# Graph Theory

## Concepts

|  |  |
| --- | --- |
| **1.1** | **Tree Structure of Graph Algorithms**  Graph  |-- Unweighted  | |-- Undirected  | | |-- Articulation Point  | |-- Directed  | | |-- SCC (Strongly Connected Components  | | |-- Topological Sort  |-- Weighted  |-- MST (Minimum Spanning Tree)  | |-- Prim’s |
| **1.2** | **BFS**  **General Information**   * Uses queue. * Gives the shortest path of all the nodes from the source node. * Optimal for *unweighted* graph. * Complexity:  |  |  |  | | --- | --- | --- | |  | In Adjacency List | In Adjacency Matrix | | Time Complexity | O(|V| + |E|) | O(|V|2) | | Space Complexity | O(|V| + |E|) | O(|V|2) |   **Applications**   * + Finding all connected components in a graph   + Finding the shortest path between two nodes *u* and *v*   + Testing a graph for bipartiteness   **Finding the shortest path between the source and another node**  Just run BFS.  **Finding the most distant node from the source node**  The lastly queued node is the most distant node from the source node.  **Detecting *any* cycle within an undirected graph**  Existence of an edge from a WHITE or GRAY node to a BLACK node.  **Testing a graph for bipartiteness**  While coloring the nodes with two colors, if we find that the adjacent’s color is already the same as the color of the current node, then as the color of the current node, then as the color of the current node, then the graph is *not* bipartite.  Every tree is bipartite.  Cycle graphs with an even number of edges are bipartite. |
| **1.3** | **DFS**  **General Information**   * Uses stack (recursive function, to be precise). * Produces a spanning tree from a graph. * Time complexity: O(|V| + |E|) [for adjacency list], O(|V|2) [For adjacency matrix]. * Space complexity: O (h), where *h* = length of the longest simple path in the graph.   **Applications**   * + Finding connected components   + Finding strongly connected components   + Topological sorting   + Edge detection   + Finding all-pair paths between source and destination nodes   + Finding articulation points.   + Solving puzzles with only one solution, such as mazes   **Edge detection**  Let, (*u*, *v*) is an edge.   * If (color[v] == WHITE), then (*u*, *v*) is a *tree* edge. * If (color[v] == GREY), then (*u*, *v*) is a *back* edge. * If (color[v] == BLACK), then (*u*, *v*) is a *forward* or *cross* edge.   + - If (start\_time[u] < start\_time[v]), then (*u*, *v*) is a *forward* edge.     - If (start\_time[u] > start\_time[v]), then (*u*, *v*) is a *cross* edge.   If the graph is *undirected*, then all of its edges are either tree edges or back edges.  **Detecting a cycle within a directed graph**  Existence of a back edge.  **Finding connected components**  In case of directed graph, convert it to undirected graph by just adding reversed edges to all the edges. Now, run DFS (or BFS) from any node. After the traversing finishes, check whether there is any node marked as WHITE. If no such node can be found, then the graph is connected.  **Topological sorting**  Sort the nodes in *non-increasing* order according to their *finishing time*. Another way is to sort the nodes in non-decreasing order of their *in-degrees*. However, some tweaks are needed to apply this algorithm.  **Unique topological sort**  While doing topological sort *using in-degrees*, if more than one lowest value can be found, then the topological sort is *not* unique.  **Finding all the paths between source and destination nodes**  Color the nodes as WHITE instead of BLACK when returning from the recursive functions.  **Finding Articulation Points**  A parent node, *v* will be an articulation point if, for any node *w* connected with *v* as a *back edge* or *tree edge*, *low(w) >= d[v]*.  **Finding Bridges**  Any edge in a graph that does not lie on a cycle is a bridge.  **Finding strongly connected components**  Run DFS and record the finishing times. Reverse all the edges. Reset all the colors. Run DFS again from the node with the *highest* finishing time. While running the second DFS, output the nodes in the DFS tree. These nodes comprise an SCC. If there are still unvisited (i.e., white colored) nodes, then run DFS again from the white node whose finishing time is highest. |

**BFS, DFS, MST, Dijkstra, Bellman-Ford, Warshall**

**Theory Questions**

|  |  |  |
| --- | --- | --- |
| ✯✯✯ | **1.1** | For each of the following statements, say whether it is *True* or *False*. If it is *True*, give a brief explanation; if it is *False*, give a simple counter example. *[In-course 1, unknown year. Marks: 12]*  1. **In a depth first search of a directed graph, if there are no back edges, then there are no cycles.** 2. **Adding one edge to a DAG always creates cycle.** 3. **If we add a directed edge to a directed graph with *S* strongly connected components, the number of strongly connected components in the new graph can equal any number between *1* & *S* but cannot exceed *S*.** 4. **Suppose all edge weights are different. Then the shortest path from *A* to *B* is unique.** 5. **True.**   By running DFS in a *directed* graph, we get a tree along with forward, back or cross edges. As the graph is *directed*, only back edges would form cycles. Hence, if there are no back edges, then there are no cycles.   1. **False.**   Counter example:   1. **True.**   In the new graph, the number of SCCs might decrease due to the added edge. And the lowest number of SCCs is 1. However, the graph might be unaffected due to the added *directed* edge. So, the number of SCCs won’t exceed S. Here’s an example:   1. **False.**   1  2  3  Counter example:  **Original DAG**  **Forms Cycle**  **Doesn’t Form Cycle**  Original graph  Decreased  No effect  (A, B and C are Strongly Connected Components) |
| ✯✯ | **1.2** | How can a cross edge be detected in a directed graph when DFS is used? *[In-course 1, unknown year. Marks: 3]* Let, (*u*, *v*) is an edge. If (color[v] == BLACK) and if (discovery\_time[u] > discovery\_time[v]), then (*u*, *v*) is a *cross* edge. |
| ✯✯✯ | **1.3** | How can we detect a vertex as an articulation point when we run DFS algorithm? *[In-course 1, unknown year. Marks: 3]* If low(w) >= discovery\_time[v], where *w* is a child of *v*, then *v* is an articulation point. However, the root node should be handled separately. If, after running DFS through one of root’s edges, we find that there are still unvisited nodes left, then the root is an articulation point; otherwise, it isn’t. |
| ✯✯ | **1.4** | Answer *True* or *False* with proper justification. *[In-course 2, unknown year. Marks: 2.5 × 2]*  1. **Bellman-Ford shortest path algorithm is an example of greedy algorithm.** 2. **Consider a graph G = (V, E), with negative weight edges. We can solve all pairs shortest paths (with negative cycle detection) on G in O(|V|3) time.** 3. **False.**   In Bellman-Ford, we can take any edge at a time. We do not opt for the shortest cost edge. So, this algorithm is not a greedy algorithm.   1. **True.**   We can solve all pairs shortest paths using Warshall algorithm, whose time complexity is O(|V|3). To detect negative cycles using Warshall, we need to search the diagonal positions of the path matrix for a negative value. This would take O(|V|) time. So, our total complexity becomes O(|V|3 + |V|). If V is very large, then we can ignore |V| comparing to |V|3. Therefore, the total time complexity becomes O(|V|3). |
|  | **1.5** | How many edges are there in an *n*-vertex complete graph? |
| ✯ | **1.6** | What is the property of bipartite graph? How can we detect it? The property of bipartite graph is that its vertices can be divided into two disjoint sets *U* and *V* such that every edge connects a vertex in *U* to one in *V*.  First, we should color the source node with a particular color – say BLUE. Then, we have to color all the adjacent nodes with another color – say RED. Similarly, we’ll color all the adjacent nodes of RED nodes as BLUE and all the adjacent nodes of BLUE nodes as RED. While performing this, if we encounter any node which already has a similar color to his adjacent node, then we can conclude that the graph is *not* bipartite. Else, after finishing running the BFS, we can conclude that the graph *is* bipartite. |
|  | **1.7** | For a graph, the total number of in-degree plus the total number of out-degree is even. Why? In a graph, if there are *n* in-degrees, then there are definitely *n* out-degrees. So, in-degree + out-degree = 2*n*, which is an even number. |
| ✯ | **1.8** | Suppose ADJ is an N × N matrix. What will be the result if we create another matrix ADJ2 where ADJ2 = ADJ × ADJ? We’ll get a matrix of (N + N) or 2N length. That is, we’ll get a matrix where the path length of each node is 2. |
|  | **1.9** | Can tree edges form cycles? Why or why not? No. Only tree edges imply we have a tree or a forest, which, by definition, is acyclic. To produce cycles, we need back edges. |
| ✯ | **1.10** | Prove that an undirected graph is *acyclic* if and only if a DFS yields no back edges. After running DFS, if no back edges can be found, then the graph has only tree edges. (As the graph is *undirected*, therefore, there will be no forward/cross edges but only tree and back edges.) Only tree edges imply we have a tree or a forest, which, by definition, is acyclic. |
| ✯ | **1.11** | Proof that if a graph, G is undirected, a DFS produces only tree and back edges. Let’s assume there is a forward edge *f* in *fig. (a)*. But when we’ll traverse from A to C, we’ll find that *f* is actually a back edge. So, there is no forward edge.  *f*  *c*  Source  Source  (a)  (b)  Again, let’s assume there is a cross edge *c* in *fig. (b)*. But if we start traversing from A, we’ll find that after (B, C), we encounter (C, D) as a tree edge and then (D, B) as a back edge. So, the figure is actually wrong and hence, there is no cross edge.  Therefore, in an undirected graph, a DFS produces only tree and back edges. |
|  | **1.12** | What is the maximum number of spanning trees that can be generated from a graph? NN-2, where N is the number of vertices. |
|  | **1.13** | Are shortest paths well-defined in a DAG? Yes. Because, (negative) cycles does not exist in a DAG. |
|  | **1.14** | Find the shortest path between *A* and *B*. When will it fail? *Run BFS*. It will fail if the edges have weights assigned. |
| ✯✯ | **1.14** | **What is bridge? How can it be detected?**  A bridge is an edge deleting which causes the graph to become disconnected.  An edge will be a bridge if:   1. It connects two articulation points. 2. It connects an articulation point with such a node which has no other edges than this one. |
|  | **1.15** | **How can we determine that a graph has a *unique* topological sort?**  While doing topological sort using in-degree approach, if more than one lowest in-degree value can be found, then it can be said that the graph does *not* have a unique topological sort. |
|  | **1.16** | **Can Dijkstra be used for finding *maximum*-cost path?**  Source  1  3  2  No. Consider the graph in the figure. As Dijkstra uses greedy approach, following the algorithm, we’ll be taking the edge (A, C). But actually, the maximum-cost path is (A, B, C). |

**Critical Questions**

|  |  |  |
| --- | --- | --- |
| ✯✯✯ | **1.1** | **The diameter of a tree T = (V, E) is given by , where is the shortest path between *u* and *v*. That is, the diameter is the largest of all shortest-path distances in the tree. Give an efficient algorithm to compute the diameter of a tree, and analyze the running time of your algorithm. *[2005. Marks: 5 + 3; Cormen exercise 22.2-7)]***  *Hints: Run BFS from any node. Now, run another BFS from the* ***longest*** *node. Now we can find the longest path between this source and the lastly queued node, which is the diameter.* |
|  | **1.2** | **What is in-degree and out-degree of a vertex? Give an algorithm to calculate the out-degree of all vertices of a graph where the graph is given as adjacency matrix. Determine the time complexity of your program. *[In-course 1, unknown year. Marks: 4]***  *Hints: run two for-loops to cover up the whole adjacency matrix. The time complexity should be O(|V|2).* |
|  | **1.3** | **Give a complete idea to solve topological sort (or, strongly connected components) with the help of Floyd-Warshall’s all pairs shortest paths algorithm. *[In-course 1, unknown year. Marks: 5]*** |
|  | **1.4** | **Consider a directed graph in which the nodes are weighted (with positive weights), while the edges aren’t. The weight of a path is now defined as the sum of the weights of the nodes that it uses. Given source and destination nodes, design algorithm that will determine the shortest path between them. *[In-course 1, unknown year. Marks: 5]***  *Hints: Same as Dijkstra’s algorithm. But initially, we’ll insert the weight of the nodes into the cost array instead of initializing the array to infinite. Also, no need to place the cost of source node as 0.* |
|  | **1.5** | **Consider the following matrix which corresponds to the initialized distance matrix of the all-pairs-shortest path algorithm:**  **Draw the corresponding graph and execute one iteration of Floyd-Warshall algorithm. *[In-course 1, unknown year. Marks: 5]*** |
|  | **1.6** | **Suppose you are give the following graph where edges are stored in ascending order (according to their cost). Suppose 1 is the starting vertex. Perform one iteration of Bellman-Ford algorithm and show *d* and *П* values. *[In-course 1, unknown year. Marks: 5]***  5  -3  8  0  5  -1  8  1  1 |
|  | **1.7** | **Give an efficient algorithm (preferably linear time) algorithm that takes as input a directed acyclic graph G = (V, E) and two vertices s and t and returns the number of paths s to t in G. (Your algorithm only needs to count the paths, not list them.) *[In-course 1, unknown year. Marks: 7]***  *Hints: Use all-pair paths using DFS (i.e., set color[v] = WHITE instead of BLACK].* |
|  | **1.8** | **A directed graph G = (V, E) is said to be *semi-connected* if for all pairs of vertices *u*, *v* V, we have *u* ↝ *v*, or *v* ↝ *u*. Give an efficient (preferably O(V+E)) algorithm to determine whether or not G is semi-connected. (Hint: You can think about strongly connected component’s idea to solve this problem). *[In-course 1, unknown year. Marks: 7]*** |
|  | **1.9** | **Apply Bellman-Ford to the graph below, using *vertex 1* as the source vertex and perform *first 3 iterations*. Show the full contents of the distance array d[] and predecessor array π[] after each iteration. (The edges are stored as the weights in ascending order. If the weights of (*u1*,*v1*) and (*u2*,*v2*) are equal then (*u1*,*v1*) will be stored earlier if *u1* < *u2*). *[In-course 1, unknown year. Marks: 8]***  2  4  9  3  5  10  1  4  10  2 |
|  | **1.10** | **Suppose that, in the single-source shortest path problem, we wish to find not just any shortest path, but among those the shortest path that has the fewest hops. There are at least two way to do this: (a) Modify slightly Dijkstra’s algorithm by adding an array…….., or, (b) Apply the original Dijkstra algorithm but with the edge weights changed slightly to reflect our new interest in few hops. Describe briefly one of these ways (or, if you prefer, your own new idea for an algorithm for this problem). *[In-course 1, unknown year. Marks: 7]*** |
|  | **1.11** | **There are two types of professional wrestlers – good guys and bad guys. Between any pair of professional wrestlers, there may or may not be a rivalry. Suppose we have *n* professional wrestlers and we have a list of *r* pairs of wrestlers for which there are rivalries. Give an *O(n + r)*-time algorithm that determines whether it is possible to designate some of the wrestlers as good guys and the remainder as bad guys such that each rivalry is between a good guy and a bad guy. If it is possible to perform such a designation, your algorithm should produce it. *[Cormen exercise, 22.2 – 6]***  If the graph is bipartite, then it is possible. Otherwise, not. |
|  | **1.12** | **Find the minimum spanning tree of the following graph using prim’s algorithm. *[2004. Marks: 8]***  14  4  11  8  2  4  2  7  6  8  7  1  9  10 |

**Recurrence Relation, Backtracking, Branch and Bound**

**Theory Questions**

|  |  |
| --- | --- |
| **1.1** | **Find the asymptotic bound for T(n) in each of the following recurrences: *[2006, Marks: 3]***   1. **T(n) = 49T(n / 7) + n** 2. **T(n) = 5T(n – 1) – 6(n – 2)** |
| **1.2** | **Write down the recurrence relation for the following program: *[2006, Marks: 3]***  **void my\_function(int n) {**  **int i, j;** |