DAA PROJECT: GRAPH LIBRARY IN C

Raunak Sengupta PES1201700072 4-G 2018-2019

Abstract

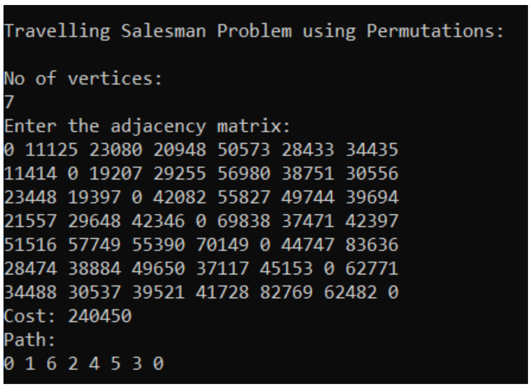
The Graph is a rather abstract data structure, as it could be modelled into anything, the possibilities are limitless. Due to this nature, it presents itself to be quite a challenge to work with, leading to it still being a vital area of research in Computer Science. Though the algorithms that have been devised to extract meaningful information out of graphs are not few, there still exists no concrete Library of their implementations in any general-purpose programming language.

Graphs are turning out to be predominant in today's world. Their applications are copious: Google Maps' Algorithm, Social Media Friends Suggestion Algorithms, Process-Resource Allocation in Operating Systems, Data Organisation, Molecular Research, Mathematical Applications in Geometry and Topology, Sociology, and many more that are still cropping up. Therefore, the implementations of fundamental algorithms are turning out to be extremely necessary. That is the main issue that this project aims to combat.

The Implementation Language chosen is C.

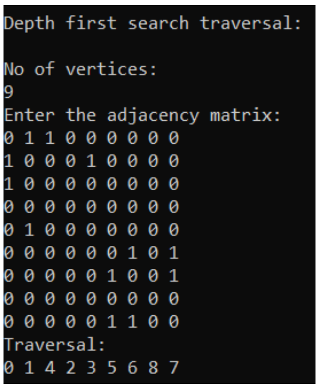
Implementations and Results

- **Travelling Salesman Problem using Permutations**
  The Input is an Undirected Graph with the vertices representing Cities and the edge weights corresponding to the Cost to travel between the edge's terminals. The output is the Minimum Total Cost and Path to traverse a Hamiltonian Circuit, ie, travel through each of the cities starting and ending with a particular city.
  The results are:

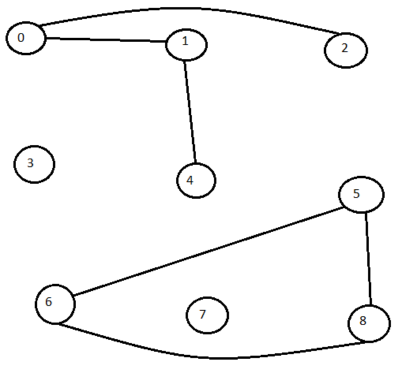


```
Travelling Salesman Problem using Permutations:

No of vertices:
7
Enter the adjacency matrix:
0 11125 23080 20948 50573 28433 34435
11414 0 19207 29255 56980 38751 30556
23448 19397 0 42082 55827 49744 39694
21557 29648 42346 0 69838 37471 42397
51516 57749 55390 70149 0 44747 83636
28474 38884 49650 37117 45153 0 62771
34488 30537 39521 41728 82769 62482 0
Cost: 240450
Path:
0 1 6 2 4 5 3 0
```

- **Depth First Search Traversal**
  The Input is a Graph. The Output is the order in which its vertices are visited when following the Depth First Search Strategy. It involves visiting the vertices adjacent to one another, going to the deepest point where there are no other vertices to be visited, then visiting other branches along the path in the same manner.
  The results are:

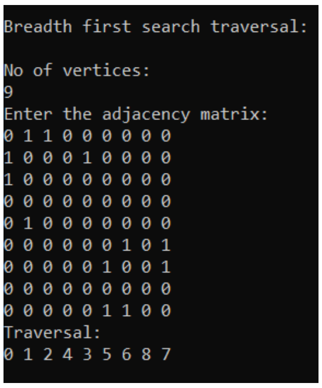


```
Depth first search traversal:

No of vertices:
9
Enter the adjacency matrix:
0 1 1 0 0 0 0 0 0
1 0 0 0 1 0 0 0 0
1 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0 1
0 0 0 0 0 1 0 0 1
0 0 0 0 0 0 0 0 0
0 0 0 0 0 1 1 0 0
Traversal:
0 1 4 2 3 5 6 8 7
```

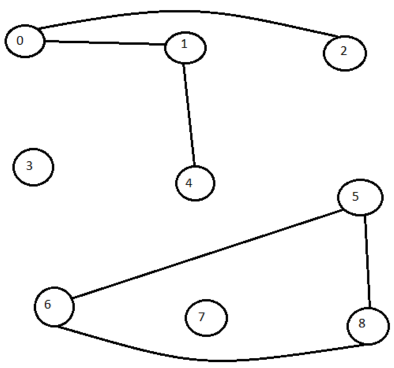


- **Breadth First Search Traversal**

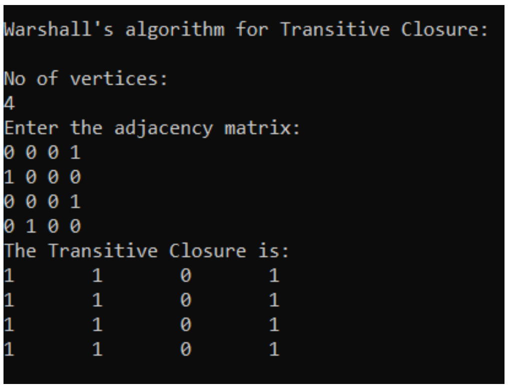
The Input is a Graph. The Output is the Order in which its vertices are visited when following the Breadth First Search Strategy. It involves visiting all the vertices adjacent to one vertex, then, in turn, visiting all the vertices adjacent to those, in order.
The results are:



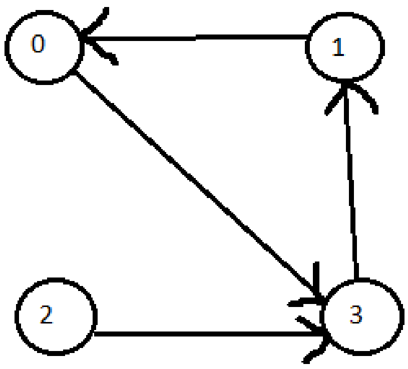
```
Breadth first search traversal:

No of vertices:
9
Enter the adjacency matrix:
0 1 1 0 0 0 0 0 0
1 0 0 0 1 0 0 0 0
1 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0 1
0 0 0 0 0 1 0 0 1
0 0 0 0 0 0 0 0 0
0 0 0 0 0 1 1 0 0
Traversal:
0 1 2 4 3 5 6 8 7
```



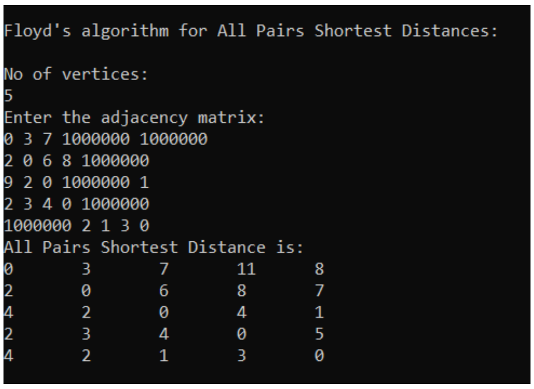
- **Warshall's Algorithm for Transitive Closure**
  The Input is a Graph. The Output is the Transitive Closure of the graph using Warshall's Algorithm. The Transitive Closure of a graph is another graph that contains a direct edge between any two vertices that can be directly or indirectly connected using the other initial edges of the graph.
  The results are:



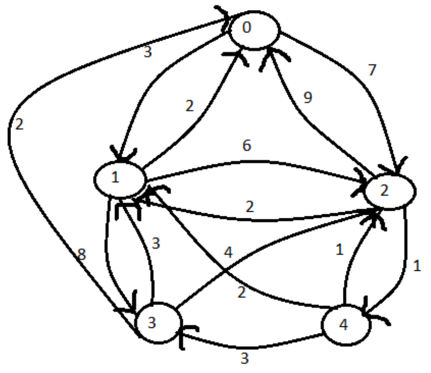
```
Warshall's algorithm for Transitive Closure:

No of vertices:
4
Enter the adjacency matrix:
0 0 0 1
1 0 0 0
0 0 0 1
0 1 0 0
The Transitive Closure is:
1       1       0       1
1       1       0       1
1       1       0       1
1       1       0       1
```



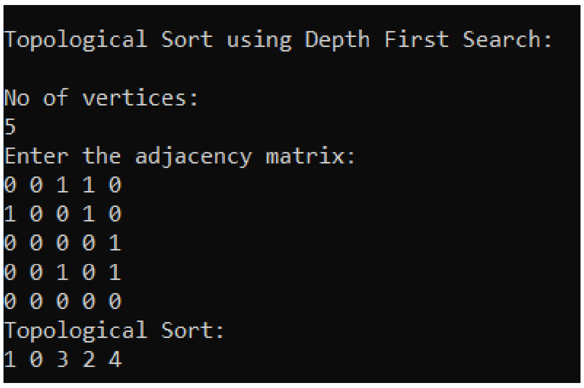
- **Floyd's Algorithm for All Pairs Shortest Distances**
  The Input is a Graph. The Output is a graph with direct edges between two vertices with an edge weight that is the sum of the edge weights of the minimum weight direct or indirect path between them.
  The results are:



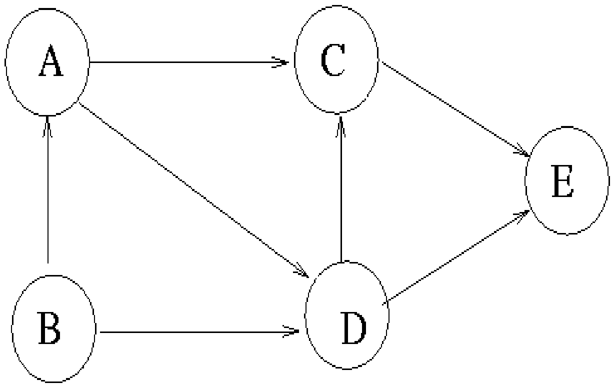
```
Floyd's algorithm for All Pairs Shortest Distances:

No of vertices:
5
Enter the adjacency matrix:
0 3 7 1000000 1000000
2 0 6 8 1000000
9 2 0 1000000 1
2 3 4 0 1000000
1000000 2 1 3 0
All Pairs Shortest Distance is:
0       3       7       11      8
2       0       6       8       7
4       2       0       4       1
2       3       4       0       5
4       2       1       3       0
```



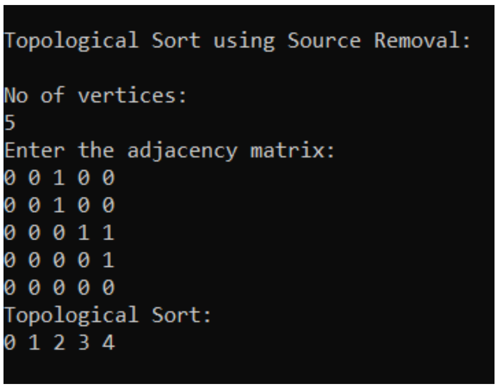
- **Topological Sort using Depth First Search Pop Reversal**
  The Input is an Acyclic Directed Graph. The Output is the result of the Topological Sort of the graph using the Depth First Search Pop Reversal. Topologically Sorting a graph leads to an order of vertices such that, for a vertex v2 that appears after a vertex v1, there is no directed edge from v2 to v1. This is accomplished by reversing the order in which vertices are popped from the Depth First Search's Stack.
  The results are:



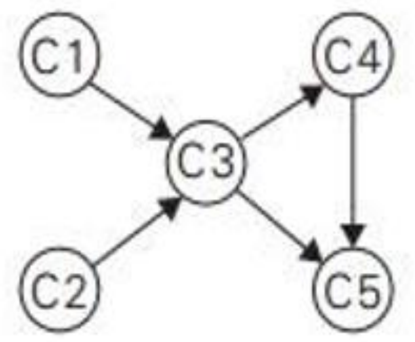
```
Topological Sort using Depth First Search:

No of vertices:
5
Enter the adjacency matrix:
0 0 1 1 0
1 0 0 1 0
0 0 0 0 1
0 0 1 0 1
0 0 0 0 0
Topological Sort:
1 0 3 2 4
```



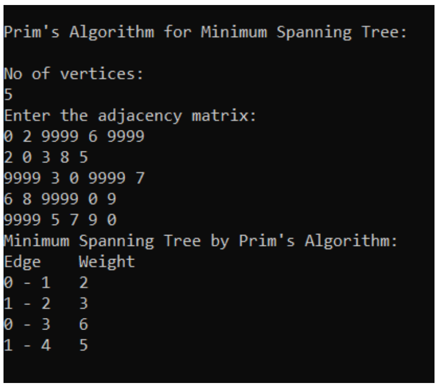
- **Topological Sort using Source Removal**
  The Input and Output are the same as the other Topological Sort Implementation. However, the Strategy followed here is to remove Source Vertices, ie, remove a vertex that has no incoming edges and remove all its outgoing edges, till none are left. The order of removal is the Topological Sorting of the graph.
  The results are:



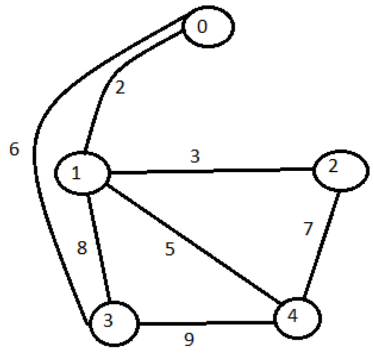
```
Topological Sort using Source Removal:

No of vertices:
5
Enter the adjacency matrix:
0 0 1 0 0
0 0 1 0 0
0 0 0 1 1
0 0 0 0 1
0 0 0 0 0
Topological Sort:
0 1 2 3 4
```



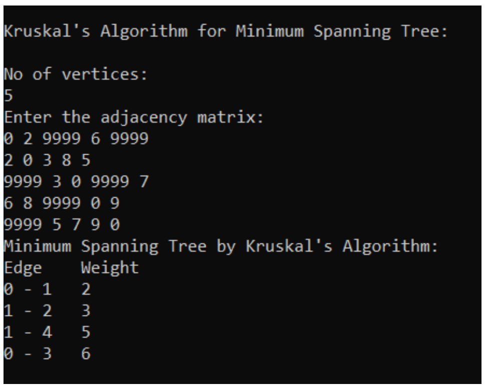
- **Prim’s Algorithm for Minimum Spanning Tree**
  The Input is an Undirected Graph. The Output is a Graph, in the form of an adjacency list, that contains all the vertices, and a configuration of edges (one less in number than the number of vertices) such that the sum of all their weights is minimum. Prim’s Algorithm accomplishes this by choosing minimum weight edges from a set of fringe (vertices adjacent to the vertices in the current MST) vertices’ edges for every vertex in the graph and adding them to the MST.
  The results are:



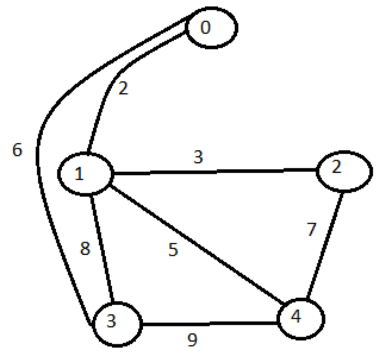
```
Prim's Algorithm for Minimum Spanning Tree:

No of vertices:
5
Enter the adjacency matrix:
0 2 9999 6 9999
2 0 3 8 5
9999 3 0 9999 7
6 8 9999 0 9
9999 5 7 9 0
Minimum Spanning Tree by Prim's Algorithm:
Edge    Weight
0 - 1   2
1 - 2   3
0 - 3   6
1 - 4   5
```



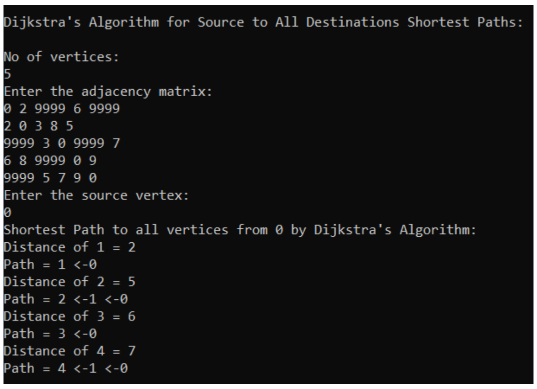
- **Kruskal’s Algorithm for Minimum Spanning Tree**
  The Input and Output are the same as Prim’s Algorithm Implementation for finding Minimum Spanning Tree. However, Kruskal’s Algorithm accomplishes this by sorting all the edges in the graph by non decreasing order of weight, and the inserting them into the MST only if the insertion does not cause any cycle to be formed in the MST.
  The results are:



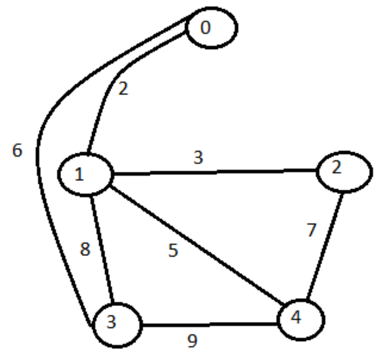
```
Kruskal's Algorithm for Minimum Spanning Tree:

No of vertices:
5
Enter the adjacency matrix:
0 2 9999 6 9999
2 0 3 8 5
9999 3 0 9999 7
6 8 9999 0 9
9999 5 7 9 0
Minimum Spanning Tree by Kruskal's Algorithm:
Edge    Weight
0 - 1   2
1 - 2   3
1 - 4   5
0 - 3   6
```



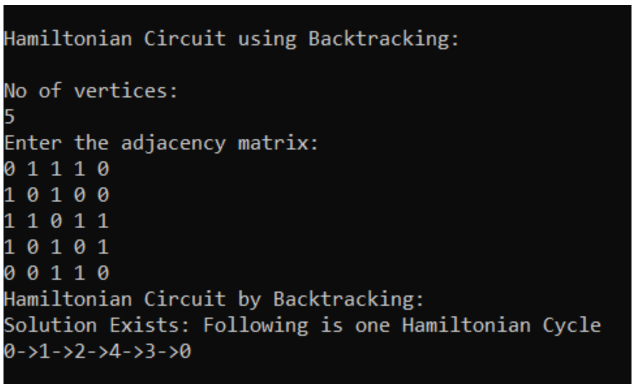
- **Dijkstra's Algorithm for Source to All Destinations Shortest Paths**
  The Input is a Graph and a Source Vertex. The Output is the shortest distance and path to all the other vertices from the Source. Dijkstra's Algorithm accomplishes this by finding the Shortest Path to its adjacent vertices first, then using these to find the shortest paths to their adjacent vertices and so on. It could be called an implementation of Breadth First Search.
  The results are:



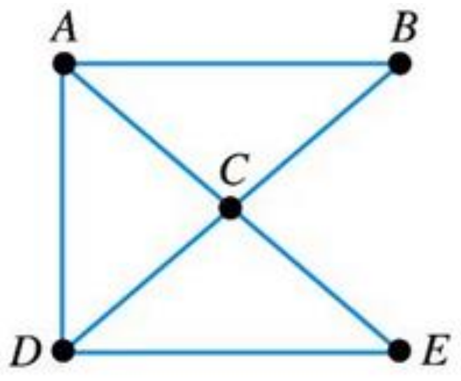
```
Dijkstra's Algorithm for Source to All Destinations Shortest Paths:

No of vertices:
5
Enter the adjacency matrix:
0 2 9999 6 9999
2 0 3 8 5
9999 3 0 9999 7
6 8 9999 0 9
9999 5 7 9 0
Enter the source vertex:
0
Shortest Path to all vertices from 0 by Dijkstra's Algorithm:
Distance of 1 = 2
Path = 1 <-0
Distance of 2 = 5
Path = 2 <-1 <-0
Distance of 3 = 6
Path = 3 <-0
Distance of 4 = 7
Path = 4 <-1 <-0
```



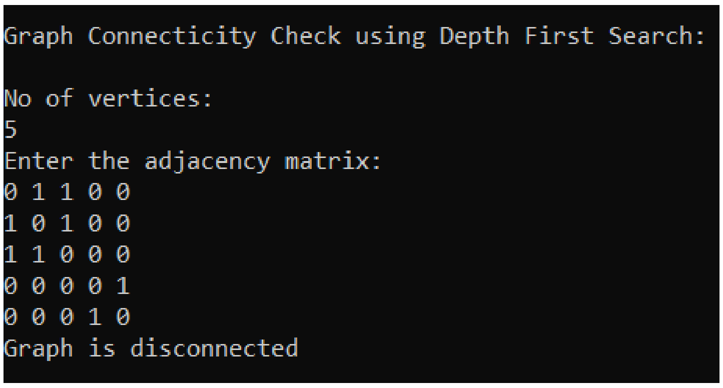
- **Hamiltonian Circuit using Backtracking**
  The Input is a Graph. The Output is any Hamiltonian Circuit that exists, else, no solution if none exists. This is accomplished using Backtracking which involves checking the feasibility of a solution (here, if a path back to initial vertex exists from the final visited vertex), and backtracking to take a different approach (here, visiting a different vertex) if found not feasible. This is done till a feasible solution is found.
  The results are:



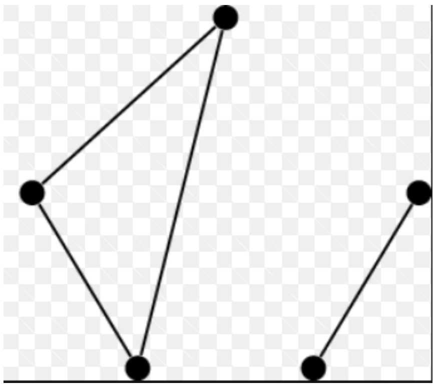
```
Hamiltonian Circuit using Backtracking:

No of vertices:
5
Enter the adjacency matrix:
0 1 1 1 0
1 0 1 0 0
1 1 0 1 1
1 0 1 0 1
0 0 1 1 0
Hamiltonian Circuit by Backtracking:
Solution Exists: Following is one Hamiltonian Cycle
0->1->2->4->3->0
```



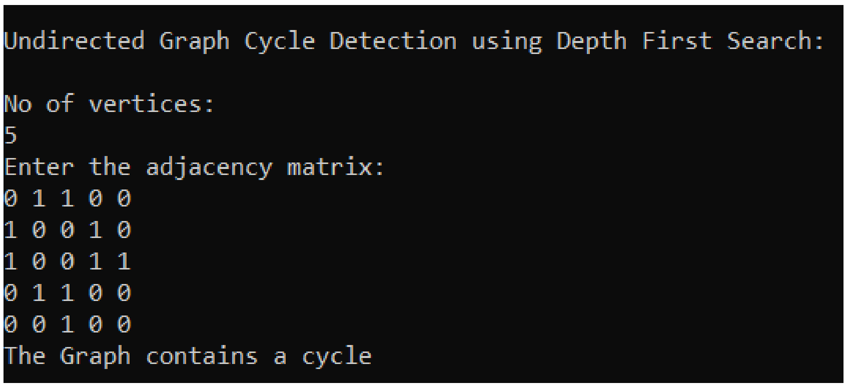
- **Connectivity Check using Depth First Search**
  The Input is an Undirected Graph. The Output is 1 if the Graph contains more than 1 Component, 0 if not. This is accomplished by checking if it takes more than one DFS traversal to visit every vertex.
  The results are:



```
Graph Connecticity Check using Depth First Search:

No of vertices:
5
Enter the adjacency matrix:
0 1 1 0 0
1 0 1 0 0
1 1 0 0 0
0 0 0 0 1
0 0 0 1 0
Graph is disconnected
```



- **Undirected Graph Cycle Detection using Depth First Search**
  The Input is an Undirected Graph. The Output is 1 if the graph contains a Cycle, 0 otherwise. To accomplish this, we check if, during a DFS traversal, the current vertex contains an edge to a vertex that is already visited.
  The results are:



```
Undirected Graph Cycle Detection using Depth First Search:

No of vertices:
5
Enter the adjacency matrix:
0 1 1 0 0
1 0 0 1 0
1 0 0 1 1
0 1 1 0 0
0 0 1 0 0
The Graph contains a cycle
```