Dokumentation Projektarbeit Carcasonne

Leopold Keller, Laurenz Kammeyer, Frederick Wichert, Jonas Milkovits

Inhaltsverzeichnis

1 Dokumentation - 6.1.4(b)

Die 6.1.4(b) befasst sich mit der Berechnung der Spielpunkte während und am Ende des Spieles.

- **Grundgerüst**

  Als Grundlage für die Punkteberechnung haben wir eine allgemeine Funktion geschrieben, die alle nötigen Randwerte für die Punkteberechnung generiert und diese anschließend je nach übergebenem Parameter entsprechend der Regeln für Burgen, Straßen oder Feldern weiterverarbeitet. Die Punkte werden dem berechtigten Spieler anschließend angerechnet.

  Zuerst wird eine Liste `nodeList` mit allen Kanten des Graphen erstellt. Anschließend wird das erste Element von `nodeList` der Funktion `goToAllConnectedNodes()` übergeben, die eine Liste mit allen mit diesem Knoten verbundenen Knoten des selben Typen erstellt. Sie erzeugt also einen Subgraphen, speziell den Subgraphen der den an `goToAllConnectedNodes()` übergeben Knoten enthält. Anschließend werden alle diese Elemente aus `nodeList` entfernt. Jeder Subgraph wird dann einzeln behandelt. Dieser Prozess wiederholt sich solange, bis `nodeList` leer ist, der Spielfeldgraph also in alle Teilgraphen eines Types zerlegt wurde.

  Der momentan behandelte Subgraph aus `Node<FeatureType>` Elementen wird in der ArrayDeque `queue` gespeichert. Jetzt erden noch zwei weitere Listen befüllt. Einmal die ArrayList< FeatureNode> `connectedFeatureNodes()`, dass alle FeatureNodes des Subgraphen enthält und die ArrayList<Tile> `connectedTiles`, die jede Kachel enthält, die mindestens einen Knoten aus `queue` enthält. Beide werden aufgefüllt indem durch alle Kacheln und ihre `FeatureNodes` iteriert wird und die entsprechenden Elemente herausgefiltert werden.

```
1   ArrayList<Tile> connectedTiles = new ArrayList<>();
2   ArrayList<FeatureNode> connectedFeatureNodes = new ArrayList<>();
3
4   for (int width = 0; width < board.length; width++) {
5    for (int height = 0; height < board[1].length; height++) {
6     for (Node<FeatureType> node : queue) {
7      if (isTileAtPosition(width, height)) {
8       if (board[width][height].getNodes().contains(node)) {
9        for (Position pos : Position.getAllPosition()) {
10        if (board[width][height].getNodeAtPosition(pos) != null) {
11        if (board[width][height].getNodeAtPosition(pos).equals(node)) {
12         connectedFeatureNodes.add(board[width][height].getNodeAtPosition(pos));
13         }
14        }
15       }
16       if (!connectedTiles.contains(board[width][height])) {
17       connectedTiles.add(board[width][height]);
18       }
19      }
20     }
21    }
22   }
23  }
```

- **Gewinner feststellen**

  Nun wird ein IntArray `spieler[]` erstellt, in dem jede Position für eine `MeepleColor` steht. Jetzt werden die Meeple aller Knoten des Subgraphen gezählt und entsprechend ihrer Farbe im Array gezählt. Wir haben uns für diesen Weg entschieden, um die Anzahl der Meeple kompakt speichern zu können. Danach werden alle Meeple ihren Spielern wieder gutgeschrieben.

```
1   for (FeatureNode meepleNode : connectedFeatureNodes) {
2    if (meepleNode.hasMeeple()) {
3     FeatureNodesWithMeeples.add(meepleNode);
4      for (int i = 0; i < 6; i++) {
5       if (MeepleColor.YELLOW == meepleNode.getPlayer().getColor()) {
6        spieler[0]++;
7       }
8       if (MeepleColor.BLACK == meepleNode.getPlayer().getColor()) {
9        spieler[1]++;
10      }
11      if (MeepleColor.BLUE == meepleNode.getPlayer().getColor()) {
12       spieler[2]++;
13      }
14      if (MeepleColor.GREEN == meepleNode.getPlayer().getColor()) {
15       spieler[3]++;
16      }
17      if (MeepleColor.GREY == meepleNode.getPlayer().getColor()) {
18       spieler[4]++;
19      }
20      if (MeepleColor.RED == meepleNode.getPlayer().getColor()) {
21       spieler[5]++;
22      }
23     }
24    }
25   }
26
27   for (FeatureNode node : FeatureNodesWithMeeples) {
28    for (Player play : Players.getPlayers()) {
29     if(node.getPlayer() != null){
30      if (node.getPlayer().equals(play)) {
31       node.setPlayer(null);
32       play.returnMeeple();
33        }
34     }
35    }
36   }
```

  Nun wird der Spieler ermittelt, der die meisten Meeple auf dem Subgraphen hatte und somit Anspruch auf die Punkte hat.

```
1   ArrayList<Integer> highestColor = new ArrayList<>();
2   highestColor.add(0);
3   int compare = spieler[0];  //Vergleichswert fuer evtl gleiche Punktzahlen
4   for (int i = 0;i<6;i++) {
5    if(spieler[i]>compare) {
6     highestColor.set(0, i);
7     compare=spieler[i];
8    }
9   }
10  for(int i = 0;i<6 && i != highestColor.get(0);i++){
11   if(spieler[i] == compare){
12    highestColor.add(i);
13   }
14  }
```

  Im Anschluss wird die Position in `spieler[]` wieder in eine `MeepleColor` zurückübersetzt.

- **Punkte addieren**

  Da jetzt feststeht wer die Punkte erhalten soll, können diese nun berechnet werden. Dies geschieht abhängig von dem übergeben FeatureType `type`. Je nach seinem Wert, aktiviert sich eine der If-Verzweigungen und setzt den zu addierenden `score`. Dessen Berechnung ist dank der bereits erstellten Eckdaten, wie Menge an Kacheln des Subgraphen und Zahl der Kanten vergleichsweise simpel. Eine Hilfsmethode `structureCompleted()` stellt fest, ob ein Subgraph abgeschlossen ist, indem sie überprüft, ob alle relevanten Knoten mit einer weiteren Kachel verbunden sind. Der vorhin erstellte `score` kann jetzt aufaddiert werden.

2 Dokumentation - 6.1.5

1. **Der Graph ist ein azyklischer Graph**

   Die Behauptung ist falsch. Durch das Aneinanderlegen zweier Klosterkarten mit Straße (siehe Abschnitt 3, Abb. 9a) an ihren Straßen, wird ein zyklischer Wiesengraph erzeugt

2. **Gegeben seien k gespielte Legekarten $\{l_1, ..., l_k\}$. Die Anzahl n der Zusammenhangskomponenten im dazugehörigen Graphen ist durch die Ungleichung $n \leq k * 9$ beschränkt**

   Die Behauptung ist wahr. Betrachten wir ein theoretisches Szenario, indem wir auch Karten betrachten, die so nicht im Spiel implementiert sind. Diese theoretischen Karten können beliebig viele verbundene oder nicht verbundene Nodes besitzen. Dies bedeutet, dass die maximale Anzahl an Zusammenhangskomponenten der Anzahl der maximal möglichen unverbundenen Nodes entspricht. Mit der Annahme, dass eine Karte nur maximal 9 Nodes besitzen kann, ist die maximale Anzahl an Zusammenhangskomponenten einer Karte gleich 9. Die maximal mögliche Anzahl an Zusammenhangskomponenten, die durch das Anlegen einer neuen Karte hinzukommen, ist somit auch 9. Nämlich genau dann, wenn eine Karte angelegt wird und beim Anlegen keine der Nodes verbunden werden. Somit ist bewiesen, dass $n > k * 9$ nicht erfüllt sein kann. Gehen wir desweiteren davon aus, dass es möglich ist, dass beim Anlegen Nodes verbunden werden. Durch das Verbinden beim Anlegen wird nicht mehr die volle Anzahl der Zusammenhangskomponenten der angelegten Karte zum Graphen hinzugefügt. Desweiteren kann die Anzahl der Zusammenhangskomponenten einer Karte kleiner gleich 9 sein. Somit ist die Anzahl der Zusammenhangskomponenten bei k gelegten Karten durch $n \leq k * 9$ festgelegt.

3. **Sei der Graph $G = (V, E)$ mit der Menge der Knoten V und der Menge der Kanten E gegeben. Die Komplexität einer Suche nach einem Element in einer Liste legen wir der Einfachheit halber mit $O(1)$ fest. Die Laufzeit der Methode `calculatePoints(FeatureType type, State state)` ist durch $O(|E||V|)$ beschränkt**

   Die Behauptung ist falsch. Obwohl die Zerlegung des Graphen in Subgraphen durch $|E||V|$ beschränkt ist, berücksichtigt dies nicht die zusätzliche Laufzeit, die zum Addieren der Punkte und Auswerten der Meeple benötigt wird. Die Laufzeit fur die gesamte Funktion kann somit höher liegen.

3 Dokumentation - 6.3.1

Die graphische Oberfläche wurde um folgende Elemente erweitert:

- **Tabelle**
  - **Ziel der Spaltengestaltung:**
    1. Möglichst übersichtliche Gestaltung der Spalten
    2. kurze, aussagekräftige Namen
    3. Informationen so übersichtlich wie möglich → wenige Spalten
  - Die Tabelle wurde mit folgendem Code erstellt:

```
1     private JTable scoreTable;
2     List<ScoreEntry> list = resources.getScoreEntries();
3     Object[] column = { "Date", "Name", "Score" };
4     Object[][] data = new Object[list.size()][3];
5     scoreTable = new JTable(data, column);
```

  - Das zweidimensionale Array `data` wurde mit folgendem Code befüllt:

```
1    for (int i = 0; i < list.size(); i++) {
2     if (list.get(i) != null) {
3      data[i][0] = list.get(i).getDate();
4      data[i][1] = list.get(i).getName();
5      data[i][2] = list.get(i).getScore();
6     }
7    }
```

  - **Problem:** Beim Standardmodell editierbare Zellen

    ⇒ **Lösung:** Eigenes Table-Modell

    Das Überschreiben der Methode `isCellEditable(int row, int column)` erlaubt es uns das Editieren der Zellen zu verhindern. Das Überschreiben der zweiten Methode benötigen wir später für das Sortieren der Spalten.

```
1    DefaultTableModel tableModel = new DefaultTableModel(data, column) {
2
3     @Override
4     public boolean isCellEditable(int row, int column) {
5      // all cells false
6      return false;
7     }
8
9     public Class<?> getColumnClass(int columnIndex) {
10     return getValueAt(0, columnIndex).getClass();
11    }
12
13   };
14
15   scoreTable.setModel(tableModel);
```

  - **Ergebnis:**

    Wir haben eine Tabelle mit drei Spalten (Datum, Name, Punkte) erstellt. Das Datumsformat ist absichtlich kürzer gehalten, da unserer Meinung nach der Tag des Spiels ausreichend ist.

- **Scrollbar**
  - Eine Scrollbar wird benötigt, falls zuviele Daten vorhanden sind
  - Die Scrollbar wurde mit folgendem Code erstellt:

```
1    private JScrollPane scrollPane;
2    scrollPane = new JScrollPane(scoreTable);
3    add(scrollPane);
```

  - Dies verknüpft die Tabelle mit der Scrollbar

- **Sortierbarkeit der Spalten**

  Die Sortierbarkeit der Spalten wurde uns hier zwar nicht vorgegeben, war für uns aber ein wichtiges Feature bei einer Highscore-Tabelle. Diese dient auch der einfacheren Verwendbarkeit für den Endnutzer. Alle Spalten sind nach ihren jeweiligen Werten sortierbar.

  - Dies wurde mit folgendem Code erreicht:

```
1    TableRowSorter<TableModel> sorter = new TableRowSorter<TableModel>(scoreTable.
         getModel());
2    scoreTable.setRowSorter(sorter);
```

  Die Vorarbeit, die hierfür benötigt wurde, ist weiter oben in der Erstellung von `DefaultTableModel` zu finden. Dies erlaubt es uns, auch das Datum sowie die Punktzahl zu sortieren. Ohne diese Vorarbeit könnte man das Ganze nur korrekt nach Namen sortieren, da der Autosorter den Wert des Strings vergleicht, was bei dem Datum und der Punktzahl zu falschen Sortierungen führt.

4 Dokumentation - 6.3.2

Eine Wiese soll 3 Bonuspunkte einbringe für jede enthaltene oder angrenzende abgeschlossene Burg. Dazu iterieren wir durch alle Kacheln des Subgraphen durch, um festzustellen welche von ihnen Burgabschnitte enthalten und ob dieser Burgabschnitt bereits besucht wurde. Dies wird mittel einer 'Blacklist' entschieden, der jeder besuchte Knoten einer Burg und die ihrer Zusammenhangskomponente hinzugefügt werden. So wird die Doppelzählung von Burgen vermieden. Was der Subgraph einer Burg ist, wird durch die Hilfsmethode `returnAllConnectedFeatureNodes()` entschieden. Diese Methode ruft wiederum eine eigene Hilfsmethode auf, die sich analog zu `goToAllConnectedNodes()` verhält und konvertiert das Ergebnis anschließend in FeatureNodes analog zu dem Code in **1.**. Weiterhin gibt es die Funktion `allowedTile()`, die Randfälle für besondere Kacheln abdeckt.

Für jede neu besuchte Burg wird `score` um 3 erhöht.

5 Dokumentation - 6.3.3

- **Mission 1:**

  Endet eine Runde in der ein Spieler 3 Burgen mehr, oder mehr, als einer seiner Kontrahenten besetzt, gilt die Bedingung von Mission 1 als erfüllt und das Spiel wird sofort beendet mit diesem Spieler als Gewinner. In diesem Fall soll ein zweites Fenster geöffnet werden, dass Die Namen, Anzahl an besetzten Burgen und Punkte aller Mitspieler enthält.

  Dafür haben wir der Playerklasse einen Parameter `public int castles` hinzugefügt, der die Anzahl an besetzten Burgen eines Spielers enthält. Er wird jedes mal incrementiert, wenn ein Spieler eine neue Burg besetzt. Außerdem haben wir die GamePlay Klasse um die Flag `private boolean castleWin` erweitert, die mit `false` initialisiert wird. Außerdem haben wir die Methode `winByCastle()` in der Gameboard Klasse geschrieben. Diese stellt fest, ob Mission 1 abgeschlossen wurde.

  Diese wird nun in der GamePlay Klasse in der Methode `nextRound()` in einem If-Zweig aufgerufen. Falls sie `true` zurückgibt, wird die `castleWin`-Flag gesetzt und der Spielzustand wir in`State.Game_Over` geändert.

  Unter diesen Bedingungen wird in der `game_Over_Mode()`-Methode eine alternative Schlussnachicht aufgerufen, die in der Klasse `MessagesConstants` unter `showCastleWinner()`beschrieben ist und ein Fenster mit allen in 6.3.3.1 geforderten Informationen öffnet.

- **Mission 2:**

  Die zweite Mission lautet "besitze 3 Kloster und stelle diese fertig". Zu diesem Zweck wurde der `Player.java` ein neues Attribut `public int finishedMonasteries` hinzugefügt, welches in `calculateMonasteries` jedes mal wenn ein Kloster fertiggestellt wird, für den jeweiligen Spieler inkerementiert wird. Nun wird in `Gameboard.java` in der Methode `winByMonasteries` , die in der Methode `nextRound` aufgerufen wird, geprüft, ob ein Spieler aus `getPlayers()` 3 oder mehr fertiggestellte Kloster besitzt. (3 oder mehr da es theoretisch möglich ist 2 Kloster in einer Runde fertigzustellen) Sollte dies der Fall sein gibt die Methode true zurück und die vorher definierte Flagge `private boolean winByMonasteries` wird auf `true` gesetzt. Desweiteren wird der Gamestate `GAME_OVER` gesetzt. In der Methode `game_Over_Mode()` wurde eine extra Verzweigung eingerichtet, die prüft ob ein Sieg durch Klosterbesitz vorliegt. Sollte dies der Fall sein wird eine in `showMonasteriesWinner` in `MessagesConstants.java` definierte Nachricht ausgegeben.

6 Dokumentation - 6.3.4

Der Computergegner besteht aus zwei Methoden:

- `draw(GamePlay gp, Tile tile)`

  Als ersten Schritt suchen wir uns alle möglichen Platziermöglichkeiten.
  Dies geschieht indem wir die Liste der platzierten Tiles des Boards durchlaufen und bei jedem Tile alle Positionen (nördlich, südlich, westlich, östlich) überprüfen und diese, falls das zu platzierende Tile anlegbar ist, abspeichern.
  Dieses Abspeichern geschieht in einer Liste einer selbst erstellten Klasse namens PointRotation, die Koordinaten und die Rotation der Position abspeichern. Danach lassen wir uns eine zufällige Zahl (zwischen 0 und der Länge Liste - 1) ausgeben und verwenden diese zufällige Position dann um an dieser Stelle dann das zu platzierende Tile zu platzieren.

  – Suche der möglichen Platzierungsmöglichkeiten:

```
1   List<Tile> tiles = gc.getGameBoard().getTiles();
2    List<PointRotation> possibleLocations = new ArrayList<PointRotation>();
3
4    for (Tile t : tiles) {
5     // check top
6     for (int r = 0; r < 4; r++) {
7      if (gc.getGameBoard().isTileAllowed(topTile, t.x, t.y - 1) && !
8      gc.getGameBoard().isTileAtPosition(t.x, t.y - 1)) {
9       possibleLocations.add(new PointRotation(t.x, t.y - 1, topTile.getRotation()));
10     }
11     topTile.rotateRight();
12    }
13
14    // check right
15    for (int r = 0; r < 4; r++) {
16     if (gc.getGameBoard().isTileAllowed(topTile, t.x + 1, t.y) && !
17
18     gc.getGameBoard().isTileAtPosition(t.x + 1, t.y)) {
19      possibleLocations.add(new PointRotation(t.x + 1, t.y, topTile.getRotation()));
20     }
21     topTile.rotateRight();
22    }
23
24    // check left
25    for (int r = 0; r < 4; r++) {
26     if (gc.getGameBoard().isTileAllowed(topTile, t.x - 1, t.y) && !
27     gc.getGameBoard().isTileAtPosition(t.x - 1, t.y)) {
28      possibleLocations.add(new PointRotation(t.x - 1, t.y, topTile.getRotation()));
29     }
30     topTile.rotateRight();
31    }
32
33    // check bottom
34    for (int r = 0; r < 4; r++) {
35     if (gc.getGameBoard().isTileAllowed(topTile, t.x, t.y + 1) && !
36     gc.getGameBoard().isTileAtPosition(t.x, t.y + 1)) {
37      possibleLocations.add(new PointRotation(t.x, t.y + 1, topTile.getRotation()));
38     }
39     topTile.rotateRight();
40    }
41   }
```

  – Suche nach zufälliger Zahl:

```
1   Random randomGen = new Random();
2   int randomNum = randomGen.nextInt(possibleLocations.size());
3   PointRotation randomLocation = possibleLocations.get(randomNum);
```

– Rotation der zu platzierenden Tile und Platzieren dieser:

```
1    // Rotate tile to be placed
2    while (topTile.getRotation() != randomLocation.getRotation()) {
3     topTile.rotateRight();
4    }
5
6    // Add rotated tile to GameBoard
7    gc.getGameBoard().newTile(topTile, randomLocation.getX(), randomLocation.getY());
```

- `placeMeeple(GamePlay gp)`

  Das Platzieren des Meeples läuft folgendermaßen ab. Wir speichern mithilfe der Methode `getMeepleSpots()` alle möglichen MeepleSpots des zuletzt platzierten Tiles ab und nutzen diese um uns eine ArrayList des Types Position zu erstellen. In dieser werden alle Positionen gespeichert, an denen ein Meeple platzierbar ist. Wir lassen uns dann wieder eine zufällige Zahl ausgeben (basierend auf der Länge der ArrayListe) und platzieren den Meeple an dieser Stelle, falls die Anzahl der Meeple größer als 0 ist. Falls es keinen verfügbaren MeepleSpot gibt, wird die Methode `nextRound()` aufgerufen, die die momentane Runde beendet.

  – Überprüfung, ob MeepleSpot vorhanden ist:

```
1    if (meepleSpots == null) {
2     gp.nextRound();
3     return;
4    }
```

  – Abspeichern aller möglichen MeepleSpots:

```
1    // boolean array of length 9, true where meeple can be placed on current tile
2    boolean[] meepleSpots = gc.getGameBoard().getMeepleSpots();
3
4    // array with all positions
5    Position[] positions = Position.getAllPosition();
6
7    // Puts all positions with a valid meeple spot into an ArrayList
8    ArrayList<Position> possibleMeepleSpots = new ArrayList<Position>();
9
10   for (int i = 0; i < positions.length; i++) {
11    if (meepleSpots[i] == true) {
12     possibleMeepleSpots.add(positions[i]);
13    }
14   }
```

  – Zufälliger MeepleSpot und Platzieren des Meeples:

```
1    // Getting random meeple spot
2    Random randomGen = new Random();
3    int randomNum = randomGen.nextInt(possibleMeepleSpots.size());
4
5    Position randomMeepleSpot = possibleMeepleSpots.get(randomNum);
6
7    if(meeples > 0) {
8     gp.placeMeeple(randomMeepleSpot);
9    }
10   else {
11    gp.nextRound();
12   }
```