Projektgruppe: 036
Abgabedatum: 26.03.2019

TECHNISCHE UNIVERSITÄT DARMSTADT

FOP-Projektarbeit

Aufgabe 3.1.2- Kanten bilden und überprüfen – Weiterführende Aufgabe

Um die Knoten miteinander zu verbinden nutzen wir eine Hilfsfunktion namens „findNearestCastles“, welche die euklidische Norm eines Punktes ermittelt und nutzt um die drei Burgen mit dem kürzesten Abstand zu unserer gewählten zu liefern.
Die Methode „generateEdges“ nutzt die Liste der Hilfsmethode um zu diesen Knoten entsprechende Kanten zu erzeugen. Die Hilfsmethode wird für jeden existierenden Knoten aufgerufen, sodass die Burgen adäquat miteinander vernetzt werden und dennoch das Risiko von überschneidenden Kanten minimiert wird.

Um zu überprüfen, ob alle Knoten miteinander verbunden sind traversieren wir mit Hilfe einer „for-Schleife“ die Liste aller verfügbaren Kanten und entfernen jeweils den Start – und den Zielknoten von der zuvor zwischengespeicherten Liste aller Knoten. Falls die Methode isEmpty(), angewandt auf die Knotenliste wahr (true) zurückliefert, so sind alle Knoten miteinander verbunden. Andernfalls ist dies nicht der Fall.

Aufgabe 3.1.4 – Theorie

a) Da die Knoten in einer LinkedList gespeichert werden, beträgt die Laufzeit zur Suche nach einem bestimmten Knoten im „Worst-Case“ O(n).
In diesem Fall können n Knoten n Nachbarn besitzen. Nach jedem Durchlauf wird jedoch auch ein Element von der Methode „getSmallestNode()“ aus der Liste entfernt.
Die Komplexität im „Worst-Case“ lässt sich somit durch den kleinen Gauß, also n*(n-1)/2 beschreiben, wobei n die Anzahl der Knoten bezeichnet.
Da n*(n-1)/2 <= n^2 |*2 |-n^2 ⇔ n <= n^2 |: n ⇔ 1<=n ist.
Somit ist O(n^2) eine obere Schranke unserer Methode, da für n0=1 alle Funktionswerte von n^2 größer sind als die des kleinen Gauß.
Da Zuweisungen und direkte Zugriffe konstante Laufzeit besitzen, d.h. von O(1) sind, sind diese für die Wahl der oberen Schranke nicht relevant.

b) Aktuell wird eine „LinkedList“ zum Speichern der Knoten verwendet. Sofern weiterhin eine das List Interface implementierende Klasse als Datenspeicher gewollt ist, würde sich eine „ArrayList“ anbieten, da diese Datenstruktur schnellere Zugriffszeiten ermöglicht, sofern es sich um einen direkten Zugriff (und keine Suche) handelt.
Andernfalls würde es sich anbieten eine eigene Datenstruktur, wie einen B-Tree zu implementieren, welcher sich selbst durch entsprechende Rotation balanciert.
Die Zugriffszeit würde im „Worst-Case“ somit O(log(n)) betragen, um ein Element zu finden.
Eine Datenstruktur, welche eine baumähnliche Implementation besitzt ist das TreeSet, welches für Suchen ebenfalls eine Zeitkomplexität von O(log(n)) aufweist

c) Schleifeninvariante:

Nach h>=0 Durchläufen gilt:
-Die Liste enthält die noch nicht abgearbeiteten Knoten und verfügt über n-h Knoten (size() = n-h), da nach jedem Durchlauf der ausgewählte Knoten aus der Liste entfernt wird. (mit n,h e N)

-Die Knoten, welche noch nicht abgearbeitet wurden werden durch die Liste referenziert. Zudem besitzen Knoten, welche nicht bearbeitet wurden einen symbolischen Wert (-1) oder einen suboptimalen Wert (d.h. der Wert ist ggf. nicht der kleinstmögliche)

-Knoten, welche abgearbeitet wurden sind nicht länger in der Liste (durch diese referenziert). Zudem wird ihnen ein neuer Knotenwert zugewiesen, falls der ursprüngliche Wert -1 (d.h. nicht gesetzt) oder der neue Wert kleiner als der ursprüngliche Wert ist.

Aufgabe 3.1.5 - Theorie

a)
Der Algorithmus würde nicht wie beabsichtigt funktionieren, da ein negativer Zirkel dafür sorgen würde, dass die Summe der Kantengewichte
nach jedem Durchlauf reduziert wird, weshalb es zu einer Endlosschleife
kommen könnte.
Da höchstens h+1 Kanten vorhanden sein dürfen (h aber bei n-1 abbricht)
könnte jeder weitere Durchlauf die Kantensumme reduzieren. Somit liegt
nicht der kürzeste Abstand vor (dieser würde nach jedem Durchlauf durch einen negativen Zirkel kleiner).

Anmerkung:
Für Kantengewichte k>=0 kann der Pfad nicht durch das addieren von
weitere Kanten optimiert werden. Somit führt die Addition der lokal
kleinsten Kantengewichte zum global optimalen Pfad.
Existieren beispielsweise zwei Verbindungen von A nach C, wobei eine direkt
mit Kantengewicht =2 und eine indirekt (über Knoten B) existiert mit Kantengewicht 5, so wird die direkte Verbindung genommen, da der "Umweg" bei Kantengewichten k>=0
nicht kleiner als 2 sein kann. Negative Kantengewichte würden diese
Invariante verletzen, weshalb der Algorithmus nicht für negative
Kantengewichte anwendbar ist.

Der beschriebe Algorithmus zur Pfadfindung weist zudem Gemeinsamkeiten mit dem Dijkstra-Algorithmus auf, da dieser ebenfalls die lokal kürzesten Pfade wählt um den optimalen Pfad zum jeweiligen Zielknoten zu finden. Dessen Invariante würde bereits durch einen negativen Pfad verletzt werden, aufgrund der bereits genannten Aspekte.

b)
Es handelt sich um eine nxn-Matrix, weshalb im schlimmsten Fall n Knoten auch n Nachbarknoten (bzw. Pfade zu diesen) haben können. Der Zugriff auf die Matrix selbst, sofern es sich um ein Array handelt, ist konstant und kann somit vernachlässigt werden. Die genannte Konstruktion lässt somit darauf schließen, dass Theta(c*n^2) vorliegt im „Worst-Case“, wobei c ein beliebiger Koeffizient ist.
Im „Best-Case“-Szenario lässt sich die Laufzeit durch Theta(c*n) ausdrücken, d.h. n verschiedene Knoten besitzen jeweils einen Nachbarn.

Aufgabe 3.3.2 – Neue Missionen

Die erste Mission, welche von uns implementiert wurde trägt den Namen „Need For Speed“. Dabei ist der Name auch Programm, denn abhängig von der Kartengröße gewinnt der Spieler, welcher zuerst 1000, 2000 bzw. 5000 (bei kleiner/moderater/großer Kartengröße) erreicht. Diese stellt einen komplementären Spielmodus zur bereits implementierten Mission „Eroberung“ dar, denn anders als bei diesem handelt es sich bei „Need For Speed“ um einen schnelleren (oder auch „fast-paced“) Spielmodus.

Auch bei unserer zweiten Mission „Time Race“ ist Geschwindigkeit ein relevanter Faktor, denn die Siegbedingung sieht vor, dass der Spieler, welcher innerhalb der ersten zwei Minuten die meisten Burgen erobert und hält, das Spiel gewinnt.
Anders als bei unserer ersten Mission ist es hier somit länger wichtig die meisten Punkte, beispielsweise durch eine defensive Spielstrategie zu erreichen, sondern Spieler werden für aggressive Spielzüge belohnt, wenn sie dadurch die meisten Burgen erobern können.

Die dritte Mission gestaltet das Spiel dynamischer, da bei dieser Mission eine von drei möglichen Siegbedingungen erfüllt werden muss, um als Sieger hervorzugehen. Es ist möglich entweder durch die Eroberung aller gegnerischer Burgen zu gewinnen, als erster Spieler 2000 Punkte zu erreichen oder für die nächsten zwei Minuten die meisten Burgen zu halten. Um die erste Siegbedingung festzustellen wird der Besitzer jeder Burg ausgelesen. Falls alle Burgen einem Spieler gehören, so tritt die Siegbedingung ein. Die zweite Siegbedingung wird mit Hilfe der Methode „getPoints“ festgestellt. Ferner wird die dritte mögliche Siegbedingung mit der Methode „getPlayerWithMostCastles()“ geprüft, indem diese nachdem die Zeit abgelaufen ist aufgerufen wird.

Aufgabe 3.3.3 – Joker

Der Joker ist nach einem verlorenen Kampf vom jeweiligen Spieler über den Button unten rechts im Spielmenü auswählbar. Dieser erhält die Bezeichnung „Joker“ anstelle von „Überspringen“ während eines Zeitfensters von 5 Sekunden. Aktiviert ein Angreifer seinen Joker, so wird eine Niederlage abgewendet und der Joker wird aufgebraucht.
Aktiviert hingegen der verteidigende Spieler seinen Joker um die Niederlage abzuwenden, so endet kann der Angreifer in diesem Spielzug nicht länger angreifen.
Dieser Joker ist insbesondere in den neu implementierten Missionen relevant, da er beispielsweise in Mission „Time-Race“ über Sieg oder Niederlage entscheiden kann, wenn im passenden Moment der Joker aktiviert und die Siegbedingung damit nach Ablauf der Zeit erfüllt wird.