```
using System;
using System.Collections.Generic;
using System.Linq;

namespace Assignment_2
{
    /// <summary>
    /// Main class for the application
    /// </summary>
    class Program
    {
        /// <summary>
        /// Main method for the application
        /// </summary>
        /// <param name="args"></param>
        static void Main(string[] args)
        {
            // Create global variables
            int cityCount;
            List<City> CityList = new List<City>();
            List<State> StateList = new List<State>();
            List<Connection> ConnectionList = new List<Connection>();
            State SelectedState = new State();
            bool verbose = false;
            // End of create global variables

            // verbose option input from user
            Console.WriteLine("Do you want the verbose version (y/n):");
            string v = Console.ReadLine();
            if (v =="y")
            {
                verbose = true;
            }// End of verbose option input from user

            // Begin input-graph input
            Console.WriteLine("Please enter total number of cities including Base
              city:");
            cityCount = int.Parse(Console.ReadLine());
            // Call functionality to enter city name from the graph
            CityNameInput(cityCount, CityList);
            // Call functionality to enter path cost of each path
            PathCostInput(cityCount, CityList, ConnectionList);
            // End input-graph input

            // Create base class and calculate its minimum threshold
            if (verbose)
            {
                Console.WriteLine("Creating Base class of the graph.\n");
            }
            // Call priorityList implementation to insert the base state
            InsertInOrder(StateList, new State
            {
```

```
                StateName = "BaseState",
                Cities = CityList,
                ConcideredList = ConnectionList,
                ExcludeList = new List<Connection>(),
                IncludeList = new List<Connection>(),
                MinimumThreshold = 0,
                Status = true
            });
            // Call functionality to calculate minimum threshold of base state
            StateList.First().MinimumThreshold = CalculateMinimumThreshold
              (StateList.First(), StateList, verbose);
            if (verbose)
            {
                Console.WriteLine("Minimum threshold of base state ={0}\n",
                  StateList.First().MinimumThreshold);
            }//End base state initialization

            // Call functionality to get next parent by Best first Search
            SelectedState = GetNextState(StateList);
            if (verbose)
            {
                Console.WriteLine("Generating children nodes for parent: {0}\n",
                  SelectedState.StateName);
            }
            // Call functionality to generate children of the parent state
            GenerateChildState(CityList, StateList, SelectedState.ConcideredList,
              SelectedState, verbose);
        }// End of main method

        /// <summary>
        /// Functionality to generate children of the Parent state and continue
        /// </summary>
        /// <param name="CityList"></param>
        /// <param name="StateList"></param>
        /// <param name="ConnectionList"></param>
        /// <param name="SelectedState"></param>
        /// <param name="verbose"></param>
        private static void GenerateChildState(List<City> CityList, List<State>
          StateList, List<Connection> ConnectionList, State SelectedState, bool
          verbose)
        {
            // Finding out the next connection to be considered for branching
            Connection ConcerenedConnection = new Connection();
            // Choosing from parent's list of connections handed down by the parent
            if (ConnectionList.Any(x=>x.Considered==false))
            {
                // Selecting connection that has not been considered before
                ConcerenedConnection = ConnectionList.Where(x => x.Considered ==
                  false).First();
            }
            else
            {
```

```
            // Else return with null if all connections have been considered
            return;
        }
        // Mark the selected connection as considered for future reference
        ConcerenedConnection.Considered = true;

        if (verbose)
        {
            Console.WriteLine("Considering connection: {0}\n",
              ConcerenedConnection.ConnectionName);
        }

        // Creating child state that includes the concerend connection and
          calculating its minimum threshold
        // Creating seperate copy of parent state
        State IState = SelectedState.CreateDeepCopy(SelectedState);
        // Adding connection to be considered in its include list data structure
        IState.IncludeList.Add(ConcerenedConnection);
        // Call generate name functionality to name the child state
        IState.StateName = GenerateStatename(IState.IncludeList,
          IState.ExcludeList);
        // Initialize the state as active
        IState.Status = true;
        // Call functionality to calculate the state's minimum threshold
        IState.MinimumThreshold = CalculateMinimumThreshold(IState, StateList,
          verbose);
        // If the state is active
        if (IState.Status)
        {
            // Insert it in the state prioritylist
            InsertInOrder(StateList, IState);
        }
        // If the state has a valid minimum threshold
        if (IState.MinimumThreshold > 0)
        {
            if (verbose)
            {
                Console.WriteLine("Minimum threshold for {0} = {1}",
                  IState.StateName, IState.MinimumThreshold);
            }
            // And if it id the minimum among all state after considering all
              connections
            if ((IState.MinimumThreshold<=SelectedState.MinimumThreshold)&&
              (IState.IncludeList.Count
              +IState.ExcludeList.Count==IState.ConcideredList.Count))
            {
                // Call print functionality to print optimal path as the state is
                  the optimal state
                PrintFinalState(IState);
            }// Else continue
        }// Else continue
        // End of creating child state that includes the concerend connection
```

```
            // Creating child state that excludes the concerend connection and
              calculating its minimum threshold
            // Creating seperate copy of parent state
            State EState = SelectedState.CreateDeepCopy(SelectedState);
            // Adding connection to be considered in its exclude list data structure
            EState.ExcludeList.Add(ConcerenedConnection);
            // Call generate name functionality to name the child state
            EState.StateName = GenerateStatename(EState.IncludeList,
              EState.ExcludeList);
            // Initialize the state as active
            EState.Status = true;
            // Call functionality to calculate the state's minimum threshold
            EState.MinimumThreshold = CalculateMinimumThreshold(EState, StateList,
              verbose);
            // If the state is active
            if (EState.Status)
            {
                // Insert it in the state prioritylist
                InsertInOrder(StateList, EState);
            }
            // If the state has a valid minimum threshold
            if (EState.MinimumThreshold > 0)
            {
                if (verbose)
                {
                    Console.WriteLine("Minimum threshold for {0} = {1}\n",
                      EState.StateName, EState.MinimumThreshold);
                }
                // And if it id the minimum among all state after considering all
                  connections
                if ((EState.MinimumThreshold <= SelectedState.MinimumThreshold) &&
                  (EState.IncludeList.Count + EState.ExcludeList.Count ==
                  EState.ConcideredList.Count))
                {
                    // Call print functionality to print optimal path as the state is
                      the optimal state
                    PrintFinalState(EState);
                }// Else continue
            }// Else continue
            // End of creating child state that excludes the concerend connection

            if (verbose)
            {
                Console.WriteLine("Deleting Parent: {0}\n", SelectedState.StateName);
            }
            // Delete existing parent from state priority list
            StateList.RemoveAll(x => x.StateName == SelectedState.StateName);

            // Call functionality to get next parent by Best First Search
            SelectedState = GetNextState(StateList);
            if (verbose)
```

```
            {
                Console.WriteLine("The next state to be parent: {0}\n",
                  SelectedState.StateName);
            }
            if (verbose)
            {
                Console.WriteLine("Generating children nodes for parent: {0}\n",
                  SelectedState.StateName);
            }
            // Call functionality to generate children of the new parent state
            GenerateChildState(CityList, StateList, SelectedState.ConcideredList,
              SelectedState, verbose);
        }// End of GenerateChildState method

        /// <summary>
        /// Functionality to print the optimal path of the TSP
        /// </summary>
        /// <param name="state"></param>
        private static void PrintFinalState(State state)
        {
            // Initialize string to display optimal path
            string FinalAnswer = "";
            // Delete all connections from the final state that are in its exclude
              list data structure
            foreach (Connection delCon in state.ExcludeList)
            {
                // Consider each city in the final state
                foreach (City delCity in state.Cities)
                {
                    if (delCity.Connections.Exists(x => x.ConnectionName ==
                      delCon.ConnectionName))
                    {
                        delCity.Connections.RemoveAll(x => x.ConnectionName ==
                      delCon.ConnectionName);
                    }
                }
            }// End of excluding connections from final state
            // Print filtered connection for the optimal path
            foreach (City finalCity in state.Cities)
            {
                // Consider all cities of the final state
                foreach (Connection finalConnection in finalCity.Connections)
                {
                    // Avoid duplicate connections
                    if (!FinalAnswer.Contains(finalConnection.ConnectionName))
                    {
                        FinalAnswer += " " + finalConnection.ConnectionName + ";";
                    }
                }
            }
            // Print the optimal path
            Console.WriteLine("The optimal path for the TSP is :{0} with a path cost
```

```
                of {1}.", FinalAnswer, state.MinimumThreshold);
            Console.ReadLine();
            // Exit application
            Environment.Exit(0);
        }// End of PrintFinalState method

        /// <summary>
        /// Functionality to get next parent state
        /// </summary>
        /// <param name="StateList"></param>
        /// <returns></returns>
        private static State GetNextState(List<State> StateList)
        {
            // Initialize local variable to hold the result
            State SelectedState;
            // Retrieve best state to be considered
            SelectedState = StateList.Where(x => x.Status == true).First();
            // Return result
            return SelectedState;
        }// End of GetNextState method

        /// <summary>
        /// Functionality to generate name of children states
        /// </summary>
        /// <param name="Include"></param>
        /// <param name="Exclude"></param>
        /// <returns></returns>
        private static string GenerateStatename(List<Connection> Include,
          List<Connection> Exclude)
        {
            string include = "";
            string exclude = "";
            string stateName = "State (";

            if (Include != null)
            {
                // Include all connections in the include part of the state name
                foreach (Connection item in Include)
                {
                    include += item.ConnectionName + "; ";
                }
            }
            else
            {
                // Else give - if no connections are present in the include data
                  structure
                include = "-";
            }
            if (Exclude != null)
            {
                // Include all connections in the exclude part of the state name
                foreach (Connection item in Exclude)
```

```
                {
                    exclude += item.ConnectionName + "; ";
                }
            }
            else
            {
                // Else give - if no connections are present in the exclude data
                  structure
                exclude = "-";
            }
            // Generate state name by concatenating all parts
            stateName = stateName + "Including: " + include + " Excluding: " + exclude
               +")";
            // Return the state name
            return stateName;
        }// End of GenerateStatename method

        /// <summary>
        /// Functionality to calculate the minimum threshold of the given state
        /// </summary>
        /// <param name="currentState"></param>
        /// <param name="stateList"></param>
        /// <param name="verbose"></param>
        /// <returns></returns>
        private static double CalculateMinimumThreshold(State currentState,
          List<State> stateList, bool verbose)
        {
            // Initiate local variable to hold the minimum threshold of the class
            double minimumthreshold = 0;
            // Create copy of given state for manipulation
            State MinConState = currentState.CreateDeepCopy(currentState);
            // Delete all connections in state that are present in its exclude data
              structure
            foreach (Connection delCon in MinConState.ExcludeList)
            {
                foreach (City delCity in MinConState.Cities)
                {
                    if (delCity.Connections.Exists
                      (x=>x.ConnectionName==delCon.ConnectionName))
                    {
                        delCity.Connections.RemoveAll(x => x.ConnectionName ==
                      delCon.ConnectionName);
                        // If a city in the given state is left with only one
                      connection
                        if (delCity.Connections.Count<2)
                        {
                            // Prune the given state
                            currentState.Status = false;
                            if (verbose)
                            {
                                Console.WriteLine(currentState.StateName + " pruned
                      because city '" + delCity.CityName + "' needs at least two
```

```
                    connections.");
                        }
                        // Return minimum threshold as 0 and exit the method
                        return 0;
                    }
                }
            }
        }
        // Prioritize all connections in state that are present in its include
          data structure
        foreach (Connection inCon in MinConState.IncludeList)
        {
            foreach (City inCity in MinConState.Cities)
            {
                var result = inCity.Connections.Where(p =>
                  MinConState.IncludeList.Any(p2 => p2.ConnectionName ==
                  p.ConnectionName));
                // If a city in the given state has more than two connection in
                  the include data structure
                if(result.Count()>2)
                {
                    // Prune the given state
                    currentState.Status = false;
                    if (verbose)
                    {
                        Console.WriteLine(currentState.StateName + " pruned
                  because city " + inCity.CityName + " already has two of it's
                  connections included.");
                    }
                    // Return minimum threshold as 0 and exit the method
                    return 0;
                }
                // Otherwise prioritize the include data structure connections in
                  every city of the given state
                if (inCity.Connections.Exists(x => x.ConnectionName ==
                  inCon.ConnectionName))
                {
                    inCity.Connections.RemoveAll
                  (x=>x.ConnectionName==inCon.ConnectionName);
                    inCity.Connections.Insert(0, inCon);
                }
            }
        }
        // Calculate the minimum threshold for the given state
        foreach (City city in MinConState.Cities)
        {
            minimumthreshold += (city.Connections.ElementAt(0).PathCost +
              city.Connections.ElementAt(1).PathCost);
        }
        // Return minimum threshold of the given state
        return minimumthreshold /= 2.00;
    }// End of CalculateMinimumThreshold method
```

```
        /// <summary>
        /// Functionality to take path cost of individual path from the user
        /// </summary>
        /// <param name="cityCount"></param>
        /// <param name="cityList"></param>
        /// <param name="connectionList"></param>
        private static void PathCostInput(int cityCount, List<City> cityList,
          List<Connection> connectionList)
        {
            // Consider all cities
            for (int i = 0; i < cityCount; i++)
            {
                // Consider all combinations between all cities
                for (int j = i + 1; j < cityCount; j++)
                {
                    Console.WriteLine("Please enter path cost from " +
                      cityList.ElementAt(i).CityName + " to " + cityList.ElementAt
                      (j).CityName +
                        "(Enter 0 if no path exists):");
                    // Take input from keyboard
                    int pathCost = int.Parse(Console.ReadLine());
                    // Create instance of connection class
                    Connection path = new Connection { ConnectionName=
                      cityList.ElementAt(i).CityName + cityList.ElementAt(j).CityName,
                        Source = cityList.ElementAt(i), Destination =
                      cityList.ElementAt(j), PathCost = pathCost };
                    // If the path is valid
                    if (path.PathCost > 0)
                    {
                        // Insert it into the priority list of source city
                        InsertInOrder(cityList.ElementAt(i).Connections, path);
                        // Insert it into the priority list of destination city
                        InsertInOrder(cityList.ElementAt(j).Connections, path);
                        // Avoid duplicates
                        if (!connectionList.Contains(path))
                        {
                            // Insert it into the priority list of all connections
                            InsertInOrder(connectionList, path);
                        }
                    }
                }
            }
        }// End of PathCostInput method

        /// <summary>
        /// Functionality to take city names of individual city from the user
        /// </summary>
        /// <param name="cityCount"></param>
        /// <param name="CityList"></param>
        private static void CityNameInput(int cityCount, List<City> CityList)
        {
```

```
            // Retrieving name for base city from the keyboard
            Console.WriteLine("Please enter name of Base city:");
            // Creating an instace of city class and adding it to the list of all
              cities
            CityList.Add(new City { CityName = Console.ReadLine() });
            for (int i = 2; i <= cityCount; i++)
            {
                // Create new instance of city class and naming it by reading input
                  from keyboard
                Console.WriteLine("Please enter name of city " + i + ":");
                // Adding the instance in list of all cities
                CityList.Add(new City { CityName = Console.ReadLine() });
            }
        }// End of CityNameInput class

        /// <summary>
        /// Priority list implementation for connections
        /// </summary>
        /// <param name="inputConList"></param>
        /// <param name="inputCon"></param>
        private static void InsertInOrder(List<Connection> inputConList, Connection
          inputCon)
        {
            // If existing list is not empty
            if (inputConList.Count>0)
            {
                // If existing list has an element with minimum threshold higher than
                  the considered connection
                if (inputConList.Exists(x=>x.PathCost>=inputCon.PathCost))
                {
                    // Insert the considered connection before the found element
                    inputConList.Insert(inputConList.FindIndex(x => x.PathCost >=
                      inputCon.PathCost), inputCon);
                }
                else
                {
                    // Else add the element in the end of the list
                    inputConList.Add(inputCon);
                }
            }
            else
            {
                // If existing list is empty, insert the element in the first position
                   in the list
                inputConList.Insert(0, inputCon);
            }
        }// End of InsertInOrder method

        /// <summary>
        /// Priority list implementation for states
        /// </summary>
        /// <param name="inputStateList"></param>
```

```
        /// <param name="inputState"></param>
        private static void InsertInOrder(List<State> inputStateList, State
          inputState)
        {
            // If existing list is not empty
            if (inputStateList.Count > 0)
            {
                // If existing list has an element with minimum threshold higher than
                  the considered connection
                if (inputStateList.Exists(x => x.MinimumThreshold >=
                  inputState.MinimumThreshold))
                {
                    // Insert the considered connection before the found element
                    inputStateList.Insert(inputStateList.FindIndex(x =>
                      x.MinimumThreshold >= inputState.MinimumThreshold), inputState);
                }
                else
                {
                    // Else add the element in the end of the list
                    inputStateList.Add(inputState);
                }
            }
            else
            {
                // If existing list is empty, insert the element in the first position
                   in the list
                inputStateList.Insert(0, inputState);
            }
        }// End of InsertInOrder method
    }// End of main program
}
```