Praktikum 5: Dateiverarbeitung, Algorithmen

Lernziele

- Anwenden von Graphalgorithmen
- Implementieren von Operatorüberladungen

1 Gegner mit „künstlicher Intelligenz“

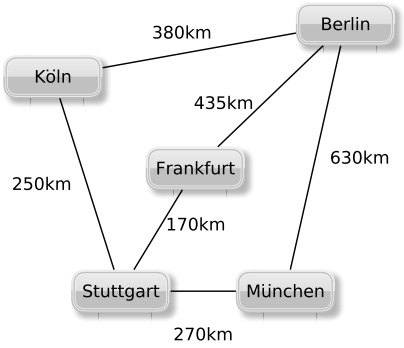
Es soll ein neuer Gegnertyp hinzugefügt werden, der sich in jeder Runde einen Schritt in Richtung des Spielers bewegt und diesen angreift. Dazu implementieren Sie folgendes:

- Eine Methode `DungeonMap::getPathTo(Position from, Position to)` welche einen Pfad zwischen den beiden Kacheln `from` und `to` und diesen zurück liefert. Der Pfad kann dargestellt sein als Container von Positionen (`vector`, `list`, etc.)
- Einen neuen `Controller`-Typ `AttackController`, welcher die `move`-Methode so überlädt, dass dieser in jedem Zug einen Schritt in Richtung des Spielers macht, falls möglich.

Die Frage, wie der Pfad von einer Kachel zu einer anderen aussieht, bzw. ob dieser überhaupt existiert, lässt sich mit einem klassischen Algorithmus aus der Graphentheorie[1] lösen: Dem Algorithmus von Dijkstra[2].

Ein **Graph** besteht aus zwei verschiedenen Mengen: Zum einen eine Menge an **Knoten**, und eine weitere Menge an **Kanten**. Eine Kante verbindet dabei stets zwei Knoten. Eine Kante ist **gerichtet**, wenn Sie nur eine Verbindung in eine Richtung darstellt, und **ungerichtet**, falls die Kante Verbindungen in beide Richtungen darstellt. Eine Kante kann ebenfalls **gewichtet** sein, in diesem Fall ist der Kante ein Wert, manchmal auch **Kosten** genannt, zugeordnet.

Ein „klassisches“ anschauliches Beispiel für einen Graphen wäre ein Streckennetz, welches Städte verbindet. In folgendem Graphen wären die Städte jeweils die Knoten des Graphen und die Kanten sind die möglichen Verbindungen zwischen diesen Städten. Alle Kanten sind bidirektional, also ungerichtet. Darüber hinaus besitzen Sie ein Kantengewicht in Form einer Entfernung.

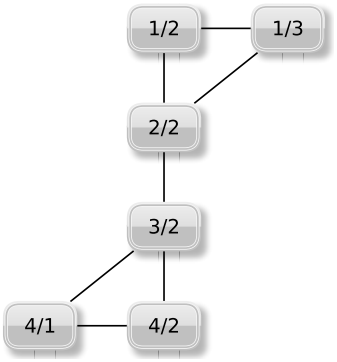


Der Algorithmus von Dijkstra würde für diesen Graphen nun die kürzeste Entfernung zwischen zwei Knoten (=Städten) berechnen, und eine Liste von allen Knoten zurück liefern, die dazwischen zu besuchen wären.

[1] `https://de.wikipedia.org/wiki/Graph_(Graphentheorie)`
[2] `https://de.wikipedia.org/wiki/Dijkstra-Algorithmus` bzw. `https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm` (Ich finde den Pseudocode der englischen Version leichter zu verstehen.)

Auf den Dungeon Crawler übertragen, gäbe es für folgende Karte den neben stehenden Graphen:

```
#####
##..#
##.##
##.##
#..##
#####
```



Die Kosten, also das Kantengewicht, betragen beim Dungeon Crawler immer **eins**, da wir ja ein benachbartes Feld stets mit genau einem Schritt erreichen können.

1.1 Methode `getPathTo`

In dieser Methode müssen Sie zunächst einen Graphen erzeugen, der den aktuellen Stand der Karte wieder gibt. Sie müssen den Graphen vermutlich bei jedem Aufruf der Methode neu erzeugen, da sich die Karte ja verändert haben könnte, beispielsweise durch das Öffnen einer Tür.

Implementierungshinweise:[3] Da Sie jeweils eine **Menge** für die Knoten und Kanten benötigen, bietet es sich an, den einer Menge entsprechenden Container aus der STL zu verwenden: Das `std::set`[4]. Ähnlich wie `vector` hat ein `set` einen Templateparameter, nämlich den zu speichernden Datentyp. Ich empfehle, ein `set<Position>` als Menge der Knoten zu verwenden und dort die Koordinaten aller Kacheln abzuspeichern, die betreten werden können. Unter Umständen müssen Sie für die Eigenschaft „ist betretbar“ eine virtuelle Methode in `Tile` implementieren. Ein `set` legt Elemente stets sortiert und einmalig ab. Elemente werden durch die Methode `insert` eingefügt, bei der Sie die Position nicht bestimmen können. Ist ein Element, welches Sie gerade einfügen wollen bereits im `set` enthalten, so macht `insert` einfach nichts.
Damit die Sortierung und auch der Vergleich auf Gleichheit funktioniert, müssen Sie in `Position` den $<$ -Operator überladen. Ich schlage vor, dass Sie diesen so implementieren, dass zuerst nach Reihe und danach nach Spalte zu sortieren. **Beispiel:** 2/4 ist kleiner als 3/4, 2/4 ist kleiner als 2/5.
Für die Kanten definieren Sie am Besten ein `struct`, welches zwei Positionen enthält, beispielsweise als Attribute `knoten1` und `knoten2`. Bedenken Sie, dass Kanten bidirektional sein sollen, also eine Kante von A nach B auch gleichzeitig eine Verbindung von B nach A darstellt. Dazu könnten Sie im Konstruktor einer Kante sicher stellen, dass `knoten1` stets die kleinere Position enthält. Damit wäre dann die Kante $A \leftrightarrow B$ identisch zur Kante $B \leftrightarrow A$ und würde damit nicht doppelt in Ihrem `set` abgespeichert.
Darüber hinaus benötigt eine Kante ebenfalls noch den $<$ -Operator, damit das `set` damit umgehen kann.

Anschließend müssen Sie auf dem bestehenden Graphen noch den Dijkstra-Algorithmus implementieren und entsprechend die Knotenmenge zurück geben. Die Knotenmenge müsste, wenn Sie richtig implementiert haben, auch den Start- und Endknoten enthalten. Wenn kein Weg zwischen den Knoten existiert, können Sie einfach eine leere Menge zurück geben. **Hinweis:** In Bezug auf den Pseudocode auf Wikipedia: Es gibt keinen Weg, wenn die Menge `Q` keinen Knoten mehr enthält, dessen Distanz kleiner als `unendlich` ist.

Bedenken Sie auch, falls Sie eine fertige Implementierung übernehmen, den Ursprung der Lösung zu kennzeichnen!

Implementierungshinweise: Für den Dijkstra benötigen Sie zwei Datenstrukturen, die in den Pseudecodebeispielen häufig als Arrays oder Vektoren erscheinen. Diese speichern für jeden Knoten die kumulierte Distanz bzw. den Vorgängerknoten. Ich empfehle hierfür, dass Sie sich die Datenstruktur `std::map`[5] ansehen. Eine `map` speichert Paare von **Key** und **Value**, wobei die `map` bezüglich der Keys funktioniert wie ein `set`, und zu jedem Key ein Value mit gespeichert werden kann. Sie können sich das vorstellen wie ein Array, welches im Prinzip einer Zahl, nämlich dem

[3] Der folgende Text stellt wirklich nur Hinweise dar. Falls Sie eine andere, eventuell bessere, Idee haben die funktioniert, dann sind Sie natürlich völlig frei, diese zu implementieren!
[4] `http://www.cplusplus.com/reference/set/set/`
[5] `http://www.cplusplus.com/reference/map/map/`

Index, einen bestimmten Wert zuordnet, nämlich das an diesem Index gespeicherte Objekt.
Eine `map` überlädt praktischerweise den `[ ]` -Operator, so dass Sie eine `map` quasi als Array verwenden können, bei dem der Index (=der Key) nicht zwangsläufig ein `int` ist. In diesem Beispiel wäre der Key dann ein Objekt von `Position` (=ein Knoten).

1.2 `AttackController`

Die Klasse benötigt im Gegensatz zu den anderen `Controller`-Klassen deutlich mehr Informationen über die Spielwelt. Sie benötigen einen Zeiger auf `DungeonMap`, auf den zu attackierenden Spieler und auf die Figur, die durch diesen `AttackController` gesteuert wird.

Den `DungeonMap*` benötigen Sie, um die gerade implementierte Methode zu verwenden. Die beiden `Character*` benötigen Sie, um die jeweils aktuellen Positionen zu ermitteln.

Wenn es einen Pfad gibt, Sie also eine nicht-leere Menge an Positionen von `getPathTo` zurück bekommen, dann können Sie einen Schritt näher an den Gegner heran machen. Hierzu müssten Sie die Position an Index `1` der Pfadliste noch in eine Bewegungsrichtung umrechnen (im Prinzip die umgekehrte Operation, die Sie schon in der `turn`-Methode machen) und diese Richtung zurück geben.

Klarstellung: Der Unterschied zwischen „Sichtlinie“ und „Pfad“ ist, dass die Sichtlinie wirklich nur eine Gerade ist, während das oben erwähnte Verfahren tatsächlich auch „Um die Ecke“ geht.

Zusatzaufgabe: Implementieren Sie den `AttackController` so, dass dieser erst anfängt zu laufen, wenn die Spielerfigur in das Sichtfeld des Gegners gekommen ist.

2 Operatoren

Implementieren Sie die geforderten Operatoren und stellen Sie sicher, dass Sie diese in Ihrem Programm auch durchgehend verwenden. Beispielsweise sollten Sie prüfen, ob der Operator aus Unterabschnitt 2.1 nicht im Parser bzw. beim Einlesen der Datei verwendet werden kann.

2.1 Eingabeoperator >> für `Position`

2.2 Ausgabeoperator << für zwei beliebige Klassen

3 Kampf zwischen zwei Spielfiguren

Implementieren Sie einen Kampf zwischen zwei Spielfiguren. Eine Kampfrunde soll dadurch ausgelöst werden, dass eine Figur auf ein bereits durch eine andere Figur besetztes Feld zieht.

In einer Kampfrunde fügt zunächst der angreifende Spieler dem angegriffenen Spieler Schaden in Höhe seiner Stärke (incl. der Boni durch Items) zu. Ist der angegriffene Spieler danach noch am Leben (HP > 0), so schlägt dieser zurück. Anschließend ist die Kampfrunde beendet.

Erweitern Sie die `turn`-Methode so, dass diese zu Beginn jedes Spielerzuges prüft, ob eine Figur tot ist, also weniger als einen Lebenspunkt besitzt. In diesem Fall wird die Figur vom Feld genommen und für die Figur wird kein Spielzug mehr durchgeführt.
Hinweis: Sie könnten beispielsweise die Figur löschen (`delete`) und den Zeiger aus dem Container mit den `Character*` entfernen. Damit sollte sicher gestellt sein, dass diese Figur nicht mehr „dran“ kommt.

Fügen Sie eine weitere Überprüfung ein, dass das Spiel abbricht sobald kein „menschlicher“ Spieler mehr aktiv ist. Ein menschlicher Spieler ist dadurch definiert, dass dieser durch einen `ConsoleController` gesteuert wird.

Hinweis: Sie sollten auch „Mehrspielerrunden“ unterstützen können. Es soll also möglich sein, zwei (oder mehr) Spieler zu haben, die durch einen `ConsoleController` gesteuert werden.