Aufgabe 1: Stromrallye

Teilnahme-Id: 52884

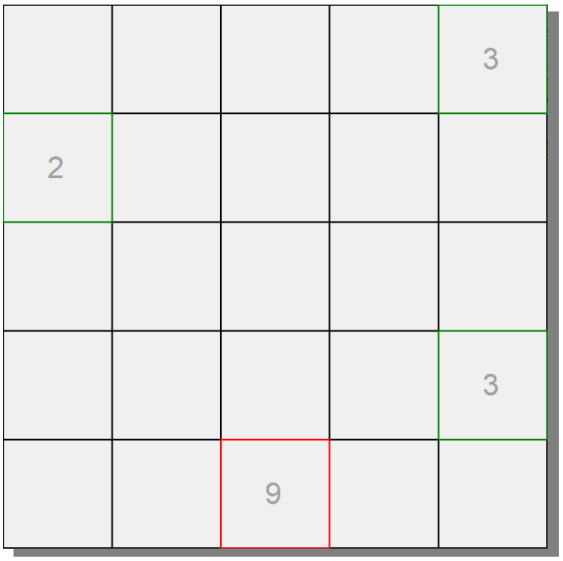
Bearbeiter/-in dieser Aufgabe:
Lasse Friedrich

19. April 2020

Inhaltsverzeichnis

Aufgabenstellung

Es handelt sich um ein Spiel, bei dem sich ein Roboter (Rot), mit einer Batterie auf einem quadratischen Spielbrett mit quadratischen Feldern bewegen kann. Der Roboter kann sich pro Zug nur ein Kästchen nach rechts, links, oben oder unten bewegen und verliert dabei jeweils eine Ladung seiner Batterie. Zusätzlich gibt es Batterien (Grün) auf dem Spielbrett. Diese Batterien besitzen auch eine bestimmte Ladung, welche mit der Ladung des Roboters getauscht wird, sobald er dieses Spielfeld erreicht. Ziel des Spiels ist es alle Batterien, sowie den Roboter auf dem Spielfeld zu leeren, also die Summe aller Ladungen auf null zu bringen.



(a) Lösen

Die erste Aufgabenstellung besteht darin, ein gegebenes Spielbrett mitsamt einem Roboter und Batterien einzulesen und anhand dieser Daten herauszufinden, ob es eine Lösung zu diesem Szenario gibt. Falls möglich, soll eine Schrittfolge ausgeben werden, die den genauen Weg zur Lösung wiedergibt.

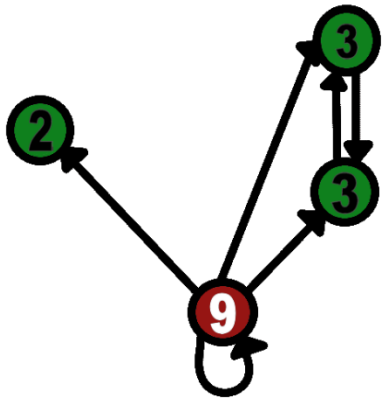
(b) Generieren

Die zweite Aufgabenstellung befasst sich mit der Erstellung solcher Spielszenarien. Die Aufgabe besteht darin Spielbretter zu generieren welche zwar generell lösbar, jedoch gleichzeitig für Menschen schwer zu durchblicken sind. Zudem soll erläutert werden, wie das Schwierigkeitsmaß definiert wird und welche Aspekte ein Spiel schwierig machen.

Lösungsidee

(a) Graphen Umwandlung

Die Grundidee besteht darin, das gegebene Spielbrett in einen Graphen umzuwandeln und einen Weg durch diesen zu finden, bei dem alle Batterien erreicht werden. Die Besonderheit hierbei ist, dass sich der Graph jedes Mal ändert, sobald eine neue Batterie angefahren wird. Das bewirkt das die Ladungen des Roboters und der Batterie getauscht werden und sich danach die Kanten des Graphens aufgrund der Erreichbarkeit ändern. Das führt zu neuen Anfangsvoraussetzungen, mit welchen erneut einen Weg gefunden werden muss. Dadurch bildet sich eine Art Hierarchie, die einem Baum gleicht. Diesen Baum durchsucht man um einen Graphen zu finden, bei dem alle Batterien entladen wurden und sich nur noch der Robobter bewegen müsste. Sollte dies nicht möglich sein, obwohl der ganze Baum schon einmal durchsucht wurde, gibt es keine Lösung.



(b) Kriterienkatalog

Bei der Generierung wird versucht anhand gewisser Kriterien, die in der Umsetzung näher erläutert werden, die Batterien so zu platzieren, dass man möglichst viele Schritte und abstraktes Denkvermögen braucht, um einen Weg durch den Graphen zu finden. Das heißt, dass ein Lösungsweg generiert wird, auf welchem man nun aber noch den Roboter und die Batterien so platzieren muss, dass es ein lösbares Spielbrett wird.

Umsetzung

Die oben beschriebene Lösung wurde in der Programmiersprache C# und den dazugehörigen Standard Bibliotheken, sowie der Grafikoberfläche WinForms umgesetzt.

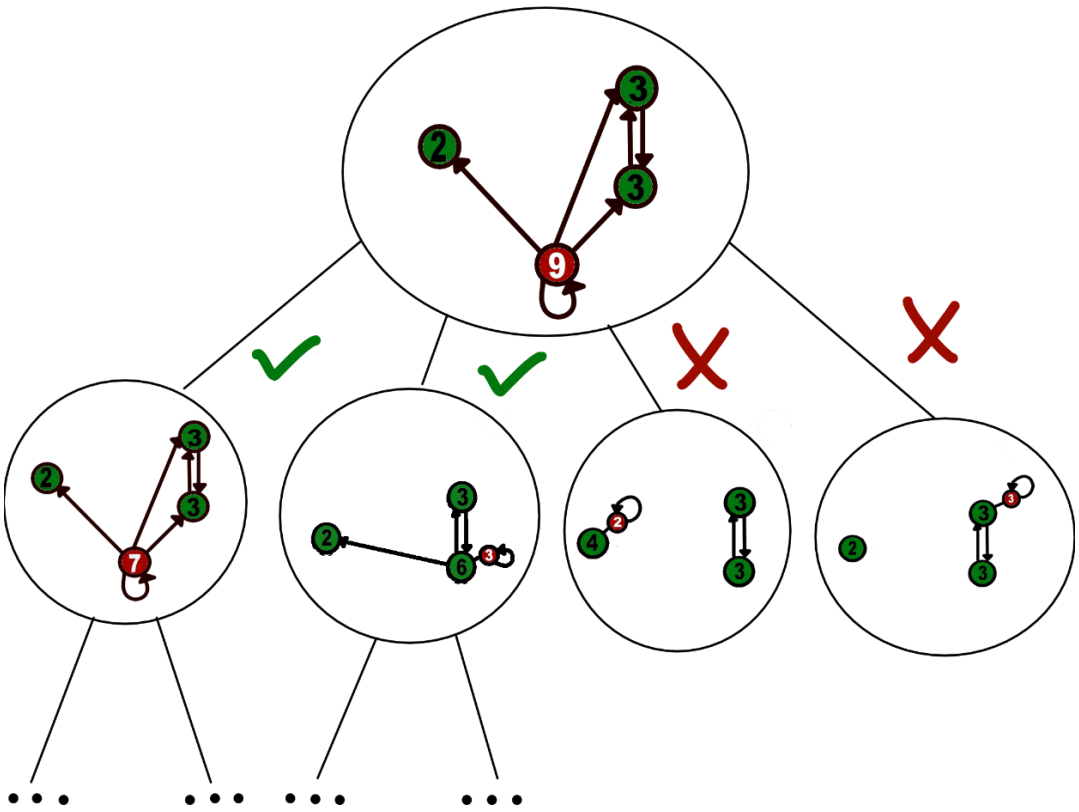
(a) Baumdurchlauf

Nach der Verarbeitung und Umwandlung des Inputs [1], wird damit begonnen die Entfernungen zwischen den einzelnen Batterien untereinander, sowie zum Roboter herausfinden. Diese bleiben durch den ganzen Programmablauf konstant, da sie sich nicht verändern. Hierzu wird ein einfacher Pfadfinde- Algorithmus, der den kürzesten Weg zwischen zwei Punkten herausfindet, benützt. In diesem Fall wurde dazu der „Mazerunner“ [2] Algorithmus verwendet. Bei diesem handelt es sich um eine einfache Breitensuche, bei welcher pro Iteration die vier Adjazenzzellen einer Zelle zu einer Schlange hinzugefügt werden und sich die Distanz zum Ziel um eins erhöht. Der so gefundene Pfad wird zusammen mit der Distanz als Schrittfolge zwischen zwei Punkten, in Form einer Kante hinterlegt. Dabei gilt es zu beachten, dass der Graph stets gerichtet ist und die Schrittfolge in einer Richtung, die der anderen Richtung nicht gleicht. Deshalb wird zum Erstellen des Graphens jede einzelne Batterie mit allen anderen verglichen, sodass man die gerichteten Kanten erhält.

Nachdem alle Kanten definiert wurden, wird nun mit der Berechnung der Nachfolger Graphen begonnen. In dem oben genannten Beispiel gibt es insgesamt, nach der ersten Iteration, vier Nachfolger. Der Kern besteht darin, jede Batterie einzeln vom jetzigen Standpunkt anzufahren. Ein weiteres Szenario, bei dem der Roboter lediglich ein Ausfallschritt zur Seite und wieder zurück macht, wird dann wichtig, wenn er noch mehr Ladung hat als er eigentlich benötigen würde, um zu einer Batterie zu kommen. Er verschwendet durch diesen Schritt zwei Ladung und kann sich somit beliebig entladen, bevor er sich auf eine Batterie stellt.

Zur Optimierung der Rechenzeit können bestimmte Szenarien vorab ausgeschlossen werden. Dazu wird nach jeder Iteration geprüft, ob es Szenarien gibt, die aufgrund ihrer Verkettung sicher nicht mehr lösbar sind. Infolgedessen wird geschaut, ob es in dem jeweiligen Graph einen Weg gibt, auf dem man alle Batterien erreichen kann. Diese Prüfung erfolgt indem die Adjazenzmatrix des Graphens iterativ durchlaufen wird. Das Ziel besteht darin zu schauen ob es eine Position gibt, von der alle Batterien erreicht werden können.

Wenn das nicht der Fall ist, markiert man diesen Graphen als nicht lösbar und ignoriert diesen und seine Nachfolger Graphen. Je früher das passiert, desto mehr Rechenleistung wird insgesamt gespart, weil große Äste des Baumes wegfallen und nicht mehr berechnet werden müssen.



In dieser Grafik gibt es vier von der Wurzel entspringende Szenarien, der Erste Graph wiederspiegelt das Szenario, bei dem der Roboter einen Ausfallschritt macht. Die danach folgenden Graphen repräsentieren die Szenarien, bei denen jede Batterie einzeln angefahren wird. Die so benötigte Rechenleistung wird um ein Vielfaches geringer, weil zwei der vier Szenarien wegen fehlender Erreichbarkeit sofort entfallen. Die zwei verbleibenden Szenarien werden auf einen Stapel gelegt, von welchem in der nächsten Iteration wieder das oberste Element entfernt und von diesem wieder die Nachfolger bestimmt werden. Durch das Benutzen eines Stapels handelt es sich um eine Tiefensuche durch den Baum. Das ist in der Hinsicht nützlich, dass lediglich eine Lösung und nicht alle Lösungen eines Szenarios gefunden werden müssen.

Wenn man durch diese Vorgehensweise einen lösbaren Graphen gefunden hat, gilt es noch die benötigten Schritte anzugeben. Diese Schritte werden durch die sequenzielle Anreihung der Kanten, welche zum finalen Knoten führen, zurückgegeben. Dadurch das jede Kante eine gewisse Schrittfolge hat und erklärt wie man von einem Knoten zum anderen kommt, müssen lediglich all diese Schrittfolgen in einer Liste zusammengefasst und ausgegeben werden.

(b) Zufallsverteilung

Bei der Generierung von Spielbrettern wird mit einem Roboter in der Mitte des Spielfeldes gestartet, welcher mit einer gewissen Ladung zufällige Schritte auf diesem geht. Hierbei markiert er alle schon begangenen Felder als besucht. Wenn nun die Ladung des Roboters leer ist, platziert er eine neue Batterie mit einer erneut zufälligen Ladung. Da er sofort auf dieser Batterie steht, leert er diese während er im Tausch ihre Ladung erhält. Das Platzieren von Batterien geht allerdings nur, wenn er nicht auf einem Feld steht, dass bereits markiert wurde. Falls dies der Fall ist bekommt er und die letzte Batterie, welche er platziert hat, noch eine extra Ladung, mit welcher er einen Schritt zusätzlich machen kann, um dann wieder zu überprüfen ob jetzt eine Batterie hinlegt werden kann. Dieses Vorgehen wird so oft wiederholt, bis die Batteriegrenze, Spielfeld im Quadrat geteilt durch vier, erreicht wurde.

Dadurch das immer ein markierter Pfad freigehalten wird der direkt zur Lösung führt, kann angenommen werden, dass es auch immer eine Lösung gibt. Um das Spiel schwieriger zu gestalten, bestimmen folgende Aspekte das Schwierigkeitsmaß:

1. der Roboter startet immer in der Mitte des Spielfeldes, somit ist es nicht von Anfang an ersichtlich in welche Richtung man starten muss.
2. Das Spielfeld sollte relativ groß sein, um den Überblick zu erschweren.
3. Es sollte relativ viele Batterien geben, da auch das schnell zur Überforderung führt.
4. bestimmte Batterien erhalten eine Ladung von n + 2, das führt dazu das vor dem Anfahren der nächsten Batterie ein Ausfallschritt gemacht werden muss, um Ladung zu verlieren.
5. Viele Ladungswechsel, bei denen der Roboter bereits auf einer Batterie steht, aber mit dieser die Ladung tauschen muss, erschweren das Spielgeschehen.
6. Batterien Ansammlungen, mit vielen Batterien und unterschiedlichen Ladung auf einem Fleck lassen sich nicht intuitiv lösen.

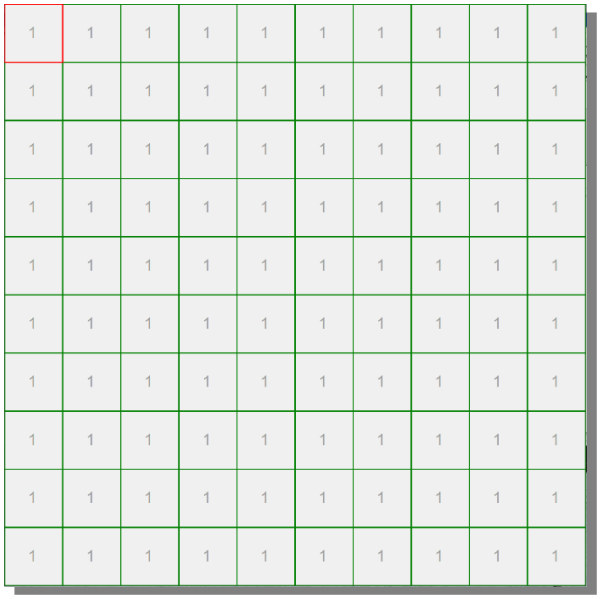
Um das Spiel dann auch für Menschen spielbar zu machen, werden die Pfeiltasten Inputs eingelesen und damit der Roboter um jeweils einen Schritt bewegt.

Beispiele

Im folgendem werden die auf der BWINF Webseite gegebenen Beispiele sowie ein eigens ausgedachtes Beispiel behandelt.

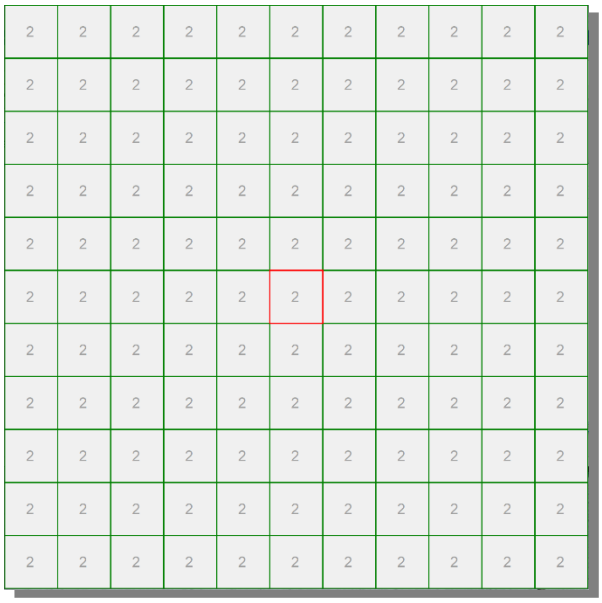
Beispiel 1: siehe Seite 1, ohne Probleme lösbar.

Beispiel 2:



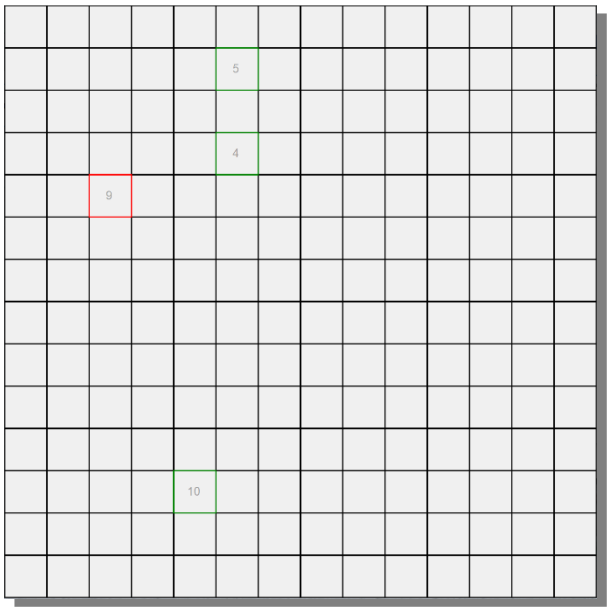
Bei diesem Spielbrett ist auf jedem Feld, eine Batterie mit der Ladung eins. Auch das ist lösbar. Hier kommt das Prinzip der Tiefensuche zu tragen, da hier eine Breitensuche sehr ineffizient ist.

Beispiel 3:



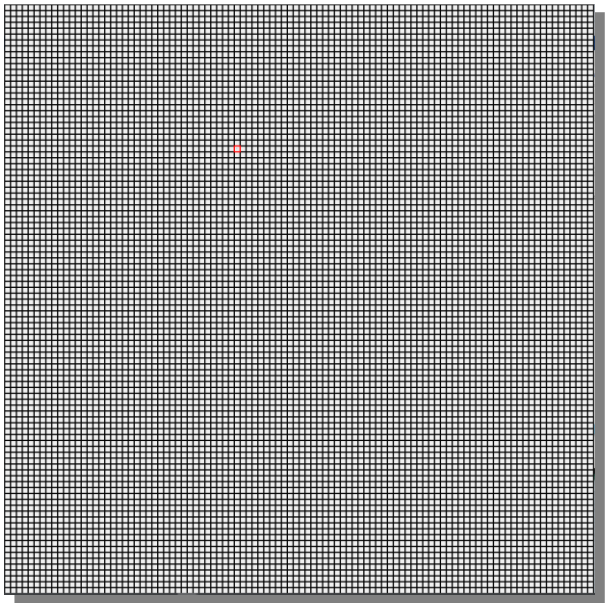
Bei diesem Spielbrett ist auf jedem Feld, eine Batterie mit der Ladung zwei. Auch das ist lösbar, indem zuerst alle Zweien in Einsen und im Anschluss in Nullen umgewandelt werden. Damit das passiert, müssen immer die Batterien mit der höchsten Ladung als nächstes angefahren werden. Sonst kommt es zu einer massiven Erhöhung der Rechenzeit.

Beispiel 4:



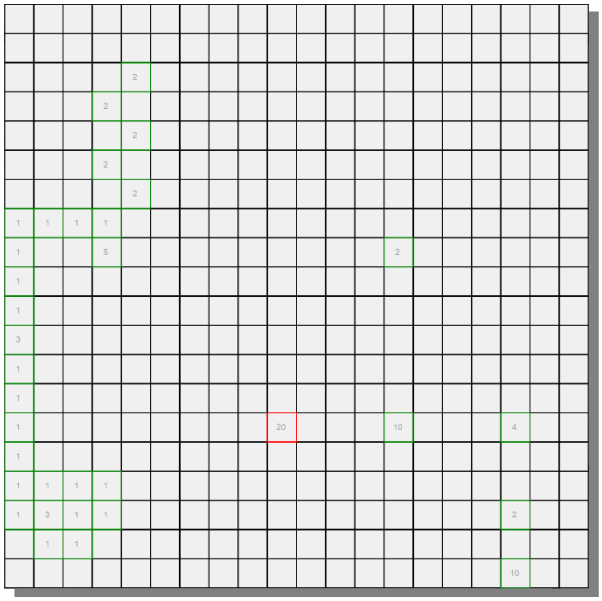
Dieses Spielbrett ist nicht lösbar, da der Roboter kurz vor dem Ende mit der Ladung zwei vor einer Batterie mit der Ladung eins steht. Wenn er nun einen Ausfallschritt machen würde, hätte der Roboter selbst keine Ladung mehr. Auf der anderen Seite könnte er noch auf die Batterie drauf gehen. Hierbei kommt es dann zu dem Szenario, dass der Roboter mit der Ladung eins auf einer Batterie mit der Ladung eins steht. Egal wie er sich jetzt bewegt, er kommt nicht mehr zurück auf die Batterie und kann diese somit nicht mehr entladen.

Beispiel 5:



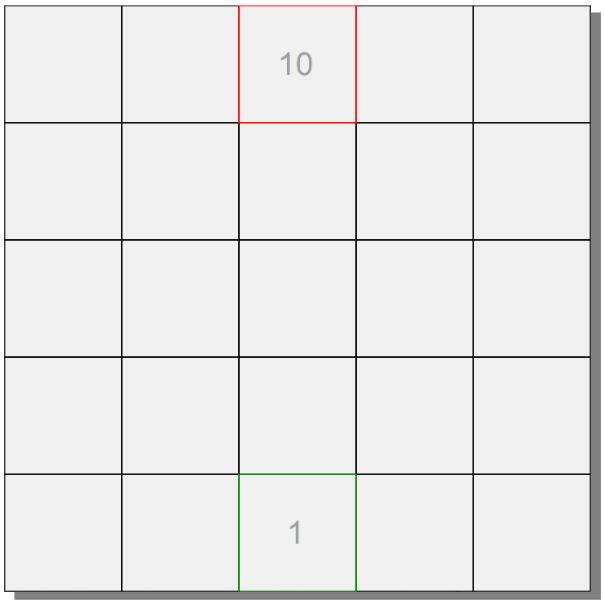
Dieses Spielbrett ist lösbar. Hier gibt es lediglich einen Roboter auf dem Spielfeld, welcher seine Ladung in Form von Ausfallschritten abgeben muss.

Beispiel 6:



Dieses Spielbrett ist nicht lösbar. Hier kommt das oben beschriebene Ausschlussverfahren zur Geltung. Denn egal welche Seite des Spielfeldes zuerst geleert werden wird, es entstehen immer mindestens zwei Subgraphen, welche untereinander nicht mehr erreichbar sind.

Eigenes Beispiel 7:



Dieses Szenario ist lösbar. Hier wird nochmal der Ausfallschritt verdeutlicht. Normalerweise würde der Roboter sofort auf die Batterie gehen, könnte sich dann aber nach dem Ladungstausch nicht ausreichend weiterbewegen. Mit dem Ausfallschritt gibt der Roboter zuerst so viel Ladung ab, dass er die Batterie sofort entlädt, wenn er auf diese geht. Damit er zum Schluss nur noch die eine Ladung der Batterie besitzt, die er mit einem Schritt entladen kann.

Quellcode

```
//findet ein lösbares Szenario indem man eine Tiefensuche vollzieht
 public void SolveDFS()
 {
    //zurzeitige Situation
    Gamestate state = null;

    //Stack zur Tiefensuche
    Stack<Gamestate> s = new Stack<Gamestate>();

    //Liste an Nachfolger einer Situation
    List<Gamestate> neighbours = new List<Gamestate>();

    //fügt die Anfangssituation zu dem Stack hinzu
    s.Push(start_state);

    //testet, ob die Anfangssituation schon fertig ist
    bool isDone = IsDone(start_state);

    //Tiefensuche durch alle Szenarien durch
    while(!isDone && s.Count != 0)
    {
       //nimmt das oberste Element
       state = s.Pop();

       //macht die zurzeitgen Nachfolger leer und fügt die Nachfolger   dieser Situation dem Stapel hinzu
       neighbours.Clear();
       neighbours = GetFollowingStates(state);
       for (int i = 0; i < neighbours.Count; i++)
       {
         s.Push(neighbours[i]);
       }

       //überprüft ob man fertig ist, sonst löscht man dieses Objekt
       isDone = IsDone(state);
       if (!isDone) state = null;
    }

    if (state == null) state = start_state;

    //ruft das Finale Anzeigefenster auf
    box = new Solvable(state, this);

    //setzt die Batterien zurück
    for (int i = 0; i < start_state.batteries.Count; i++)
    {
       all_batteries[i].charge = start_state.batteries[i].charge;
    }

   //falls es eine Lösung gibt, zeigt man die Anfangssituation mit den Bewegungen zur Endsituation an
    if (isDone && LastSteps(state))
    {
       box.IsSolved(true);
```

```
    }
    box.Show();
}



//durchläuft den zurzeitigen Graphen und prüft ob dieser voll begehbar ist
 private bool IsConnected(bool[,] adj, int verteces, int start = 0)
 {
    //Falls es nurnoch den Robobter gibt, gibt es keinen Graphen
    if (verteces == 0) return true;

    //Stapel zur Tiefensuche durch die Adjazenzmatrix
    Stack<int> next_vertex = new Stack<int>();

    //Knoten die schon besucht wurden
    List<int> cur_visited = new List<int>();

    //Schleife durch die gesamte Adjazenzmatrix
    for (int i = 0; i < verteces; i++)
    {
       //löscht die besuchten Knoten für diesen durchlauf
       cur_visited.Clear();

       //packt den Anfangsknoten auf den Stapel
       next_vertex.Push(i);

       //iterativer durchlauf durch den Stapel
       while(next_vertex.Count != 0)
       {
          //nimmt das oberste Element des Stapels
          int current_vertex = next_vertex.Pop();

          //durchsucht alle Nachbarn des obersten Elements
          for (int j = 0; j < verteces; j++)
          {
                //Falls man zu diesem Nachbarn eine Verbindung hat und man diesen noch nicht besucht hat
                fügt man diesen zum Stapel hinzu
             if(adj[current_vertex, j] && !cur_visited.Contains(j))
             {
                next_vertex.Push(j);
             }
          }

                //Fügt das zurzeitige Element der besuchten Knoten liste hinzu, falls dieses nicht schon
                vorhanden ist
          if(!cur_visited.Contains(current_vertex))
          {
             cur_visited.Add(current_vertex);
          }
       }

                //Falls man insgesamt genauso viele Knoten besucht hat, wie es auch gibt, ist der Graph voll
                begehbar
       if (cur_visited.Count == verteces) return true;
    }

    return false;
}
```

```
//geht zufällige Schritte und plaziert Batterien, bis alle Batterien platziert wurden
 void GenerateBatteries()
 {
    //macht do viele Bewegungen bis alle Batterien platziert wurden
    while(all_batteries.Count < batterie_amount)
    {
       MoveDir((Movements)rnd.Next(-2, 3));

                //falls der Roboter keine Ladung hat und man auf dieser Position noch nicht war, platziert man
                eine Batterie
       if (robot.charge == 0)
       {
          if(!visited.Contains(robot.position))
          {
             int next_charge = rnd.Next(batterie_charge_range[0], batterie_charge_range[1]);
             robot.charge = next_charge;
             all_batteries.Add(new StateVector(robot.position, next_charge));
          }
          else
          {
             robot.charge++;
             if(all_batteries.Count != 0) all_batteries[all_batteries.Count - 1].charge++;
          }
       }
       visited.Add(robot.position);
    }

    //erstellt ein Spielbrett mit den gegebenen Batterien
    all_batteries.ForEach(i => map.AddBatterie(i.position.X, i.position.Y, i.charge));
    map.start_state = new Gamestate(map.robot, map.all_batteries, new List<int>());
    map.CreateMatchfield();
    map.GetDistances();

    //zeigt das besagte Spielbrett an
    map.start_state.ShowState();
 }
```

Literaturverzeichnis

[1] Stack Overflow
https://stackoverflow.com/questions/1763613/convert-comma-separated-string-of-ints-to-int-array
(abgerufen am 18.04.2020)

[2] Geeks for Geeks
https://www.geeksforgeeks.org/shortest-path-in-a-binary-maze/
(abgerufen am 18.04.2020)