namespace SearchTrees

{

public class RomeniaMapProblemDepthFirstSearch: IMiniProblem

{

private decimal \_costOfTheWay;

private int \_depth;

private string \_initialState;

private string \_objectiveState;

private string \_actionsAlongTheWay;

private Node \_solutionNode;

private IList<string> \_statesSpace;

private IList<Node> \_nodes;

private int \_depthLimit;

private Stack<Node> \_stackNodesOfTree;

public RomeniaMapProblemDepthFirstSearch(string initialStateName, string objectiveStateName, IList<string> statesSpace, int? depthLimit = -1)

{

\_costOfTheWay = 0;

\_depth = 0;

\_depthLimit = depthLimit ?? -1;

\_nodes = new List<Node>();

\_statesSpace = statesSpace;

\_actionsAlongTheWay = string.Empty;

\_initialState = InstanceStateByTheName(initialStateName);

\_objectiveState = InstanceStateByTheName(objectiveStateName);

}

private string InstanceStateByTheName(string stateName)

{

if (string.IsNullOrEmpty(stateName)) {

throw new ArgumentNullException("O nome do estado não pode ser nulo ou vazio.");

}

var state = \_statesSpace.Where(field => field.Equals(stateName));

if (!state.Any())

{

throw new ArgumentException($"O nome do estado '{stateName}', não pertence ao espaço " +

"de estados do problema.");

}

\_nodes.Add(new Node(stateName, null, NodeAction.UNDEFINED, 0));

return state.FirstOrDefault();

}

public IList<string> statesSpace { get => \_statesSpace; }

public string InitialState { get => \_initialState; }

public IList<Node> Nodes { get => \_nodes; }

public Node SolutionNode { get => \_solutionNode; }

public int Depth { get => \_depth; }

public int DepthLimit { get => \_depthLimit; }

public decimal CostOfTheWay { get => \_costOfTheWay; }

public string ActionsAlongTheWay { get => \_actionsAlongTheWay; }

public void AddChildToParent(string childNodeName, string parentNodeName, decimal costOfTheWay, NodeAction action)

{

if (string.IsNullOrEmpty(childNodeName)) {

throw new ArgumentNullException("O nome da nó filho não pode ser um valor nulo ou vazio.");

}

if (string.IsNullOrEmpty(parentNodeName)) {

throw new ArgumentNullException("O nome da nó pai não pode ser um valor nulo ou vazio.");

}

GetParentNode(childNodeName, parentNodeName, costOfTheWay, action);

}

private void GetParentNode(string childNodeName, string parentNodeName,

decimal costOfTheWay, NodeAction action)

{

var parentNode = \_nodes.FirstOrDefault(field => field.State == parentNodeName);

if (parentNode == null) {

throw new ArgumentException($"O nome do estado pai '{parentNodeName}', não pertence ao espaço " +

"de estados do problema.");

}

AddChildNodeTo(parentNode, childNodeName, costOfTheWay, action);

}

private void AddChildNodeTo(Node parentNode, string childNodeName,

decimal costOfTheWay, NodeAction action)

{

var childState = \_statesSpace.Where(field => field.Equals(childNodeName)).FirstOrDefault();

if (childState == null)

{

throw new ArgumentException($"O nome do estado filho '{childNodeName}', não pertence ao espaço " +

"de estados do problema.");

}

\_nodes.Add(new Node(childState, parentNode, action, costOfTheWay));

}

public void SearchTree()

{

var rootNode = \_nodes

.Where(field => field.ParentNode == null && field.State == \_initialState)

.ToList();

\_depth += 1;

\_stackNodesOfTree = new Stack<Node>();

\_stackNodesOfTree.Push(rootNode.FirstOrDefault());

DepthFirstSearch(rootNode.FirstOrDefault());

}

private void DepthFirstSearch(Node node)

{

if (!IsSolutionContainedIn(node)){

HasDepthLimited(node);

}

}

private void HasDepthLimited(Node node)

{

if (\_depthLimit == -1 || \_depth < \_depthLimit)

{

ExpandNode(node);

\_depth += 1;

}

}

private bool IsSolutionContainedIn(Node nodeToValidate)

{

\_solutionNode = nodeToValidate.State == \_objectiveState ? nodeToValidate : null;

if (\_solutionNode == null){

return false;

}

return true;

}

private void ExpandNode(Node node)

{

Sucessor(node);

while (\_stackNodesOfTree.Count > 0 && \_solutionNode == null)

{

var childNode = \_stackNodesOfTree.Pop();

SetVisitedNode(childNode, node);

\_costOfTheWay += childNode.CostOfTheWay;

\_actionsAlongTheWay += node.State + " -> " + node.Action + " -> " + childNode.State + " | ";

DepthFirstSearch(childNode);

}

}

private void Sucessor(Node selectedNode)

{

List<Node> childrenNodes = new List<Node>();

if (selectedNode.ParentNode == null)

{

childrenNodes.AddRange(\_nodes.Where(node =>

node.ParentNode != null &&

node.ParentNode.State == selectedNode.State)

.OrderByDescending(field => field.CostOfTheWay) .ToList());

}

else

{

// It is needed because the nodes contains ways from City1 to City2 and City2 to 1 (GO and BACK actions)

childrenNodes.AddRange(\_nodes.Where(node =>

node.ParentNode != null &&

node.ParentNode.State == selectedNode.State &&

node.State != selectedNode.ParentNode.State &&

(node.Action == NodeAction.GO || node.Action == NodeAction.BACK)) //<-- Excluding those with actions equals to 'GONE' and 'BACKED'. ONLY NODES STILL DON'T VISITED!

.OrderByDescending(field => field.CostOfTheWay)

.ToList());

}

childrenNodes.ForEach(node => {

\_stackNodesOfTree.Push(node);

});

}

private void SetVisitedNode(Node childNode, Node parentNode)

{

var visitedNodeAction = childNode.Action == NodeAction.GO ? NodeAction.GONE :

childNode.Action == NodeAction.BACK ? NodeAction.BACKED : parentNode.Action;

\_nodes.ToList().ForEach(auxNode =>

{

if ((auxNode.ParentNode != null) &&

(auxNode.ParentNode.State == parentNode.State && auxNode.State == childNode.State))

{

auxNode.SetNodeAction(visitedNodeAction);

}

});

}

}

}