Smart Cars (NAME IN PROGRESS)

Rami und Christoph

June 24, 2017

1 Theoretisches Modell und Vorgehensweise

Im Zentrum des Projektes steht das Verhalten von intelligent gerouteten Autos mit Hilfe von Routingtables zu simulieren. Dabei werden etliche Idealisierungen und Abstraktion getroffen, um das Problem auf die zentralen Aspekte zu reduzieren(bei zusätzlicher Zeit kann überlegt werden, ein paar Aspekte noch einzuarbeiten):

- Jeder Knotenpunkt, bei dem sich mehr als zwei Kanten befinden (also irgendwie 2 Straßen dranhängen) hat eine Ampel
- Eine Straße erstreckt sich immer nur zwischen zwei Knoten, danach beginnt eine neue.
- Jede Straße hat nur eine Spur und die Richtung ist klar definiert (da gerichterter Graph).
- Autos haben die gleiche Geschwindigkeit. Des Weiteren wird ihre Position nicht explizit ausgerechnet, demzufolge haben sie auch keinen "Körper", global festgelegter abstand zwischen Autos existiert
- Autos haben keine Beschleunigung (außgenommen von der implziiten Beschleunigung durch den Durchsatz der Ampel)
- Autos werden nur an Spawnpunkten generiert, wobei ein Spawnpunkt immer ein Keypoint ist.
- Autos werden nur an Keypoints destroyed.
- Nur Autos die an den Spawnpunkten generiert werden, werden bei den Berechnungen und dem Modell betrachtet
- Es existieren keine außergewönhlichen Events wie Bauarbeiten oder Änderung der Straßen

1.1 Theoretisches Modell und Vorgehnsweise

Die Grundidee ist es einen gerichten Graphen zu haben (dieser wird durch Openstreetmap genriert -> natürlich noch überarbeitet). Jeder Randpunkt des Graphen ist ein sogennante KeyPoint, also ein Knoten der auf jeden Fall in der Routing-table enthalten sein muss. Denn interessant ist unsere Anwendung vor allem für Anwendungen, in denen das ziel des Autos außerhalb des eigentlichen Graphens liegt und man den optimalen Weg sucht, um den Graphen zu "verlassen" und dabei relativ nah an dem Zielpunkt zu sein.
Anwendungsbeispiel: Für den Stadtteil Stellingen möchte ermittelt werden, wie Autos die von außerhalb Stellingen kommen und zu einem beliebigen Ort in der Innenstadt fahren möchten, den Stadtteil möglichst optimal durchqueren.
Möglichst optimal heißt hier, dass die Durchschnittszeit aller Autos zum durchschreiten des Stadtteils minimiert wird.
Wir berechnen eine Routing Table in der für jeden Keypoint zu jedem Keypoint die kürzeste Strecke berechnet wird. Dies geschieht mit dem A* Algorithmus der für jedes Knotenpaar der Keypoint (optimierung aufgrund symmetrie einbezogen) durchgeführt wird. **Parallelsierung**: Das kann man wohl so parallelisieren, dass ein Master Prozess Paare aus Start- und Endknoten möglichst gleichmäßig an alle anderen Prozesse aufteilt. Es entsteht also am Ende eine Matrix, zwischen Start und Endknoten, die symmetrisch ist(Da jeder Endknoten ja auch startknoten in die andere Richtung ist) in der die optimalen Pfade gespeichert sind.

Als nächsten Schritt würde man dann die Autos auf der Karte simulieren. Hierbei beginnt ein Auto bei einem Startknoten und hat einen Zielpunkt, der auch außerhalb der Zielknoten liegen kann. Von diesem Zielpunkt aus würde man nach Zielknoten suchen, die möglichst nah an dem Zielknoten dran sind.(Das sind die Zielknoten die für die Berechnung in Frage kommen). Als nächstes berechnet man welche der Strecken die beste ist. Da man die Zielknoten kennt, die in frage kommen, kann man sich die optimalen Pfade dank der Routing table raussuchen und die Kosten berechnen.(Auch hier hat jeder Knoten eine Array die jeder Zeit das Gewicht speichert). Hat man sich für eine Strecke entschieden, aktualisiert man die Gewichte und packt die in den richten ArraySlot der Kanten. Unsere Lösung berechnet also nur den Weg für ein Auto am Anfang, da jedoch die Kantengewichte aktualisiert werden, sollte das Auto trotzdem Staus umgehen. **Unser Programm berechnet also nicht immer die optimale Strecke, sollte aber signifikant dazu beitragen Staus zu verhindern.**. Momentan würden ja auch viele Strecken garnicht in betracht gezogen werden, da sie in keiner Routingtable als optimale Strecke gespeichert sind. Falls man dann noch Zeit haben sollte, könnte man überlegen, nicht nur die Optimale Route zu berechnen, sondern die besten n routen(für die Routing tables: Also eine n zu 1 beziehung Strecken zu (Knoten, Knoten)) und von den n routen für jeden Ausgang

immer die beste wählen. Dadurch werden alle Straßen gleichmäßiger verteilt. (Da es ja durchaus vorkommen kann, dass alternative Pfade nicht beachtet werden).
Parallelisieren: Könnte man das, indem man alle Spawnpoints(oder Keypoints?!?!) gleichmäßig auf die Prozesse verteilt, die dann dafür zuständig sind, die Gewichte der Strecken zu updaten. Man könnte weitere Prozesse verwenden die die Karte in Sektoren einteilen, die dafür sorgen, dass die Position der Autos aktualisiert wird. Das würde man wohl aus einer Mischung MPI Openmp lösen. (Das Separieren ist möglicherweise nicht soo sinnvoll. Andernfalls könnten die Prozesse die sowieso die Gewichte der Kanten aktualisieren auch die Bewegung der Autos steuern, dann müsste man aber doppelte Zugriffe blocken (Optimale Wege können ja gleiche Kanten beinhalten). Das Blockieren könnte durch eine Flag in der Kante realisiert werden. ODER: Jedes Auto merkt sich, welcher Prozess ihn gespawned hat und dieser hat die komplette Zeit über die Bewegung die Kontrolle, damit würde man das Problem lösen, da da keine Überschneidungen vorhanden sind. (Natürlich muss in diesem Fall viele Abfragen über das Auto ablaufen).

Vorgehensweise: Erstellung der Routingtables(OpenStreetmap export to Graph, A* lib einbinden, Routingtables parallelisiert generieren) Modellierung der Autos auf dem Graphen (Queue, Kosten über Zeit für Kanten, Kostenaktuallisierung, Parallelisierung der Kostenaktuallisierung und Autobewegung) Erweiterung der Routingtables auf n Routen pro paar Weitere Sachen, die vielleicht unschön sind.

1.2 genauere Erläuterungen

TODO QUEUE beschreiben
UML eingehen
TIMETABLE beschreiben