Dokumentation

Projektgruppe 065

Gruppenmitglieder:

- *Yusuf Emre Berber*
- *Christoph Lukas Berst*
- *Jan Kai Braun*
- *Julian Imhof*

3.1.2 Verbinden der Burgen (Kanten bilden)

Für jede Burg wird zuerst eine Liste erstellt, die in aufsteigender Reihenfolge sortiert, nach ihrem Abstand zur der aktuell betrachteten Burg, alle anderen Burgen enthält. Damit die Linien zwischen den Burgen nicht irgendwie kreuz und quer entstehen, wird die Burg mit bis zu 4 ihrer nächsten Burgen (meistens zwischen 2 und 4 Nachbarburgen) verbunden. Dies geschieht mit ein paar Einschränkungen:
Wenn die Burg schon mit einer Nachbarburg verbunden ist, wird stattdessen eine andere Burg als Partner gesucht, damit die Burg immer genügend Verbindungen zu anderen Burgen hat. Dies soll das Spiel ausgeglichen halten. Rein lineare Kämpfe zwischen zwei Gebieten haben schon in Risiko dafür gesorgt, dass es Regionen gab, die man nur über einen einzigen Punkt mit Truppen zu stopfen konnte, um eine allumfassende Verteidigung herzustellen, ähnlich wie eine Flasche an ihrem Hals mit einem einfachen Korken. Damit jedoch Gebiete nicht zu viele Angriffsvektoren besitzen, wird einer Burg keine weitere Angriffsmöglichkeit hinzugefügt, sobald sie das vorgesehene Maximum (4 andere Burgen) an Nachbarn besitzt.

3.1.3 Iteratoren und Wege finden I

1/3 Es wird durch alle *availableNodes* gegangen und so in einem durchlauf die im Werk kleinste, nicht negative korrespondierende *AlgorithmNode* gefunden und am Ende zurückgegeben. Wurde keine gefunden, wird wie in der Aufgabenstellung verlangt *null* geliefert.
2/3 Es wird jeden Knoten der *availableNodes* verwendet, um den neuen Wert aller Nachbarknoten zu bestimmen. Der neue Wert wird allerdings nur übernommen, sollte der Weg zum benachbarten Knoten passierbar und der Knoten selbst entweder noch keinen Wert oder einen größeren als den Neuen haben. Sind diese Bedingungen erfüllt, wird auch der *Vorgängerknoten* des benachbarten Knoten gleich dem aktuellen Knoten gesetzt.
3/3 Es wird ein vier Elementiger Iterator gebaut, der aus dem Hauptknoten, dessen p*revious Node* und den jeweils korrespondierenden *AlogorithmNodes* besteht. In jedem Iterationsschritt, wird der Hauptknoten auf die entsprechende *previous Node* gesetzt und alle anderen Elemente angepasst. Des Weiteren wird jeweils die Kante zwischen den Knoten einer Liste hinzugefügt, welche am Ende umgedreht und zurückgegeben wird.

Erstellt von: *Yusuf Emre Berber*, *Christoph Lukas Berst*, *Jan Kai Braun* und *Julian Imhof*

3.1.4 Wege finden II

A. $f \in O(n^2 * n)$

Man gehe von dem Fall aus, dass in einem Graphen jeder Knoten mit jedem anderen Knoten und sich selbst verbunden sei:
Die äußere Schleife wird im wesentlichen durch getSmallestNode() bestimmt, welches sich durch das Entfernen der Elemente nach dem Suchen wie Selectionsort verhält, welches quadratisch wächst. Da nun jeder Knoten mit Jedem verbunden ist, durchläuft die innere Schleife auch n Kanten. Die Bedingungen ob etwas passierbar ist, hat nur eine geringe Auswirkung auf die Laufzeit und kann daher vernachlässigt werden.

B. $f \in O(T(n) * n)$

$T(n)$ hängt von der List, welche in getSmallestNode() verwendet wird ab
Wenn man nun eine sortierte Liste wie eine SortedList oder eine PriorityQueue verwenden würde, bei der sogar ein geschickt gewählter Comparator verwendet wird, welcher die negativen Zahlen ans Ende der Liste einsortiert, wäre der Worst-Case nur noch $O(n)$, da man am ersten Element schon erkennen kann, ob das gesuchte Element vorhanden ist.

C. Invariante: Nach $h \geq 0$ Durchläufen und der Datenmenge n gilt:
   a. Liste von Knoten, deren Wert kleiner 0 ist, und maximal $(n - h)$ Knoten, welche in den vorherigen h Schritten nicht den kleinsten Wert hatten.
   b. Knoten, deren Wert kleiner 0 ist oder Knoten, welche mindestens den $h+1$ niedrigstens Wert besitzen, der nicht negativ ist.
   c. h Knoten, deren Wert definitiv positiv ist und welche nicht mehr in der Liste “availableNodes” vorhanden sind

3.1.5 Kürzester Pfad zu allen Knoten

[...]

3.3.4 Anpassung für Studierende im Studiengang CE

http://codeflow.org/entries/2010/nov/29/verlet-collision-with-impulse-preservation/
https://math.stackexchange.com/questions/1472049/check-if-a-point-is-inside-a-rectangular-shaped-area-3d
https://stackoverflow.com/questions/5666222/3d-line-plane-intersection

Änderungen der Vorlage (außerhalb der Aufgaben)

- Angreifen sollte nur möglich sein, wenn mindestens eine Truppe weniger ausgewählt wurde, als auf der angreifenden Burg vorhanden ist
    - game.Game#startAttack(Castle, Castle, int)
    - gui.components.MapPanel#mouseClicked(MouseEvent)
    - game/players/BasicAI#actions(Game)
- Wenn man nicht mit allen Truppen angreifen wollte, wurde der Kampf trotzdem so lange geführt, als hätte man mit allen Truppen angreifen wollen
    - gui.AttackThread#run()
- Würfel werden im Chat in absteigender Reihenfolge sortiert
    - gui.views.GameView#onAddScore(Player, int)
- Wenn ein Angriff fehlschlägt und man die nächste Runde starten will, wurde eine NullPointerException geworfen
    - gui.components.MapPanel#clearSelection()
- KI konnte eigene Truppen durch Feindburgen verschieben
  (und ich habe noch ein paar Kleinigkeiten ausprobiert)
    - game/players/RandomAI#actions(Game)
- nützliche Methoden
    - base.Graph#getEdge(Node, Node)
    - base.Graph#getNodes(Node)

Erstellt von: *Yusuf Emre Berber*, *Christoph Lukas Berst*, *Jan Kai Braun* und *Julian Imhof*