Torkelnde Yamyams

Inhalt

1 Lösungsidee

Um die sicheren Felder zu finden wird, zuerst ein gerichteter Graph G aus den Spielfeld aufgebaut. Dazu wird jedes besuchbare Feld (ein Feld auf dem keine Wand steht) von einem Knoten dargestellt. Dann bekommt der Knoten jedes Feldes 4 Kanten, eine zu jedem Knoten eines Feldes, das durch Fahren in eine der 4 Himmelsrichtungen bis zur ersten Wand oder zum ersten Ausgang erreicht werden kann. Kanten, die ein Feld mit sich selbst verbinden würden, werden nicht berücksichtigt. Außerdem verfügen die Knoten von Ausgangsfeldern über keine ausgehenden Kanten, da es unmöglich ist ein Ausgangsfeld wieder zu verlassen. Im Folgenden werden Knoten und ihre dazugehörigen Felder synonym verwendet.

Wenn man in G die starken Zusammenhangskomponenten betrachtet, dann handelt es sich dabei genau um die Knoten, von denen aus ein Yamyam jeden anderen Knoten in dieser Zusammenhangskomponente erreichen kann. Deshalb ist es ausreichend, wenn man nicht G an sich betrachtet, sondern nur den Graphen der Zusammenhangskomponenten von G, G_z genannt.

Für jeden Graphen aus Zusammenhangskomponenten gilt, dass es sich dabei um einen DAG handelt.[1] Folglich ist G_z auch ein DAG. Zu jedem DAG lässt sich eine topologische Sortierung seiner Knoten finden.[2] Betrachtet man die Invertierung einer solchen Sortierung, so lassen sich leicht alle sichern Felder ausfindig machen:

- Wenn ein Knoten K in G_z keine Knoten enthält, die eine Kante zu einem Knoten in einer anderen Zusammenhangskomponente haben, und K mindestens einen Ausgangsknoten enthält, dann führt jeder Knoten in K sicher zu einem Ausgang. Deshalb kann man K als sicher bezeichnen.
- Wenn alle Knoten in G_z, zu denen der aktuelle Knoten K eine Kante hat, sicher sind, dann ist automatisch auch K sicher.

Wenn die Knoten in G_z invertiert topologisch sortiert sind, dann ist sichergestellt, dass erst Knoten in G_z ohne Ausgangskanten betrachtet werden, und dann Knoten, die nur zu Knoten führen die bereits betrachtet wurden und für die deshalb schon gesagt werden kann ob sie sicher oder nicht sind.

[1] https://en.wikipedia.org/wiki/Strongly_connected_component
[2] https://en.wikipedia.org/wiki/Topological_sorting

Um diesen Ansatz umzusetzen müssen alle starken Zusammenhangskomponenten gefunden werden. Dazu kommen verschieden Algorithmen in Frage, der geeignetes ist aber der Algorithmus von Tarjan zur Bestimmung starker Zusammenhangskomponenten[3], da er die nützliche Eigenschaft hat, dass die gefundenen Zusammenhangskomponenten in invertierter topologischer Sortierung ausgegeben werden.

Der Algorithmus von Tarjan funktioniert wie folgt: Es wird solange eine Tiefensuche auf G durchgeführt, bis alle Knoten besucht wurden, auch über schwache Zusammenhangskomponenten hinweg. Jedem Knoten K werden dabei zwei Labels zugewiesen: Einmal als wievielter Knoten K zuerst von der Tiefensuche besucht wurde (Index von K) und einmal der kleinste Kontenindex zu dem K eine Rückkante hat (Lowlink von K). Zusätzlich kommt noch ein Stack zum Einsatz, auf den jeder Knoten beim ersten Besuchen gepusht wird. Wenn ein Knoten K zum ersten Mal besucht wird und der rekursive Aufruf für alle seine Nachbarn erfolgt ist, dann wird geprüft, ob der Index und der Lowlink von K gleich sind. Ist das der Fall, dann wurde eine neue Zusammenhangskomponente gefunden, die aus allen Knoten, die sich über K auf dem Stack befinden und K besteht. Deshalb werden alle Knoten in der Komponente vom Stack gepopt und die Komponente wird zurückgegeben. Um das Lowlink von K zu berechnen, wird das minimale Lowlink aller noch unbesuchten Nachbarn von K (nach dem rekursiven Aufruf auf diese) und aller Nachbarn von K die sich aktuell im Stack befinden genommen.

Eine Schwäche dieses Ansatzes ist, dass er auf einem rekursiven Verfahren basiert und deshalb in der Praxis durch Stack-Größen limitiert ist. Es ist zwar möglich, den Algorithmus zu einem iterativen Verfahren umzuformen, da es sich aber nicht um Tail-Recursion handelt, was die Umformung nichttrivial macht, und die bereitgestellten Beispiele allesamt zu klein sind, um an diese Limitierung zu stoßen, habe ich mich gegen eine Umformung entschieden.

2 Umsetzung

Die Aufgabe wurden als Konsolenanwendung in C# umgesetzt. Die Klasse `Program` enthält dabei die grundlegende IO-Logik, während die Verarbeitung der Eingabedaten in `World` erfolgt. Dazu wird der Konstruktor mit der Textdarstellung der Welt als `string` aufgerufen. Die Methode `ParseMap()` erstellt aus diesem `string` eine 2D-Array aus Tiles. Dieses Array wird von der Methode `CreateGraph()` zu dem oben beschriebenen Graphen umgesetzt. Dazu werden erst die horizontalen Nachbarn gefunden und dann die vertikalen. Damit das Einlesen in Linearzeit abläuft, werden erst alle Knoten bis zu einer Wand oder einem Ausgang gespeichert und dann werden für alle gespeicherten Knoten die Adjazenzlisten erstellt. Beide dieser Methoden werden intern vom Konstruktor aufgerufen.

Die einzelnen Knoten des Graphen werden von der Klasse `Node` dargestellt, die neben eine Liste von Nachbarn auch noch zusätzliche Eigenschaften speichert. Dazu gehören die für den Algorithmus von Tarjan benötigten Attribute, aber auch die Koordinate des Feldes, das vom Knoten dargestellt wird und die Information, in welcher Zusammenhangskomponente sich ein Knoten befindet.

Um die sicheren Felder zu ermitteln, muss die Methode `Solve()` aufgerufen werden. Diese führt erst den Algorithmus von Tarjan aus, der in der Methode `ConnectedComponent()` implementiert ist. Dieser füllt eine `List<List<Node>>` mit den Zusammenhangskomponenten. Dann wird in `Solve()` über diese Komponenten iteriert um festzustellen, ob diese sicher sind. Am Ende werden mit Hilfe

[3] https://en.wikipedia.org/wiki/Tarjan's_strongly_connected_components_algorithm

einer LINQ-Query alle sicheren Komponenten ausgewählt und es wird eine sortierte Liste aller Positionen, die die Knoten in den Zusammenhangskomponenten repräsentieren, zurückgegeben.

3 Laufzeit

Zur Betrachtung der Gesamtlaufzeit des Programmes müssen mehrere Methoden betrachtet werden:

- `ParseMap()` geht jedes Zeichen des Eingabestrings genau einmal durch und benötig für jedes Zeichen $\mathcal{O}(1)$. Die Betrachtung erfolgt deshalb in $\mathcal{O}(n)$, wobei n die Anzahl der Eingabezeichen sei.
- `CreateGraph()` betrachtet jede Zeile und jede Spalte des World-Arrays genau zweimal. Dabei wird bei jeder Betrachtung ein Element einer Liste hinzugefügt. In manchen Fällen wird durch diese Liste iteriert und sie wird danach geleert. Folglich wird die Laufzeit für das Iterieren der Liste durch die Anzahl an Hinzufügeoperationen bestimmt. Die Laufzeit von `CreateGraph()` beträgt deshalb $\mathcal{O}(2n) = \mathcal{O}(n)$.
- Der Algorithmus von Tarjan hat als modifizierte Tiefensuche eine Laufzeit von $\mathcal{O}(|V| + |E|)$. Da jeder Knoten in einem wie oben beschrieben konstruierten Graph maximal 4 Kanten hat und die Anzahl der Knoten maximal der Anzahl an Feldern entspricht, gilt $\mathcal{O}(n + 4n) = \mathcal{O}(n)$.
- Um aus den Zusammenhangskomponenten die sicheren Felder zu ermitteln, werden wieder alle Knoten und alle Kanten betrachtet, wobei die Betrachtung in $\mathcal{O}(1)$ erfolgt. Deshalb gilt für diese Laufzeit wieder $\mathcal{O}(n)$.

Letztendlich ist die Laufzeit aller Teilschritte linear abhängig von n, sodass die Gesamtlaufzeit des Programmes $\mathcal{O}(n)$ beträgt.

4 Erweiterung: Müde Yamyams

Die Yamyams haben einen langen Tag voller gegen die Wand fahren hinter sich und sind jetzt sehr müde. Leider hat ein fieser Informatiker, dem keine guten Erweiterungsideen einfallen, ihnen Steine in den Weg gelegt. Darauf haben die Yamyams aber überhaupt keine Lust, denn um über einen Stein zu fahren muss man Anlauf nehmen und das ist anstrengend (=nicht gut). Danach ist allerdings auch der Stein kaputt. Das selbe ist der Fall, wenn ein Stein sich auf dem Startfeld eines Yamyams befindet. Jetzt wollen die Yamyams wissen, über wie viele Steine sie von einem sichern Feld aus mindestens fahren müssen, um zu einem beliebigen Ausgang zu kommen.

Um den Yamyams aus dieser verzwickten Situation zu helfen, kommt der Algorithmus von Floyd-Warshall zum Einsatz. Damit werden alle kürzesten Strecken in G berechnet. Dabei hat eine Kante genau dann das Gewicht n, wenn sich auf der geradlinigen Strecke zwischen den Knoten, die diese repräsentiert, n Steine befinden. Sonst hat eine Kante das Gewicht 0. Hat man jetzt die Distanzmatix von G berechnet, muss man nur für alle sicheren Felder ermitteln, welcher Ausgang die minimale Distanz hat.

Um Steine beim Einlesen darzustellen, kommt der Buchtstabe ‚S' zum Einsatz. Um diese neuen Buchstaben einlesen zu können, muss natürlich die Einleselogik angepasst werden. Bei `ParseMap()` ist diese Anpassung trivial und bei CreateGraph() muss jetzt zusätzlich gespeichert werden, wo unter den gespeicherten Knoten sich Steine befinden. Anschließende musste noch `Solve()` erweitert werden. Nachdem alle sicheren Felder gefunden wurden, wird jetzt eine Distanzmatrix aufgebaut aus der dann die entsprechenden Entfernungen ausgelesen werden.

Die Laufzeit des Einlesens bleibt weiterhin linear. Sollte ein Spielfeld aber Steine enthalten, so muss der Algorithmus von Floyd-Warshall ausgeführt werde, der in $\mathcal{O}(n^3)$ abläuft. Dadurch ändert sich auch die komplette Laufzeit auf $\mathcal{O}(n^3)$.

5 Beispiele

Dateiname	**Ausgabe**
yamyams0.txt	(4, 2) (4, 7) (5, 2) (6, 2) (6, 4) (6, 7) (7, 2) (7, 4) (7, 7) (8, 2) (8, 4) (8, 7) (9, 2) (9, 4) (9, 7)
yamyams1.txt	Es wurden 113 sichere Felder gefunden.
yamyams2.txt	(2, 6) (2, 7) (2, 8) (2, 9) (2, 10) (3, 10) (4, 10) (4, 11) (4, 13) (4, 14) (5, 14) (6, 14) (26, 18) (46, 14) (47, 14) (49, 14) (50, 14)
yamyams3.txt	Es wurden 203 sichere Felder gefunden.
yamyams4.txt	(2, 12) (2, 13) (2, 14) (4, 7) (4, 12) (4, 13) (4, 14)
yamyams5.txt	Es wurden 111 sichere Felder gefunden.
yamyams6.txt	(2, 2) (2, 3) (2, 5)

	(2, 6) (2, 7) (2, 9) (2, 10) (2, 11) (2, 13) (2, 14) (2, 15) (3, 3) (3, 4) (3, 5) (3, 7) (3, 8) (3, 9) (3, 11) (3, 12) (3, 13) (3, 15) (4, 3) (4, 4) (4, 6) (4, 7) (4, 8) (4, 10) (4, 11) (4, 12) (4, 14) (4, 15) (5, 2) (5, 3) (5, 4) (5, 5) (5, 6) (5, 7) (5, 8) (5, 9) (5, 10) (5, 11) (5, 12) (5, 13) (5, 14) (5, 15)
yamyams7.txt	Es wurden 112 sichere Felder gefunden
pureEvilness.txt	(4, 2, 4) (4, 7, 0) (5, 2, 5) (6, 2, 4) (6, 4, 2) (6, 7, 1) (7, 2, 3) (7, 4, 1)

	(7, 7, 1) (8, 2, 2) (8, 4, 0) (8, 7, 0) (9, 2, 1) (9, 4, 0) (9, 7, 0)
pureEvilness2.txt	Es wurden 111 sichere Felder gefunden.

6 Quellcode

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

namespace Aufgabe_3___Torkelnde_Yamyams
{
	class World
	{
		private enum Tile
		{
			Wall = 0,
			Nothing = 1,
			Stone = 2,
			Exit = 999
		}

		[DebuggerDisplay("{Position} IsExit: {IsExit}")]
		private class Node
		{
			public List<Tuple<Node, int>> Neighbours { get; private set; } =
new List<Tuple<Node, int>>();
			public bool IsExit { get; set; } = false;
			public int Index { get; private set; }

			public int ConnectedComponent { get; set; }

			public bool OnStack { get; set; } = false;
			public int Label { get; set; } = -1;
			public int LowLink { get; set; } = -1;

			public readonly Tuple<int, int> Position;

			public Node(bool isExit, Tuple<int, int> position, int index)
			{
				IsExit = isExit;
				Position = position;
				Index = index;
			}

			public void Reset()
			{
				OnStack = false;
				Label = -1;
				LowLink = -1;
				ConnectedComponent = -1;
			}
		}
```

```
		private Tile[,] worldMap;
		private List<Node> worldGraph;
		private int stoneCount;

		public World(string asciiMap)
		{
			ParseMap(asciiMap);
			CreateGraph();
		}

		private void CreateGraph()
		{
			Node[,] positionToNode = new Node[worldMap.GetLength(0),
worldMap.GetLength(1)];
			worldGraph = new List<Node>(worldMap.Length);
			var currentCombo = new List<Node>();
			var stonePositions = new List<int>();

			//Alle horizontalen Nachbarn finden
			for (int i = 0; i < worldMap.GetLength(0); i++)
			{
				for (int j = 0; j < worldMap.GetLength(1); j++)
				{
					if (worldMap[i, j] == Tile.Exit || worldMap[i, j] ==
Tile.Nothing || worldMap[i, j] == Tile.Stone)
					{
						//Erstelle neuen Knoten für die aktuelle
Postition
						var node = new Node(worldMap[i, j] ==
Tile.Exit, new Tuple<int, int>(j + 1, i + 1), worldGraph.Count);
						positionToNode[i, j] = node;
						currentCombo.Add(node);
						worldGraph.Add(node);

						//Wenn ein YamYam über einen Ausgang
erreicht, dann erreicht es nicht die nächste Wand -> Ausgang ist Combobreaker
						if (worldMap[i, j] == Tile.Exit)
						{
							ProcessCombo(ref currentCombo, ref
stonePositions);
							currentCombo.Add(node);
						}
						else if (worldMap[i, j] == Tile.Stone)
						{
							stonePositions.Add(j + 1);
						}
					}
					else if (currentCombo.Count != 0) //Wand
					{
						ProcessCombo(ref currentCombo, ref
stonePositions);
					}
				}
				ProcessCombo(ref currentCombo, ref stonePositions);
			}

			//Alle vertikalen Nachbarn finden
			for (int j = 0; j < worldMap.GetLength(1); j++)
			{
				for (int i = 0; i < worldMap.GetLength(0); i++)
				{
```

```
                                if (worldMap[i, j] == Tile.Exit || worldMap[i, j] ==
Tile.Nothing || worldMap[i, j] == Tile.Stone)
                                {
                                        //Keinen neuen Knoten erstellen, ist bereits
oben geschehen
                                        var node = positionToNode[i, j];
                                        currentCombo.Add(node);
                                        if (worldMap[i, j] == Tile.Exit)
                                        {
                                                ProcessCombo(ref currentCombo, ref
stonePositions);
                                                currentCombo.Add(node);
                                        }
                                        else if (worldMap[i, j] == Tile.Stone)
                                        {
                                                stonePositions.Add(i + 1);
                                        }
                                }
                                else if (currentCombo.Count != 0) //Wand
                                {
                                        ProcessCombo(ref currentCombo, ref
stonePositions);
                                }
                        }
                        ProcessCombo(ref currentCombo, ref stonePositions);
                }

                //Wenn man ein Ausgangsfeld erreicht, dann kann man es nicht mehr
verlassen -> Alle ausgehenden Kanten von Ausgängen müssen gelöscht werden
                foreach (var node in worldGraph.Where(n => n.IsExit))
                        node.Neighbours.Clear();
        }

        //Fügt eine Kante für alle Knoten in einer Sequenz zum Ende und Anfang
der Sequenz hinzu
        private void ProcessCombo(ref List<Node> currentCombo, ref List<int>
stonePositions)
        {
                if (currentCombo.Count >= 2)
                {
                        bool isX = currentCombo[0].Position.Item1 ==
currentCombo[1].Position.Item1;
                        int stoneCount = 0;
                        foreach (var node in currentCombo)
                        {
                                //Wenn der nächste Stein gefunden wird, dann erhöht
sich die Anzahl an Steinen von der aktuellen Node bis zum rechten Ende
                                if (stoneCount < stonePositions.Count)
                                        if (stonePositions[stoneCount] == (isX ?
node.Position.Item2 : node.Position.Item1))
                                                stoneCount++;

                                if (node != currentCombo.First())
                                {
                                        node.Neighbours.Add(new Tuple<Node,
int>(currentCombo.First(), Math.Max(stoneCount - 1, 0)));
                                }
                                if (node != currentCombo.Last())
                                {
                                        node.Neighbours.Add(new Tuple<Node,
int>(currentCombo.Last(), stonePositions.Count - stoneCount));
                                }
```

```
                        }
                    }
                    stonePositions.Clear();
                    currentCombo.Clear();
                }

                private void ParseMap(string asciiMap)
                {
                    //Zeilenumbrüche entfernen und Dimensionen der Welt berechnen
                    asciiMap = asciiMap.Trim('\r', '\n');
                    int numberOfLines = asciiMap.Count(c => c ==
Environment.NewLine.First()) + 1;
                    asciiMap = asciiMap.Replace(Environment.NewLine, "");
                    int numberOfRows = asciiMap.Length / numberOfLines;

                    //Einzelne Zeichen parse und 2D-Array aufbauen
                    worldMap = new Tile[numberOfLines, numberOfRows];
                    for (int x = 0; x < worldMap.GetLength(0); x++)
                    {
                        for (int y = 0; y < worldMap.GetLength(1); y++)
                        {
                            switch (asciiMap[x * numberOfRows + y])
                            {
                                case ' ':
                                    worldMap[x, y] = Tile.Nothing;
                                    break;
                                case '#':
                                    worldMap[x, y] = Tile.Wall;
                                    break;
                                case 'E':
                                    worldMap[x, y] = Tile.Exit;
                                    break;
                                case 'S':
                                    worldMap[x, y] = Tile.Stone;
                                    stoneCount++;
                                    break;
                                default:
                                    throw new
FormatException($"{asciiMap[x * numberOfRows + y]} ist kein gültiges Zeichen");
                            }
                        }

                    }
                }

                List<List<Node>> connectedComponents = new List<List<Node>>();
                public IEnumerable<Tuple<int, int, int>> Solve()
                {
                    //Graph auf Ausgangszustand zurücksetzen
                    foreach (var node in worldGraph)
                        node.Reset();
                    connectedComponents.Clear();
                    index = -1;

                    //Tarjan für alle schwachen Zusammenhangskomponenten aufrufen
                    foreach (var node in worldGraph)
                        if (node.Label == -1)
                            ConnectedComponent(node.Index);

                    List<bool> componentLeadsToExit = new
List<bool>(connectedComponents.Count);
                    connectedComponents.ForEach(c => componentLeadsToExit.Add(false));
```

```
                    for (int i = 0; i < connectedComponents.Count; i++)
                    {
                            //Prüfen mit welchen Zusammenhangskomponenten die aktuelle
Zusammenhangskomponente verbunden ist
                            bool onlyLeadsToExit = true, hasOut = false;
                            foreach (var node in connectedComponents[i])
                            {
                                    foreach (var neigbour in node.Neighbours.Select(e =>
e.Item1))
                                    {
                                            if (neigbour.ConnectedComponent !=
node.ConnectedComponent)
                                            {
                                                    hasOut = true;
                                                    if
(!componentLeadsToExit[neigbour.ConnectedComponent])
                                                            onlyLeadsToExit = false;
                                            }
                                    }
                            }

                            componentLeadsToExit[i] = onlyLeadsToExit;
                            //Alle Ausgänge als sicher makieren
                            if (!hasOut)
                                    componentLeadsToExit[i] =
connectedComponents[i].Any(n => n.IsExit);
                    }

                    var distanceMatrix = new int[worldGraph.Count, worldGraph.Count];
                    if (stoneCount != 0) //Aus Performancegründen nur ausführen wenn
es auch wirklich Steine gibt
                    {
                            //Matrix mit großen Werten befüllen
                            for (int i = 0; i < worldGraph.Count; i++)
                                    for (int j = 0; j < worldGraph.Count; j++)
                                            distanceMatrix[i, j] = 0xFEFEFEF; //Eine
tolle große Zahl

                            //Jeder Knoten hat eine Distanz von 0 zu sich selbst
                            for (int i = 0; i < worldGraph.Count; i++)
                                    distanceMatrix[i, i] = 0;

                            //Restliche Distanzen eintragen
                            foreach (var node in worldGraph)
                                    foreach (var edge in node.Neighbours)
                                            distanceMatrix[node.Index, edge.Item1.Index]
= edge.Item2;

                            //Weglängen berechnen
                            for (int i = 0; i < worldGraph.Count; i++)
                            {
                                    for (int j = 0; j < worldGraph.Count; j++)
                                    {
                                            if (i == j)
                                                    continue;
                                            for (int k = 0; k < worldGraph.Count; k++)
                                                    distanceMatrix[i, j] =
Math.Min(distanceMatrix[i, j], distanceMatrix[i, k] + distanceMatrix[k, j]);
                                    }
                            }
                    }
```

```
                var distanceToClosestExit = new List<int>(worldGraph.Count);
                for (int i = 0; i < worldGraph.Count; i++)
                        distanceToClosestExit.Add(0);

                for (int i = 0; i < componentLeadsToExit.Count; i++)
                {
                        if (!componentLeadsToExit[i])
                                continue;
                        foreach (var node in connectedComponents[i])
                        {
                                int bestDistanceToExit = 0xFEFEFEF; //Die gleiche
tolle große Zahl wie oben
                                foreach (var exit in worldGraph.Where(n =>
n.IsExit))
                                {
                                        bestDistanceToExit =
Math.Min(bestDistanceToExit, distanceMatrix[node.Index, exit.Index]);
                                }
                                distanceToClosestExit[node.Index] =
bestDistanceToExit;
                        }
                }

                //Wähle alle Komponenten aus, die als sicher makierts sind und
gibt die Postitionen ihrer Knoten aus
                return connectedComponents.Zip(componentLeadsToExit, (c, b) => b ?
c : new List<Node>())
                        .SelectMany(c => c)
                        .Select(n => new Tuple<int, int, int>(n.Position.Item1,
n.Position.Item2, distanceToClosestExit[n.Index]))
                        .OrderBy(n => n);
        }


        //Invarianten für Tarjan
        int index = -1;
        Stack<int> stack = new Stack<int>();
        private void ConnectedComponent(int n)
        {
                //Aktuellen Knoten als besucht markieren
                Node currentNode = worldGraph[n];
                index++;
                currentNode.Label = index;
                currentNode.LowLink = index;
                currentNode.OnStack = true;
                stack.Push(currentNode.Index);

                //Durch Nachbarn iterieren
                foreach (var neighbour in currentNode.Neighbours.Select(e =>
e.Item1))
                {
                        //Unbesuchter Nachbar -> besuchen
                        if (neighbour.Label == -1)
                        {
                                ConnectedComponent(neighbour.Index);
                                currentNode.LowLink = Math.Min(currentNode.LowLink,
neighbour.LowLink);
                        }
                        //Nachbar auf Stack -> Lowlink aktualisieren
                        else if (neighbour.OnStack)
                        {
```

```
                                currentNode.LowLink = Math.Min(currentNode.LowLink,
neighbour.LowLink);
                        }
                    }

                    //Zusammenhangskomponeten gefunden
                    if (currentNode.LowLink == currentNode.Label)
                    {
                        //Zusammenhangskomponenten vom Stack entfernen
                        List<Node> currentComponent = new List<Node>();
                        Node nodeFromStack;
                        do
                        {
                            nodeFromStack = worldGraph[stack.Pop()];
                            nodeFromStack.OnStack = false;
                            nodeFromStack.ConnectedComponent =
connectedComponents.Count;
                            currentComponent.Add(nodeFromStack);
                        } while (nodeFromStack != currentNode);
                        //Und zurückgeben
                        connectedComponents.Add(currentComponent);
                    }
                }
            }
        }
```