**KEYWORDS A**

* Data Center - A data center is a physical location that stores computing machines and their related hardware equipment. It contains the computing infrastructure that IT systems require, such as servers, data storage drives, and network equipment
* Energy Management - Energy management is the proactive and systematic monitoring, control, and optimization of an organization’s energy consumption to conserve use and decrease energy costs.
* Servers - a computer or computer program which manages access to a centralized resource or service in a network.
* Time Interval - the amount of time between two given times.
* Overlapping - partly coincide in time.
* Scheduling Tasks - arrange or plan a task to take place at a particular time.
* Graphs (Chordal) - A chordal graph is a type of graph where every cycle with four or more vertices has a "chord," which is an edge connecting two non-adjacent vertices in the cycle.
* NP-Complete Problem - If a problem is NP and all other NP problems are polynomial-time reducible to it, the problem is NP-complete.
* Chromatic Number - the minimal number of colours needed to colour the vertices in such a way that no two adjacent vertices have the same colour.
* Gluttonous Algorithm - a class of algorithms that make locally optimal choices at each step with the hope of finding a global optimum solution.

**NOTES A**

*Graph*

* A graph is a set of points (vertices) together with a collection of lines (edges) connecting some of the points. The set of vertices must not be empty.
* A graph may have multiple edges, i.e. more than one edge between some pair of vertices, or loops, i.e. edges from a vertex to itself.
* A graph without multiple edges or loops is called simple. Many natural problems only make sense in the setting of simple graphs.
* If two vertices are joined by an edge they are called adjacent.
* The degree of a vertex v, written d(v), is the number of ends of edges which connect to that vertex.
* Empty graphs - En has n vertices and no edges.
* Complete graphs - Kn has n vertices and each vertex is connected to each other vertex by precisely one edge.
* Regular graphs. A regular graph is one in which all the vertices have the same degree
* A walk in a graph is a sequence of the form v1, e1, v2, e2 . . . vr, for some r >= 1, where the vi are vertices and ei is an edge from vi to vi+1 for each 1 <= i < r.
* A trail is a walk in which no edge appears more than once.
* A path is a walk in which no vertex appears more than once.
* A graph G is connected if and only if there is a path between every pair of vertices.

*Trees*

* A tree is a connected graph with no cycles.
* A forest is a graph with no cycles.
* Let G be a connected graph with n vertices. Then G is a tree if and only if e(G) = n − 1.
* Let G be a connected graph on n vertices. A spanning tree for G is a subgraph which is a tree containing all vertices of G. To identify a spanning tree:
  + if there are only n − 1 edges remaining then we have found a tree, so stop;
  + the graph is connected and not a tree, so find a cycle;
  + delete any edge from this cycle;
  + go to step 1.

*Kruskal’s algorithm*

* A weighted graph is a graph where each edge has a positive number (or “weight”) associated with it.
* Kruskal’s algorithm finds a minimum-weight spanning tree in a weighted graph.
* Algorithm:
  + list the edges in increasing order of length;
  + consider the smallest remaining edge;
  + if we can add that edge to T without creating a cycle, do so, otherwise discard it;
  + if we have n − 1 edges, T is a spanning tree so stop;
  + otherwise go back to step 2.

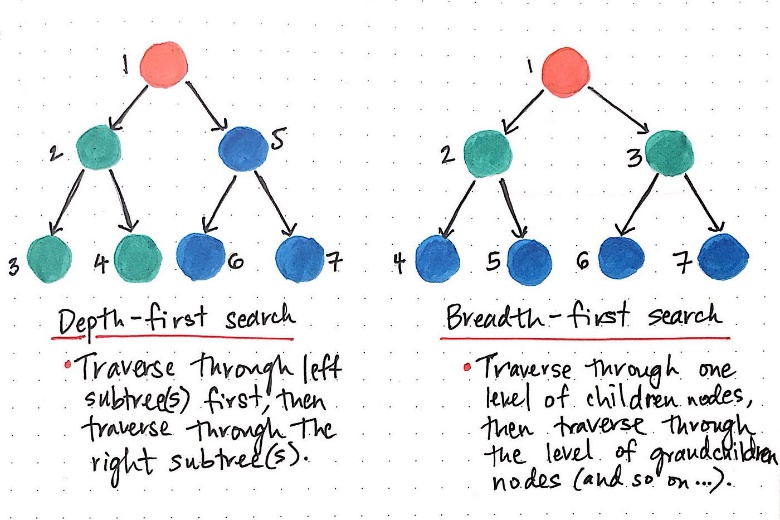
*Eulerian Graphs*

* A graph is Eulerian if it has a closed trail which includes every edge.
* A graph is semi-Eulerian if has a trail which is not closed but which includes every edge.
* Euler trail - a closed trail which includes every edge
* A connected graph G is Eulerian if and only if every vertex has even degree.
* A connected graph G is semi-Eulerian if and only if exactly two vertices have odd degree.
* A bridge in a connected graph is an edge whose removal will disconnect the graph.
* Fleury’s algorithm proceeds as follows:
  + If the graph is Eulerian, start at any vertex and move along any edge, deleting that edge once you have crossed it, but only crossing a bridge if there is no alternative.
  + If it is semi-Eulerian, start at either vertex of odd degree and then apply the same algorithm.

*Hamilton Cycles*

* A Hamilton cycle in a graph is a cycle which includes every vertex
* Hamilton path is a path which includes every vertex.
* A graph is Hamiltonian if it has a Hamilton cycle, and semi-Hamiltonian if it is not Hamiltonian but does have a Hamilton path.
* A directed graph, or digraph, is like a graph, but instead of a set of edges between pairs of vertices we have a set of arcs, where each arc goes from one vertex to another (or back to itself).

*Depth First Search and Breadth First Search*



*Dijkstra’s Algorithm*

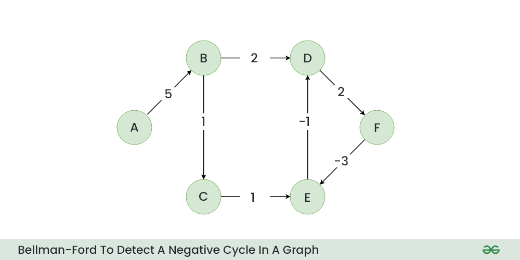
Dijkstra’s algorithm finds the shortest distance to every vertex.

* Start by marking the start vertex as distance 0, and circle it.
* For each vertex adjacent to the start vertex, we calculate a bound which is the length of the edge to that vertex.
* Find the vertex with the smallest bound among vertices which do not have final answers circled. Circle that bound, and mark the edge which gave that bound.
* For each edge leaving the vertex you have just marked with a final answer, add that answer to the length of an edge to get a bound for the vertex that edge goes to. If it is smaller than the current bound at that vertex, replace the old bound with this one.
* Repeat steps 3 and 4 until all vertices have their exact distances marked

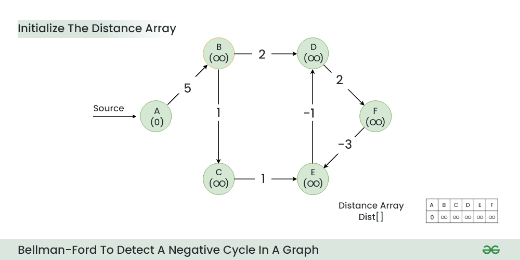
[*Bellman Ford Algorithm*](https://www.geeksforgeeks.org/bellman-ford-algorithm-dp-23/)

Bellman-Ford is a single source shortest path algorithm that determines the shortest path between a given source vertex and every other vertex in a graph. This algorithm can be used on both weighted and unweighted graphs.

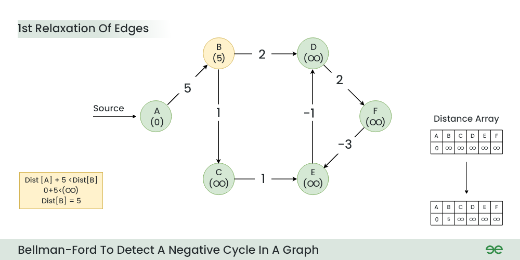
* Principle of Relaxation of Edges for Bellman-Ford:
  + It states that for the graph having N vertices, all the edges should be relaxed N-1 times to compute the single source shortest path.
  + In order to detect whether a negative cycle exists or not, relax all the edge one more time and if the shortest distance for any node reduces then we can say that a negative cycle exists. In short if we relax the edges N times, and there is any change in the shortest distance of any node between the N-1th and Nth relaxation than a negative cycle exists, otherwise not exist.
* Working of Bellman-Ford Algorithm to Detect the Negative cycle in the graph:



* + Step 1: Initialize a distance array Dist[] to store the shortest distance for each vertex from the source vertex. Initially distance of source will be 0 and Distance of other vertices will be INFINITY.



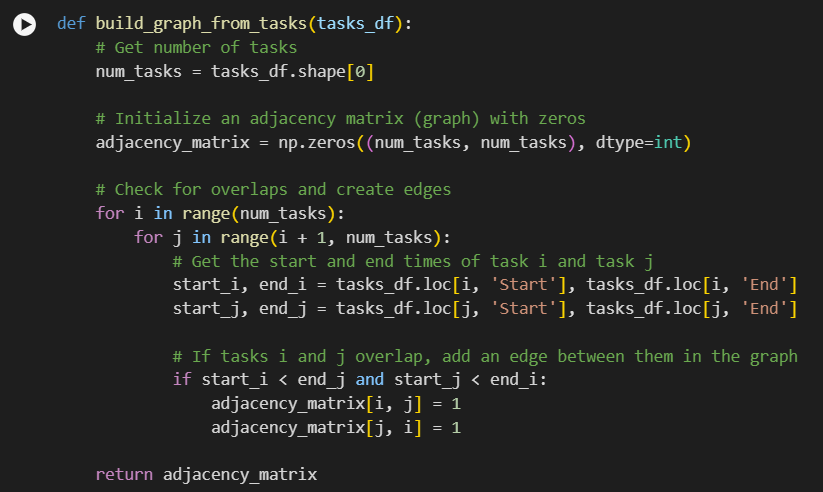
* + Step 2: Start relaxing the edges, during 1st Relaxation:
    - Current Distance of B > (Distance of A) + (Weight of A to B) i.e. Infinity > 0 + 5
    - Therefore, Dist[B] = 5



*Vertex Colouring*

* A vertex colouring of G is an assignment of a colour to each vertex such that every edge goes between two different colours.
* The chromatic number of G, written χ(G), is the smallest k such that G is k-colourable.
* Greedy Algorithm:
  + Let G be a simple graph. Choose an ordering of the vertices of G.
  + We will colour the vertices one at a time in order with colours from the set {1, 2, 3, . . .}.
  + When colouring each vertex, we choose the smallest colour which has not already been assigned to one of its neighbours.
  + It will always produce a colouring, but the number of colours used will depend on what order we colour the vertices in.
  + There is always some order for which the greedy algorithm uses only χ(G) colours.
  + The worst-case performance of the greedy algorithm can be: for each n there is a bipartite graph on 2n vertices and an ordering for which greedy uses n + 1 colours.

**CODE (+EXPLANATION)** A



The build\_graph\_from\_tasks function takes a Pandas DataFrame tasks\_df as input and returns an adjacency matrix representing a graph, where each task is a node, and two nodes are connected by an edge if the corresponding tasks overlap in time.

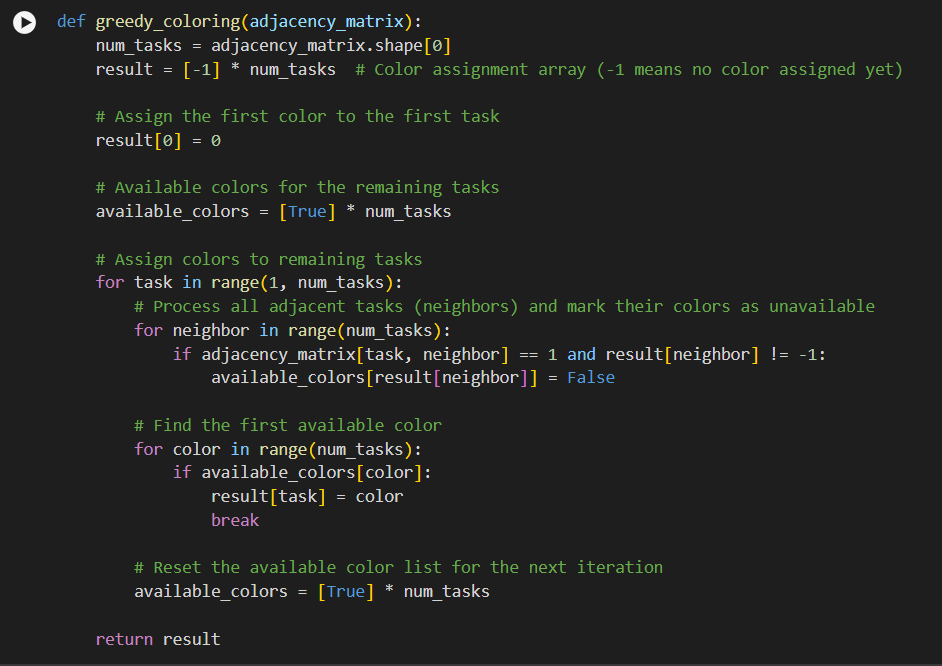
The function iterates over all pairs of tasks using two nested loops:

* The outer loop iterates over each task i from 0 to num\_tasks-1.
* The inner loop iterates over each task j from i+1 to num\_tasks-1. This is done to avoid duplicate edges and to ensure that the graph is undirected (i.e., if there's an edge between tasks i and j, there's also an edge between tasks j and i).

For each pair of tasks i and j, the function:

* Extracts start and end times: It extracts the start and end times of tasks i and j from the input DataFrame tasks\_df.
* Checks for overlap: It checks if tasks i and j overlap in time by checking if the start time of task i is less than the end time of task j and the start time of task j is less than the end time of task i. If they overlap, it adds an edge between them in the graph by setting the corresponding elements in the adjacency matrix to 1.

Finally, the function returns the constructed adjacency matrix, which represents the graph of overlapping tasks.



The greedy\_coloring function takes an adjacency\_matrix as input and returns a color assignment array, where each task is assigned a color such that no two adjacent tasks have the same color. This is a classic problem known as the Graph Coloring Problem.

The function assigns the first color (0) to the first task.

The function then iterates over the remaining tasks (from 1 to num\_tasks-1). For each task, it:

* Processes adjacent tasks: It iterates over all tasks and checks if the current task is adjacent to any task that has already been assigned a color. If so, it marks the color of the adjacent task as unavailable (available\_colors[result[neighbor]] = False).
* Finds the first available color: It iterates over the available colors and assigns the first available color to the current task (result[task] = color).
* Resets available colors: It resets the available\_colors array for the next iteration.

Finally, the function returns the result array, which contains the color assignment for each task.