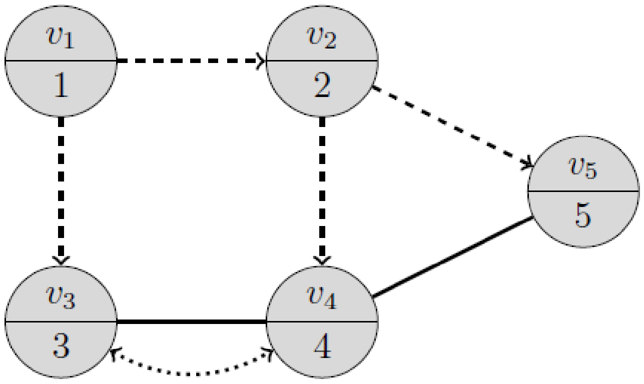


7. Vertex v2 has no more descendants, it is removed from the queue. The queue is now Q = [v3, v4, v5].

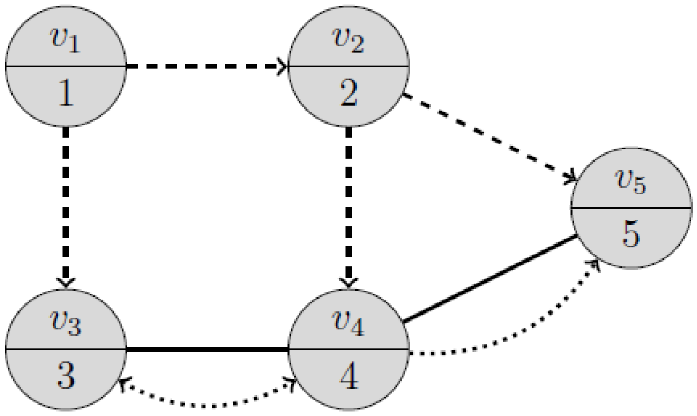
8. Vertex v3's only descendant is v4 which has already been visited. This means (v3, v4) is a cross edge. Vertex v3 is removed from the queue so Q = [v4, v5].



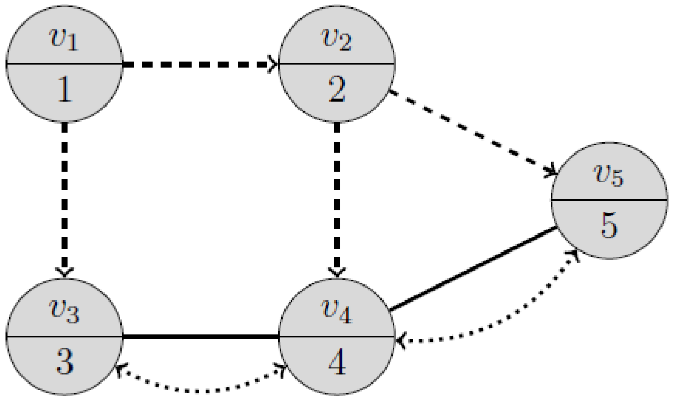
9. Vertex v4’s first descendant is v3 which has already been visited this means (v4, v3) is a cross edge.



10. Vertex v4’s second descendant is v5 which has already been visited. This means (v4, v5) is a cross edge. Since, there are no additional descendants v4 is removed from the queue. This results in Q = [v5].



11. Vertex v5’s only descendant is v4 which has already been visited. This means (v5, v4) is a cross edge. V5 is removed from the queue so Q = [ ] and our BFS has finished since there are no unvisited vertices in G.



What is the work done by the BFS algorithm in the worst case? We can identify the crucial unit of work to be considering whether or not to enqueue a vertex.
In BFS(), after marking each vertex as unvisited, the algorithm calls BFSVisit() at most once for each vertex (lines 13-15) and in BSFVisit, the vertex is added to the queue (line 8). So, this bound is $|V|$, the number of vertices in the graph.
In BFSVisit(), lines 9-12 examine every edge from this vertex to decide whether or not to enqueue each of its neighbors. Since each edge appears twice in the graph's adjacency structure, this bound is $2|E|$ or twice the number of edges in the graph.
Therefore, the number of "enqueings" that are considered is $\leq |V| + 2|E| < 2(|V| + |E|)$, so the BFS algorithm's asymptotic class is $O(|V| + |E|)$.

As we remarked with the DFS algorithm, if the BFS() algorithm finishes, without visiting all vertices in the graph, we restart it from one of the unvisited vertices. This forms a BFS forest of separate subgraphs.

There are many interesting properties we can derive from the run of the BFS algorithm on a graph. Three of these are cycles, connectivity, shortest distance.

It is easy to perform cycle detection by looking for cross-edges (instead of back edges as in DFS).
Giving the actual 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. Once BFSVisit is done running, check to see if all vertices have been visited. If not, the graph is not connected. The number of times it needs to be restarted is the number of connected components.

In an unweighted graph, the BFS tree provides us with the shortest distance between any two vertices. For any two vertices, the path from one vertex to the other using only tree edges is the shortest path between those two vertices, and the length of this path (number of edges in the path) is the shortest distance between those two vertices.

B. Mathematical preliminaries – k-combinations, a review of some summation formulas, distance between two points

B1. k-combinations

For any integer $n \geq 2$, the number of ways to choose a subset of 2 objects from a collection of n distinct object if we do not care about the order in which the objects were chosen is

$\binom{n}{2} = nC2 = \frac{n(n-1)}{2}$ which is read as "n choose 2".

example: If you wanted to list all subsets of 2 elements that can be made from the set {A, B, C, D, E}, your list of subsets would be the 10 subsets:

{A,B}, {A,C}, {A,D}, {A,E}, {B,C}, {B,D}, {B,E}, {C,D}, {C,E}, {D,E}

Rather than creating all 10 subsets, you could reason that you have 5 options for which object to take first, then 4 options for which (different) object to select second, but the same two objects selected in the opposite order would produce the same result. [We would not list {B,D} and {D,B} since those are the same subset. Recall that order does not matter in a set.] So, the number of subsets of size 2 that can be created from a master set of n = 5 distinct objects is

$$\binom{5}{2} = 5C2 = \frac{5(5-1)}{2} = \frac{5(4)}{2} = 10$$

This formula is one example of a concept from combinatorics (the branch of mathematics that deals with counting techniques) known as combinations. In general, the number of combinations (read: subsets) of k objects that can be selected from a set of n distinct objects without replacement (read: without taking the same object more than once) when the order in which the objects are selected does not matter, is given by the computation

$$\binom{n}{k} = nCk = \frac{n \cdot (n-1) \cdot (n-2) \cdots (n-k+1)}{k \cdot (k-1) \cdot (k-2) \cdots 3 \cdot 2 \cdot 1}$$

We read this as "n choose k" and call the outcomes k-combinations. In the fraction, there are k terms in the numerator and k terms in the denominator. When you do the arithmetic, every term in the denominator will cancel with a term in the numerator, leaving an integer as the answer.

For example,

$$\binom{10}{4} = 10C4 = \frac{10 \cdot 9 \cdot 8 \cdot 7}{4 \cdot 3 \cdot 2 \cdot 1} = 210$$

meaning there are 210 ways to select a subset of 4 objects from a master set of 10 objects.

The formula can also be written using factorials as

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!}$$

so that

$$\binom{10}{4} = \frac{10!}{4!\cdot 6!} = \frac{10\cdot 9\cdot 8\cdot 7\cdot 6\cdot 5\cdot 4\cdot 3\cdot 2\cdot 1}{4\cdot 3\cdot 2\cdot 1\cdot 6\cdot 5\cdot 4\cdot 3\cdot 2\cdot 1} = 210$$

Notice that by cancelling the last 6 terms in the numerator with the last 6 terms in the denominator, this version of the formula agrees with the previous version. [If we cared about the order in which the objects were selected, we would be counting permutations rather than combinations.]

B2.More summation formulas

In today's class, we will use several summations formulas that we have already seen earlier in the semester (see notes for class 2). Three that will occur numerous times are:

a) If the general term of a summation is a sum of terms, we can create separate summations for each term.

$$\sum (a_i + b_i) = \sum a_i + \sum b_i$$

ex.

$$\sum_{j=1}^{n} (j^3 + 5j) = \sum_{j=1}^{n} j^3 + \sum_{j=1}^{n} 5j$$

b) If every term of a summation includes the same constant, we can factor it out in front of the summation.

$$\sum c\cdot a_i = c\sum a_i$$

ex.

$$\sum_{j=1}^{n} 5j^2 = 5\sum_{j=1}^{n} j^2$$

c) Sums of consecutive positive integers:

$$\sum_{i=1}^{n} i = 1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}$$

ex.

$$\sum_{i=1}^{10} i = 1 + 2 + 3 + \cdots + 10 = \frac{10(11)}{2} = 55$$

B3. Formula for the distance between two points in 2-dimensions.

For two points $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$ in the x-y (Cartesian) plane, the distance between the two points is given by the Euclidean metric

$$d(p_1, p_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

example: The distance between the points $p_1 = (14, 50)$ and $p_2 = (17, 46)$ is

$$\sqrt{(14 - 17)^2 + (50 - 46)^2} = \sqrt{(-3)^2 + (4)^2} = \sqrt{9 + 16} = \sqrt{25} = 5$$

C. Data structures preliminaries – k-combinations, a review of some summation formulas, distance between two points

C1. Linked Lists, Stacks, and Queues

A list is a finite sequence of data items arranged in a linear order.
A **linked list** is a sequence of nodes that contain

data

pointer or pointers to other nodes in the list. If the pointer does not point to a node it is **nil**.

There are two types of lists:
Singly Linked: The one pointer in each node points to its successor in the list.
Doubly Linked: The two pointers in a node point to both the successor and the predecessor in the list.

The first node of the list is a special node called the **head**.
We may also optionally keep track of the **tail** of the list using a special tail node.

Think: How do you search for an item in a linked list?

Two data structures that are often implemented as linked lists are stacks and queues.

A **stack** is a last-in-first-out (LIFO) structure where insertions and deletions both occur at the head/top of the stack. A computer's operating system will use a stack to manage multiple recursive calls to the same function that are waiting to be processed.

A **queue** is a first-in-first-out (FIFO) structure where insertions are made at the tail of the queue and deletions are made at the head of the queue. This is similar to a line of cars at the Dunkin's drive-up window – new arrivals join at the end of the line and wait until they get to the front of the line to be served.

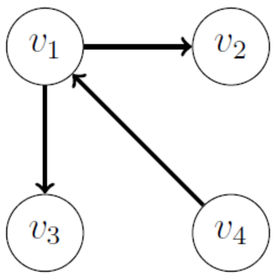
C2. Representing Graphs as Adjanceny Lists

We saw in last class that a graph G = (V; E) is a pair of two sets where,
V is the set of vertices (nodes) in the graph.
E is the set of edges in a graph.
Every edge in the edge set is of the form (u, v) where u and v are in V.

We represented a graph by a matrix with one row and column for each vertex and a 1 in row j column k if there is an edge from j to k, or a 0 if edge (j, k) does not exist.

A second way to represent a graph is to create an Adjacency List: An array of linked lists, one linked list for every vertex in the graph. Each element of the array represents the list of vertices adjacent to the given vertex. If vertex v has k neighbors (v is adjacent to k other vertices), then there are k vertices in the linked list associated with vertex v and there are k 1s in row v of the adjacency matrix.

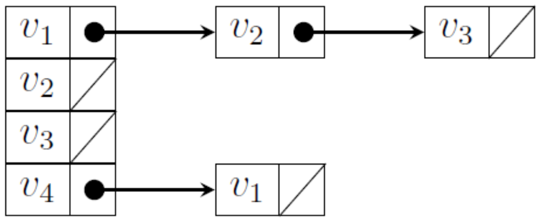
Example: For the graph



the adjacency matrix would be

$$\begin{array}{c c} & \begin{array}{cccc} v_1 & v_2 & v_3 & v_4 \end{array} \\ \begin{array}{c} v_1 \\ v_2 \\ v_3 \\ v_4 \end{array} & \begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

and the adjacency lists would be



Note: This does NOT say that v2 points to v3. It says that there are edges from v1 to both v2 and v3. The diagonal line in the pointer field indicates that there are no (or no more) elements in this linked list.