Aufgabe 1: Stromrallye, Teilaufgabe b)

Teilnahme-Id: 53280

Bearbeiter/-in dieser Aufgabe:
Raphael Gaedtke

2. April 2020

Inhaltsverzeichnis

1 Lösungsidee

1.1 Bestimmung des Schwierigkeitsgrades

Definition 1 *Es sei q mit $q \in \mathbb{N}$ der Schwierigkeitsgrad einer Startposition.*

Die Festlegung des Schwierigkeitsgrades geht von einer Startposition aus, die den Wert $q = 1$ haben soll:

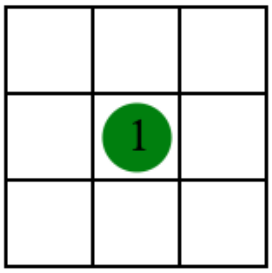

Abbildung 1: Die wohl einfachste aller möglichen Startpositionen

Für diese Startposition gilt

- $k = 3$,
- $s = 1$,
- und $b = 1$,

wobei die Benennung der Variablen wie in der Dokumentation für Teilaufgabe a) erfolgt. Davon ausgehend werden Faktoren gesucht, zu denen q proportional oder antiproportional sein soll:

- q soll zu b und s proportional sein.
- Daneben soll q auch zur Batteriedichte proportional sein. Die „echte“ Batteriedichte $\frac{b}{k^2}$ würde k allerdings zu stark gewichten, da das Brett auch vergrößert werden kann, ohne die Batteriepositionierung und damit die Schwierigkeit erheblich zu beeinträchtigen.
  Deshalb wird ersatzweise die Abbildung $\rho : \mathbb{N} \to \{1, 2, 3, 4, 5\}$ definiert. Diese bildet den Index einer Batterie B_n auf die Anzahl der Batterien ab, die sich auf dem Feld der Batterie B_n und den vier mit einer Kante angrenzenden Nachbarfeldern befinden. In Anlehnung an den graphentheoretischen Begriff soll $\rho(n)$ als der *Nachbarschaftsgrad* einer Batterie B_n bezeichnet werden.
  Zum Durchschnitt der Nachbarschaftsgrade aller Batterien soll q ebenfalls proportional sein.
- Obwohl eine Vergrößerung des Spielbretts ohne ein Verändern der Batteriekonstellation möglich ist, hat k doch auch einen Einfluss auf den Schwierigkeitsgrad, da ein Vergrößern des Spielbretts bei gleichbleibender Batterie- und Ladungszahl wegen der größeren Anzahl an begehbaren Randfeldern tendenziell einfacher zu lösende Startpositionen erzeugt. Deswegen soll q zu k antiproportional sein. Damit die Startposition in Abbildung 1 den Wert $q = 1$ erhält, soll statt k an dieser Stelle der Faktor $\frac{k}{3}$ gewählt werden.
- Um für q einen Wert aus der Menge $\mathbb{N}$ zu erhalten, wird auf das durch die oben beschriebenen Faktoren berechnete Ergebnis noch die Abrundungsfunktion angewendet.

Es ergibt sich also für den Schwierigkeitsgrad:

$$q = \left\lfloor \frac{bs \sum_{i=1}^{b} (\rho(i))}{b\frac{k}{3}} \right\rfloor = \left\lfloor \frac{3s \sum_{i=1}^{b} (\rho(i))}{k} \right\rfloor \tag{1}$$

1.2 Erzeugen einfacher Startpositionen

Zunächst soll ein Algorithmus zum Erzeugen einfacher Startpositionen entworfen werden. Diese Startpositionen können auch dann gelöst werden, wenn jedes Batteriefeld nur einmal betreten wird.
Dieses Kriterium wird bei der Berechnung von q nicht berücksichtigt, in auf diesem Algorithmus aufbauenden Optimierungen sollen Batteriefelder allerdings auch mehrmals besucht werden müssen.

Einfache Startpositionen mit einem vorgegebenen (bzw. per Zufall ermittelten) Gesamtladungswert können erzeugt werden, indem der Roboter (der zu Beginn die gesamte Ladung trägt und beim Laufen keine Ladung verbraucht) immer wieder zwei Aktionen ausführt:

1. Der Roboter platziert auf dem Feld, auf dem er sich derzeit befindet, eine Batterie mit der Ladung n (zu Beginn befindet sich außer der Bordbatterie keine Batterie auf dem Feld).
2. Daraufhin läuft er auf einem Weg mit n Schritten. Dieser Weg darf das Spielbrett nicht verlassen und über keine Felder führen, auf denen sich bereits Batterien befinden.

Das Startfeld des Roboters, die Ladung der abgelegten Batterien und der Weg des Roboters können durch einen Zufallsgenerator gegeben werden.
Dabei kann es allerdings passieren, dass sich der Roboter in eine „Sackgasse“ bewegt. In diesem Fall würde er noch Ladung tragen, könnte allerdings kein Feld mehr betreten.

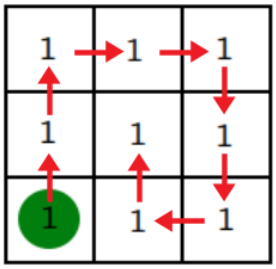

Abbildung 2: Eine Sackgasse für $s = 9$ und $k = 3$

Dann müsste er zur vorletzten Batterie zurückgehen, da von dieser aus auf jeden Fall zwei aneinandergrenzende, nicht von Batterien belegten Felder erreichbar sein müssen. Auf diesem letzten Feld wird dann die gesamte Restladung (zuzüglich der nach dieser Batterie platzierten Ladungen) abgelegt. Können vom letzten Batteriefeld nicht zwei aneinandergrenzende, unbelegte Felder erreicht werden, muss dieser Schritt ebenfalls durchgeführt werden.

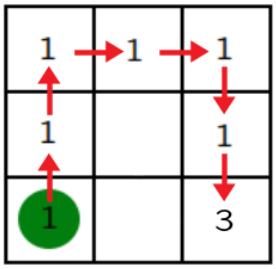

Abbildung 3: Die nach dem oben beschrieben Algorithmus gefundene Korrektur für Abb. 2

Die Lösbarkeit der Startposition ist bei diesem Verfahren immer garantiert, weil der Roboter sich von einem Batteriefeld, auf dem eine Batterie mit einer Ladung von n liegt, mit n Schritten zur nächsten Batterie bewegen kann, wobei die vorige Batterie vollständig entladen wird. Die letzten Ladung kann dabei auch immer verbraucht werden, weil garantiert ist, dass der Roboter von dem Feld, auf dem sich die letzte Ladung befindet, auf zwei weitere, aneinandergrenzende Felder gelangen kann. Auf diesen kann er sich hin- und herbewegen, bis auch die letzte Batterie entladen ist.

1.3 Erhöhen des Schwierigkeitsgrades

Um die Lösung erzeugter Startpositionen für Menschen noch schwieriger zu machen, sollen Batteriefelder im Laufe des obigen Algorithmus auch zweimal betreten werden. Zwischen zwei Besuchen eines Batteriefeldes wird zunächst ein oder kein anderes Batteriefeld erzeugt.
Das dazugehörige Verfahren baut auf dem in Kapitel 1.2 beschriebenen Algorithmus auf. Um das erweiterte Ziel zu erreichen, wird nach jedem Platzieren einer Batterie per Zufall entschieden, ob die nächste Batterie ein- oder zweimal besucht werden soll und ob zwischen diesen zwei Besuchen eine weitere Ladung abgelegt werden soll oder nicht.
Wenn der Roboter das nächste Batteriefeld zweimal betreten möchte, kann er nach dem Platzieren einer Ladung n maximal $n - 1$ Schritte laufen, bevor er die nächste Batterie platziert, die dann zweimal besucht werden soll. Die Anzahl der Schritte, die der Roboter tatsächlich läuft, wird ebenfalls mithile eines Zufallsgenerators bestimmt. Nun gibt es zwei Fälle:
Wenn der Roboter keine weitere Ladung ablegen soll, bevor er das neue Batteriefeld ein zweites Mal betritt, müssen die dort abgelegte Ladung gerade und zwei mit einer Kante aneinander grenzende Felder von dort aus erreichbar sein. Ist dies gegeben, kann der Roboter nach dem Platzieren der Ladung weiterlaufen, bis die n Schritte seit dem Platzieren der nun vorletzten Ladung komplettiert ist (Zwei aufeinanderfolgende Batteriefelder, die doppelt betreten werden müssen, werden auf diese Weise nicht erzeugt).

Soll der Roboter zwischen dem Platzieren und dem zweiten Betreten eines Batteriefeldes noch eine Batterie platzieren, kann die auf dem doppelt besuchten Feld platzierte Ladung auch ungerade sein. In diesem Fall geht er nach dem Platzieren einer Ladung m genau m Schritte und platziert auf dem Feld, auf dem er sich dann befindet, eine Ladung k. Es muss beachtet werden, dass das doppelt zu betretende Feld mit *genau* k Schritten von der Batterie B_k aus erreicht werden kann. Danach kann er wieder vom doppelt besuchten Feld aus die n Schritte für die Batterie B_n komplettieren.
Die Lösbarkeit der Startposition ist bei beiden Verfahren garantiert, da jede der platzierten Batterien auf jeden Fall vollständig entleert wird. Das Entleeren der Batterie mit der Ladung n wird dabei in zwei Schritten durchgeführt: Zunächst läuft der Roboter $n - a$ Schritte, platziert dann eine oder zwei andere Ladungen, und macht in der Lösung dann mit derselben Batterie die restlichen a Schritte.
An jeder Stelle des Verfahrens muss allerdings überpfrüft werden, ob eine weitere Batterie platziert bzw. doppelt besucht werden kann, ohne in eine „Sackgasse“ zu geraten oder die (vorgegebene) Anzahl der Schritte s zu überschreiten. Ist dies der Fall, muss die gesamte Restladung auf dem letzten Feld, von dem aus zwei mit einer Kante aneinandergrenzende Felder erreichbar sind, platziert werden.

2 Umsetzung

Die Lösungsidee wird in C++ umgesetzt.

2.1 Festlegung der Startparameter

Zu Beginn muss das Programm Werte für k, s und die Startposition des Roboters festlegen. Dies soll mithilfe eines Zufallsgenerators erfolgen.
Für s wird dabei 50 als untere und 150 als obere Schranke gewählt. Damit kann ein Mindestmaß an Schwierigkeit garantiert werden, die Startposition nimmt aber auch keine für Menschen zu große Ausmaße an. Es soll also

$$50 \leq s \leq 150 \tag{2}$$

gelten. Jeder der Werte zwischen 50 und 150 wird dabei mit gleicher Wahrscheinlichkeit gewählt.
Die Bestimmung von k orientiert sich am Wert für s, als untere Schranke wird $\lceil\sqrt{s}\rceil$ genutzt. Damit wird garantiert, dass der Roboter theoretisch auch bei jedem Schritt eine Ladung von 1 platzieren könnte, weil dann mindestens s Felder existieren müssen. Als obere Schranke wird willkürlich $\frac{s}{5}$ ausgewählt. k kann also nicht kleiner als $\lceil\sqrt{50})\rceil = 8$ werden, was auch die durch Abb. 1 gegeben untere Schranke von $k = 3$ überschreitet. Es gilt also

$$\lceil\sqrt{s}\rceil \leq k \leq \frac{s}{5}. \tag{3}$$

Auch hier wird jeder mögliche Wert mit gleicher Wahrscheinlichkeit gewählt.
Die Startposition des Roboters wird durch zwei zufällige Koordinaten (eine X- und eine Y-Koordinate) repräsentiert. Diese müssen zwischen 1 und k liegen, auch hier wird jeder Wert mit gleicher Wahrscheinlichkeit bestimmt.
Im Programm wird die Festlegung der Startparameter in der Funktion „starting values()“ durchgeführt, die Werte von s, k und die Startposition des Roboters werden in globalen Variablen gespeichert.
Die Anzahl der Batterien b und die Batteriedichte bzw. der durchschnittliche Nachbarschaftsgrad der Batterien werden nicht zu Beginn ermittelt, sondern durch das -ebenfalls durch einen Zufallsgenerator beeinflusste- Verteilen der Batterien auf dem Feld bestimmt.

2.2 Modellierung des Spielbretts

Das Spielbrett soll auf der Basis einzelner Felder modelliert werden. Dazu wird ein Datentyp „field“ definiert, der als Struktur eine Variable namens „battery“ vom Typ bool und einen Integer namens „charge“ enthält. Die Variable „battery“ wird wahr, wenn sich eine Batterie auf dem angegeben Feld befinden soll, „charge“ speichert dann die Ladung dieser Batterie.
Das Spielbrett selbst wird durch einen zweidimensionalen vector-Container mit dem Namen „game_board“ repräsentiert. Dieser speichert $k \times k$ Variablen vom Typ „field“. Zu Beginn ist die Variable „battery“ für alle Felder auf false gesetzt.
Der Roboter, der, wie in Kapitel 1 beschrieben, über das Spielbrett läuft und Ladungen verteilt, wird selbst nicht als Objekt implementiert, sondern nur durch zwei Koordinaten, die seine aktuelle Position angeben, repräsentiert. Die Programmabschnitte, die Batterien auf dem Feld verteilen, sind daher auch in globale Funktionen und nicht in eine Klasse ausgelagert.

2.3 Zufallszahlen mit gewichteten Wahrscheinlichkeiten

Im Laufe des oben beschriebenen Algorithmus werden Variablen für auf dem Spielbrett zu platzierende Ladungen und die Anzahl zu laufender Schritte per Zufall bestimmt. Für eine solche Zahl d soll dabei

$$1 \leq d \leq g \tag{4}$$

gelten, wobei g eine durch die spätere Verwendung von d gegebene obere Schranke ist (g kann z.B. von der noch verfügbaren Gesamtladungszahl abhängig sein).
Die Werte, die d annehmen kann, sollen dann mit unterschiedlicher Wahrscheinlichkeit gewählt werden. Dazu wird eine zufällige Zahl f in einem Zeilenbereich von 1 bis $1,75 \cdot g$ generiert (Entstehen nichtnatürliche Zahlen, werden diese immer abgerundet).
Wenn $f \leq g$ gilt, ist f das direkte Ergebnis. Gilt $g < f \leq 1,5 \cdot g$, so ist $f - g$ das Ergebnis und für $1,5 \cdot g < f \leq 1,75g$ ist das Ergebnis $f - 1,5 \cdot g$. Dadurch ist das Ergebnis immer kleiner oder gleich der oberen Schranke g.
In diesem Verfahren ist die Wahrscheinlichkeit, gewählt zu werden, für kleinere Zahlen deutlich höher als für größere. Durch die geringere Ladung der einzelnen Batterien auf dem Feld steigt die Anzahl der Batterien b und proportional dazu auch der Schwierigkeitsgrad q.
Im Programm wird dies in der Funktion „weighted_ probabilities()“ umgesetzt.

2.4 Implementierung des Algorithmus

Der in Kapitel 1.2 beschriebene Algorithmus selbst wird in einer while-Schleife umgesetzt, die ausgeführt wird, solange die Variable, in die zu Beginn der Wert von s geschrieben wurde, einen Wert, der größer als 0 ist, speichert.
In der Schleife wird zunächst mithilfe der Funktion „weighted probabitlities()“ die Größe der Ladung bestimmt, die auf dem aktuellen Feld platziert werden soll, woraufhin der Zustand des Containers „game_ board“ angepasst wird.
Danach wird wieder mithilfe der Funktion „weighted_ probabilities()“ entschieden, wie viele Schritte als nächstes gelaufen werden. Ist diese Anzahl größer als 2 (es liegt also ein freies Feld zwischen der aktuellen Batterie und der nächsten), wird ebenfalls per Zufall darüber entschieden, ob die nächste Batterie doppelt besucht werden soll. Dabei werden die drei unterschiedlichen Möglichkeiten mit unterschiedlichen Wahrscheinlichkeiten belegt:

Ereignis	Wahrscheinlichkeit
Das nächste Batteriefeld wird nur einmal betreten.	80%
Das nächste Batteriefeld wird zweimal betreten, dazwischen wird kein weiteres Batteriefeld erzeugt.	10%
Das nächste Batteriefeld wird zweimal betreten, dazwischen wird ein weiteres Batteriefeld erzeugt.	10%

Die gewählte Option entscheidet über den weiteren Programmverlauf.

2.5 Finden eines Weges über das Spielbrett

In jedem Fall muss der Roboter über das Spielbrett laufen ohne Batteriefelder zu betreten oder die Brettgrenzen zu überschreiten. Dafür wird die Funktion „pathfinder()“ implementiert, die für eine als Parameter übergebene Anzahl an Schritten einen Weg mit dieser Anzahl an Schritten ermittelt.
Bei jedem Schritt wird überprüft, welche der bis zu vier an das aktuelle Feld des Roboters angrenzenden Felder noch nicht von einer Batterie belegt sind. Ist die Anzahl dieser Felder größer als 0, wird zufällig eines dieser Felder ausgewählt, dass der Roboter betritt. Von diesem Feld aus wird der Prozess wiederholt bis die entsprechende Anzahl an Schritten gemacht oder das Verfahren wegen einer Sackgasse abgebrochen ist.
Sind jedoch alle an das Feld des Roboters angrenzenden Felder bereits von einer Batterie belegt, befindet sich der Roboter in einer Sackgasse und kann keinen weiteren Schritt mehr machen. Wegen der Existenz dieser Möglichkeit werden zwei Variablen des Datentyps „position“, der eine Position als X- und Y-Koordinate speichert, global gespeichert. Diese enthalten die letzte und die vorletzte Batterieposition und werden bei jedem Platzieren einer Batterie aktualisiert. Befindet sich der Roboter in einer Sackgasse, wird die Restladung auf der vorletzten Batterieposition agelegt und der Prozess der Batterieverteilung abgebrochen. Somit ist sichergestellt, dass die Entladung der Batterie auf dem nunmehr letzten Batteriefeld durch das Hin-und Herwechseln auf zwei benachbarten, leeren Feldern möglich ist.

2.6 Ausgabe

Die durch das Programm erzeugte Startposition soll in eine Datei geschrieben werden, in der sie genau wie in den Beispielen auf der BwInf-Webseite kodiert ist. Das hat auch den Vorteil, dass die Startposition dadurch von dem Programm in Teilaufgabe a) gelöst werden kann.
Im Unterschied zu den Beispielen auf der BwInf-Webseite soll in der letzten Zeile der Ausgabedatei noch ein Wert abgelegt werden, der dem Schwierigkeitsgrad q der in den oberen Zeilen kodierten Startposition entspricht. Dieser wird in der Funktion „difficulty()“ ermittelt.

3 Beispiele

Als Beispiele sollen drei verschiedene Programmausgaben dienen. Weil der Programmfluss stark durch einen Zufallsgenerator beeinflusst wird, können die Ergebnisse mitunter stark voneinander abweichen:

Beispiel 1:

11
2,4,44
5
1,1,1
2,1,4
3,3,14
4,1,4
5,2,2
Schwierigkeitsgrad: 150

Beispiel 2:

11
9,5,28
8
1,2,22
1,5,3
2,2,3
2,3,4
3,2,1
3,5,6
4,2,1
5,2,1
Schwierigkeitsgrad: 357

Beispiel 3:

26
10,5,68
10
1,9,10
1,11,2
2,7,4
2,9,5
2,12,2
3,10,2
7,11,20
10,6,8
11,11,8
14,9,12
Schwierigkeitsgrad: 244

4 Quellcode

Der Hauptteil der Programmlogik wird in der main-Funktion ausgeführt:

```
int main()
{
    int s; //Number of steps, equal to the sum of all charges
    int k; //Length of the board
    int x_robot; //the x-coordinate of the robot
    int y_robot; //the y-coordinate of the robot
    starting_values(&s, &k, &x_robot, &y_robot); //Initialize the values above
    int start_x = x_robot;
    int start_y = y_robot;
    int start_s = s;        //Used for output to file
    int x_actual_battery = x_robot;
    int y_actual_battery = y_robot;
    vector<vector<field>> game_board(k, vector<field> (k));
    init_game_board(&game_board, k); //set the values of the vector above to default values

    position ultimate_battery;
    position penultimate_battery; //These positions are made to handle dead ends
    bool dead_end = false; //True if the robot is in a dead end

    while(s > 0 && !dead_end)
    {
        //In this loop, the batteries are placed on the board
        //Choose the value of the charge which has to the actual field:
        int placed_charge = weighted_probabilities(s);
        //Place the charge on the actual field:
        game_board[x_robot][y_robot].charge = placed_charge;
        game_board[x_robot][y_robot].battery = true;
        s -= placed_charge;
        x_actual_battery = x_robot;
        y_actual_battery = y_robot;
        //How many steps have to be made from now on?
        int steps_to_make = weighted_probabilities(placed_charge-1);
        //How often to visit the next battery field and do we want placing
        //another battery-field between the to visits?
        int event; //0-7 for visit the next battery-field once,
        //8 for visit twice without another battery-field and 9 otherwise

        if(s <= 3 || steps_to_make <= 3)
        {
            //There is not enough charge any more for
            //visiting a battery-field twice or there are not enough steps!
           event = 0;
        }
        else
        {
            time_t t;
            srand((unsigned) time(&t));
            event = rand() % 10; //Values from 0-9
        }

        //Treat the result of the initialization of "event"
        int charge_to_place_next, x_twice, y_twice, charge_to_place_between; //For case 9
        switch(event)
        {
        case 9: //Visit the next battery field twice with another battery between
            pathfinder(&x_robot, &y_robot, &game_board, steps_to_make, &ultimate_battery,
            &penultimate_battery, &dead_end, s, &x_actual_battery, &y_actual_battery, k);
            //Walk to next battery
            charge_to_place_next = weighted_probabilities(s/2);
            //We need enough charge for the way back
            //Place this charge on the actual field:
            game_board[x_robot][y_robot].charge = charge_to_place_next;
            s -= charge_to_place_next;
            game_board[x_robot][y_robot].battery = true;
            x_twice = x_robot;
            y_twice = y_robot;
            //Walk to the next battery between
            pathfinder(&x_robot, &y_robot, &game_board, steps_to_make, &ultimate_battery,
            &penultimate_battery, &dead_end, s, &x_actual_battery, &y_actual_battery, k);
            charge_to_place_between = weighted_probabilities((s - charge_to_place_next)/2);
```

```
71              //Just increase the charge placed on the actual field
                charge_to_place_between *= 2;
73              charge_to_place_between += charge_to_place_next;
                //The difference between the minimal length of the way and the steps has to be even
75              game_board[x_robot][y_robot].charge = charge_to_place_between;
                s -= charge_to_place_between;
77              game_board[x_robot][y_robot].battery = true;
                //Placed a charge on this field
79              x_robot = x_twice;
                y_robot = y_twice; //Go back to last battery
81              pathfinder(&x_robot, &y_robot, &game_board, s - steps_to_make, &ultimate_battery,
                &penultimate_battery, &dead_end, s, &x_actual_battery, &y_actual_battery, k);
83              //Way is completed
                break;
85          case 8: //Visit the next battery field twice without another battery between
                pathfinder(&x_robot, &y_robot, &game_board, steps_to_make, &ultimate_battery,
87              &penultimate_battery, &dead_end, s, &x_actual_battery, &y_actual_battery, k);
                //Walk to next battery
89              //The battery on the actual field has to be visited twice!
                //Place battery:
91              if((x_robot < k-2 && !game_board[x_robot+1][y_robot].battery &&
                !game_board[x_robot+2][y_robot].battery) ||
93                 (x_robot > 1 && !game_board[x_robot-1][y_robot].battery &&
                    !game_board[x_robot-2][y_robot].battery) ||
95                 (y_robot > 1 && !game_board[x_robot][y_robot-1].battery
                   && !game_board[x_robot][y_robot-2].battery) ||
97                 (y_robot < k-2 && !game_board[x_robot][y_robot+1].battery &&
                    !game_board[x_robot][y_robot+2].battery) ||
99                 (x_robot > 0 && y_robot > 0 && !game_board[x_robot-1][y_robot].battery &&
                   !game_board[x_robot-1][y_robot-1].battery) ||
101                (x_robot > 0 && y_robot > 0 &&
                   !game_board[x_robot][y_robot-1].battery &&
103                !game_board[x_robot-1][y_robot-1].battery)  ||
                   (x_robot < k-1 && y_robot > 0 &&
105                !game_board[x_robot+1][y_robot].battery &&
                   !game_board[x_robot+1][y_robot-1].battery) ||
107                (x_robot < k-1 && y_robot > 0 &&
                   !game_board[x_robot][y_robot-1].battery &&
109                !game_board[x_robot+1][y_robot-1].battery) ||
                   (x_robot > 0 && y_robot < k-1 &&
111                !game_board[x_robot-1][y_robot].battery &&
                   !game_board[x_robot-1][y_robot+1].battery) ||
113                (x_robot > 0 && y_robot < k-1 &&
                   !game_board[x_robot][y_robot+1].battery &&
115                !game_board[x_robot-1][y_robot+1].battery) ||
                   (x_robot < k-1 && y_robot < k-1 &&
117                !game_board[x_robot+1][y_robot].battery &&
                   !game_board[x_robot+1][y_robot+1].battery) ||
119                (x_robot < k-1 && y_robot < k-1 &&
                   !game_board[x_robot][y_robot+1].battery &&
121                !game_board[x_robot+1][y_robot+1].battery))
                {
123                 //enough space to discharge the battery
                    int charge_to_place = weighted_probabilities(s/2); //We need an even charge
125                 charge_to_place *= 2; //Now, it is even
                    game_board[x_robot][y_robot].charge = charge_to_place;
127                 s -= charge_to_place;
                    game_board[x_robot][y_robot].battery = true; //Place the battery on the field
129             }
                //Go on!
131             pathfinder(&x_robot, &y_robot, &game_board, placed_charge-steps_to_make,
                &ultimate_battery, &penultimate_battery, &dead_end, s, &x_actual_battery,
133             &y_actual_battery, k);
                break;
135         default: //Do not visit the next battery field twice
                pathfinder(&x_robot, &y_robot, &game_board, placed_charge, &ultimate_battery,
137             &penultimate_battery, &dead_end, s, &x_actual_battery, &y_actual_battery, k);
                break;
139         }
        }
141     //The starting position is evaluated now

143
```

```
    //Output to file:
    ofstream file;
    string name_of_file;
    cout<<endl<<"Bitte␣geben␣Sie␣den␣Namen␣der␣Ausgabedatei␣an:␣";
    cin>>name_of_file;
    file.open(name_of_file, ios_base::out);
    file<<k<<endl;
    file<<start_x+1<<","<<start_y+1<<","<<game_board[start_x][start_y].charge<<endl;

    //count batteries
    int batteries = -1; //-1 for the battery of the robot
    for(int a = 0; a < k; a++)
    {
        for(int b = 0; b < k; b++)
        {
            if(game_board[a][b].battery)
            {
                batteries++;
            }
        }
    }
    file<<batteries<<endl;

    //print batteries
    for(int a = 0; a < k; a++)
    {
        for(int b = 0; b < k; b++)
        {
            if(!(a == start_x && b == start_y) && game_board[a][b].battery)
            {
                file<<a+1<<","<<b+1<<","<<game_board[a][b].charge<<endl;
            }
        }
    }
    int q = difficulty(start_s, k, batteries+1, game_board);
    cout<<"Die␣Startposition␣hat␣einen␣Schwierigkeitsgrad␣von␣"<<q<<endl;
    file<<"Schwierigkeitsgrad:␣"<<q<<endl;
    cout<<"Das␣Schreiben␣in␣die␣Ausgabedatei␣ist␣beendet."<<endl;
    return 0;
}
```