A* Algorithmus

Leonard Bauer

März 2024

Inhaltsverzeichnis

Kapitel 1

Die Idee des A*-Algorithmus

1.1 Ursprung

Die Idee hinter dem A*-Algorithmus entstand aus dem Bestreben, eine Steuerung für mobile Roboter zu entwickeln, die ihre eigenen Aktionen planen können.

1.2 Grundprinzip

Der A*-Algorithmus basiert auf zwei Hauptprinzipien:

- Informierte Suche: Der Algorithmus nutzt eine Heuristik, um den Abstand zwischen dem aktuellen Knoten und dem Zielknoten abzuschätzen. Diese Heuristik lenkt die Suche in Richtung des Ziels und ermöglicht es dem Algorithmus, effizienter zu sein als uninformierte Suchalgorithmen.
- Best-First-Search: Der Algorithmus wählt immer den Knoten mit der niedrigsten Gesamtbewertung als nächsten zu bearbeitenden Knoten. Die Gesamtbewertung setzt sich aus den tatsächlichen Kosten, um zum aktuellen Knoten zu gelangen, und der geschätzten Kosten, um vom aktuellen Knoten zum Ziel zu gelangen, zusammen.

1.3 Vorteile

Der A*-Algorithmus hat mehrere Vorteile:

- Optimalität: Der Algorithmus findet garantiert den kürzesten Pfad zwischen zwei Knoten, sofern eine solche Lösung existiert.

- Effizienz: Die Verwendung einer Heuristik ermöglicht es dem Algorithmus, den Suchraum schnell zu durchsuchen und den Pfad in kurzer Zeit zu finden.
- Flexibilität: Der Algorithmus kann mit unterschiedlichen Umgebungen und Hindernissen umgehen.
- Robustheit: Der Algorithmus ist auch bei ungenauen Sensorinformationen und unvorhergesehenen Ereignissen funktionsfähig.

1.4 Anwendungsgebiete

Der A*-Algorithmus findet in einer Vielzahl von Anwendungsgebieten Anwendung, z. B.:

- Robotik: Der Algorithmus wird zur Pfadplanung für mobile Roboter eingesetzt.
- Navigationssysteme: Der Algorithmus wird zur Berechnung von Routen in Navigationssystemen verwendet.
- Spieleentwicklung: Der Algorithmus wird in der Spieleentwicklung verwendet, um die Bewegung von Spielfiguren zu steuern.
- Künstliche Intelligenz: Der Algorithmus wird in der KI-Forschung eingesetzt, um Probleme wie Pfadplanung und Suchprobleme zu lösen.

1.5 Zusammenfassung

Zusammenfassend lässt sich sagen, dass der A*-Algorithmus ein effizienter, flexibler und robuster Algorithmus zur Suche nach dem kürzesten Pfad zwischen zwei Knoten in einem Graphen ist.

Kapitel 2

Funktionsweise des A*-Algorithmus

Der A*-Algorithmus durchsucht einen Graphen, um den kürzesten Pfad zwischen einem Startknoten s und einem Zielknoten t zu finden. Dabei verwendet er zwei Hauptprinzipien:

2.1 Suchalgorithmen

2.1.1 Informierte Suche

Der Algorithmus nutzt eine Heuristik $h(n)$ um den Abstand zwischen dem aktuellen Knoten n und dem Zielknoten t abzuschätzen. Diese Heuristik lenkt die Suche in Richtung des Ziels und ermöglicht es dem Algorithmus, effizienter zu sein als uninformierte Suchalgorithmen. Der A*-Algorithmus basiert auf einem Gewichteten Graphen.

2.1.2 Best-First-Search

Der Algorithmus wählt immer den Knoten n mit der niedrigsten Gesamtbewertung $f(n)$ als nächsten zu bearbeitenden Knoten. Die Gesamtbewertung setzt sich aus zwei Teilen zusammen:

- $g(n)$: Die tatsächlichen Kosten, um vom Startknoten s zum aktuellen Knoten n zu gelangen.
- $h(n)$: Die geschätzten Kosten, um vom aktuellen Knoten n zum Zielknoten t zu gelangen.

2.2 Schrittabfolge

1. Der Algorithmus initialisiert den Startknoten s mit einer Gesamtbewertung von $f(s) = g(s) + h(s)$.
2. Der Algorithmus fügt den Startknoten s in eine Open-Set-Liste ein.
3. Solange die Open-Set-Liste nicht leer ist:
   - Der Algorithmus entfernt den Knoten n mit der niedrigsten Gesamtbewertung $f(n)$ aus der Open-Set-Liste und fügt ihn in eine Closed-Set-Liste ein.
   - Für alle Nachbarknoten m des aktuellen Knoten n:
     - Berechne die tatsächlichen Kosten $g(m)$ um vom Startknoten s zum Nachbarknoten m zu gelangen.
     - Berechne die geschätzten Kosten $h(m)$ um vom Nachbarknoten m zum Zielknoten t zu gelangen.
     - Berechne die Gesamtbewertung $f(m) = g(m) + h(m)$ des Nachbarknotens m.
     - Wenn der Nachbarknoten m nicht in der Open-Set-Liste $OPEN$ ist, füge ihn mit seiner Gesamtbewertung $f(m)$ in die Open-Set-Liste $OPEN$ ein.

Kapitel 3

Anwendungszwecke für den A*-Algorithmus

3.1 Routenplanung

Navigationssysteme: A* ist die Grundlage für viele Navigationssysteme in Autos und auf Smartphones. Es berechnet die schnellste Route zwischen einem Start- und Zielpunkt unter Berücksichtigung von Straßenverkehr, Verkehrsregeln und anderen Faktoren.

Flugplanung: Fluggesellschaften nutzen A*, um optimale Flugrouten zu finden, die Treibstoffverbrauch und Flugzeit minimieren.

Roboternavigation: A* ermöglicht Robotern, sich in ihrer Umgebung zu bewegen und Hindernissen auszuweichen, um ihre Ziele zu erreichen.

3.2 Spielentwicklung

Wegfindung in Spielen: In Computerspielen wird A* verwendet, um die Wegfindung von Spielfiguren und NPCs zu steuern. So können sie sich in der Spielwelt effizient bewegen und auf ihre Ziele zugehen.

KI-Gegner: A* kann verwendet werden, um KI-Gegner in Spielen zu entwickeln, die intelligent navigieren und optimale Entscheidungen treffen können.

Levelgenerierung: A* kann in der Levelgenerierung eingesetzt werden, um automatisch Level zu erstellen, die bestimmte Anforderungen erfüllen, z.B. eine bestimmte Länge oder Schwierigkeit.

3.3 Optimierungsprobleme

Aufgabenplanung: A* kann verwendet werden, um die optimale Reihenfolge von Aufgaben in einem Projekt zu finden.

Ressourcenzuweisung: A* kann helfen, Ressourcen effizient zuzuordnen, um ein bestimmtes Ziel zu erreichen.

Netzwerkoptimierung: A* kann in der Netzwerkoptimierung eingesetzt werden, um den optimalen Weg für Daten oder Güter durch ein Netzwerk zu finden.

3.4 Weitere Anwendungsgebiete

Suchmaschinen: A* kann verwendet werden, um die optimale Reihenfolge von Suchanfragen zu finden, um die relevantesten Ergebnisse zu finden.

Bioinformatik: A* kann in der Bioinformatik eingesetzt werden, um z.B. die optimale Route für DNA-Sequenzierung zu finden.

Kapitel 4

Eigenschaften des A*-Algorithmus

A* hat viele eigenschaften, wegen denen er weit verbreitet ist.

4.1 Vollständigkeit

A* wird immer eine Lösung finden falls eine existiert, und ween der Graph endlich ist und es keine unendlichen kosten gibt.

4.2 Optimalität

A* garentiert den kürzesten Weg zu dem Ziel zu finden, wenn bestimmte Kondition erfüllt sind, zum beispiel ein zuverlässiges Heuristisches verfahren zu haben, das niemals die Kosten überschaätzt.

4.3 Effizienz

Auch wenn A* keine Garantie für die Zeitkomplexität im schlimmsten Fall bietet, schneidet es in der Praxis aufgrund seiner heuristischen Führung und der Auslese weniger vielversprechender Pfade oft gut ab.

4.4 Zulässigkeit

A* erfordert eine zulässige Heuristik, um Optimalität zu gewährleisten. Eine zulässige Heuristik überschätzt niemals die Kosten für das Erreichen des Ziels, d.h. die tatsächlichen Kosten für das Erreichen des Ziels werden niemals höher sein als die von der Heuristik geschätzten Kosten.

4.5 Speicher Benutzung

A* erfordert in der Regel die Speicherung aller besuchten Knoten, was in Umgebungen mit begrenztem Speicherplatz oder bei großen Graphen ein Problem darstellen kann. Verschiedene Optimierungen, wie die Verwendung einer Prioritätswarteschlange, können dieses Problem jedoch entschärfen.

Kapitel 5

Nachteile A*-Algorithmus

5.1 Komplexität

Die Laufzeit des A*-Algorithmus kann exponentiell mit der Größe des Suchraums wachsen, insbesondere wenn der Suchraum sehr groß oder unübersichtlich ist. Dies kann zu längeren Berechnungszeiten führen, insbesondere wenn der Graph stark verzweigt ist.

5.2 Speicherbedarf

A* benötigt Speicherplatz, um die bereits besuchten Knoten und diejenigen, die noch besucht werden müssen, zu verfolgen. Dies kann bei großen Suchräumen zu einem erheblichen Speicherbedarf führen, insbesondere wenn der Algorithmus viele Knoten im Speicher behalten muss.

5.3 Heuristikabhängigkeit

Die Effizienz des A*-Algorithmus hängt stark von der Qualität der Heuristik ab. Eine schlechte Heuristik kann dazu führen, dass der Algorithmus unnötig viele Knoten erkundet oder sogar zu einer suboptimalen Lösung führt.

5.4 Verzerrung (Bias)

Wenn die Heuristik nicht konsistent ist, kann der A*-Algorithmus zu einer Verzerrung führen, was bedeutet, dass er die kürzeste oder optimale Lösung möglicherweise nicht findet. Dies kann insbesondere bei ungleichmäßig verteilten Kosten im Graphen auftreten.

5.5 Nicht für dynamische Umgebungen geeignet

A* ist nicht gut geeignet für dynamische Umgebungen, in denen sich die Kosten oder Hindernisse während der Suche ändern können. Da A* nur einmal eine Schätzung der Gesamtkosten berechnet, kann sich dies als ineffizient erweisen, wenn sich die Umgebung während der Suche ändert.

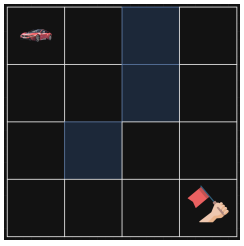
Kapitel 6

Angewandtes Beispiel

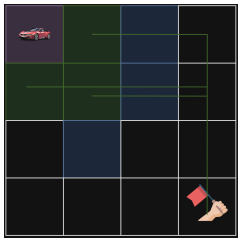
Die Kosten Berechnung setzt sich aus $F = G + H$
G ist dabei Kosten vom Start und H sind die Kosten zum Ziel.
Hier ist ein Beispiel das das Pathfinding von einem Auto zum Ziel Behandelt.

- Gesperrte Zelle (Closed List)
- Hinderniss
- Offene Zelle (Open List)

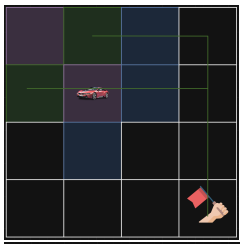
6.1 Ablauf



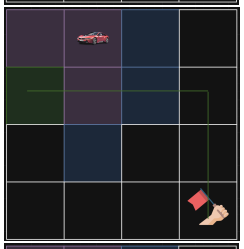
Die Karte wird mit einem Start, einem Ziel und Hindernissen initialisiert



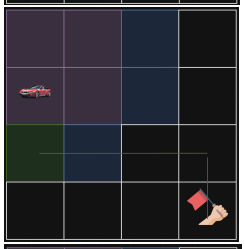
Bei jedem Durchlauf wird das aktuelle Feld zur gesperrten Liste hinzugefügt.
Dann wird die ausgewählte Heuristik ausgewählt und damit der Weg von Start bis Ziel geschätzt grüne Linien. Dann werden die Feld Kosten berechnet mit der Formel $F = G + H$.
Das Diagonal unten ist das Günstigste deswegen ist dies der nächste Schritt.



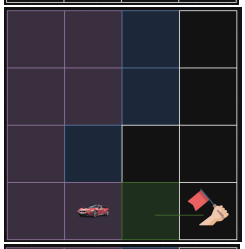
Das aktuelle Feld wird wieder Gesperrt da die anderen beiden Felder schon zur Open List hinzugefügt wurden müssen nicht nochmal die Kosten berechnet werden.



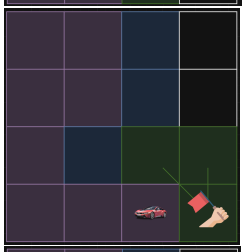
Wenn eine Deadlock Sitatuation entsteht wird zu der nächsten Top Priority in der Open List gegangen und von dort aus weiter gesucht.



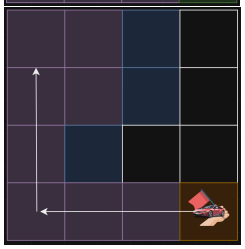
Das Feld wird wieder zur Closed List hinzugefügt. Das Einzige Mögliche Feld wird zur Open List hinzugefügt. Bei de Manhattan Heuristik hätte das Feld den Wert $H = 40$ und $G = 20$. Also insgesamt Kosten von 60.



Es kann nur diagonal gegangen werden wenn die x und die y Richtung frei sind.
Diagonal gehen kostet 14 da die Quadratwurzel aus $\sqrt{2} = 1.41...$



Wenn das Ziel in der Open List steht ist es erreicht.



Vom Ziel wird nun rückwärts der Kürzeste Pfad ausgewertet indem man den Parent der Felder nimmt die vom Start zum Ziel führen. Also man fängt man mit dem letzten Feld an und schaut wer dieses untersucht hat usw. Bis man am Start angekommen ist.

Kapitel 7

Implemntation

1. **Initialisierung:**
   - Definiere die Kartenmatrix, den Startpunkt, den Zielpunkt und den aktuellen Punkt.
   - Füge Start- und Zielpunkte zur Karte hinzu.
2. **Menüauswahl:**
   - Der Benutzer kann wählen zwischen:
     - Anzeigen der Karte
     - Einen Schritt machen
     - Den Algorithmus laufen lassen
     - Den Pfad anzeigen
     - Beenden des Programms
3. **Anzeigen der Karte:**
   - Gehe durch die Kartenmatrix und drucke jedes Feld entsprechend seines Typs aus.
4. **Einen Schritt machen:**
   - Erhalte die Nachbarn des aktuellen Punktes.
   - Berechne die Heuristik und die Gesamtkosten für jeden Nachbarn.
   - Füge gültige Nachbarn der Liste der offenen Punkte hinzu.
   - Markiere den aktuellen Punkt als geschlossen.
   - Wähle den nächsten Punkt basierend auf den niedrigsten Kosten aus.
5. **Algorithmus laufen lassen:**
   - Führe Schritte aus, bis der Zielpunkt erreicht ist.

6. **Pfad anzeigen:**
    - Verfolge den Pfad vom Zielpunkt zum Startpunkt durch die geschlossenen Punkte.
    - Gib den Pfad aus.

Methoden:

- **init():**
    - Initialisiert die Karte und fügt Start- und Zielpunkte hinzu.
- **menu():**
    - Zeigt das Benutzermenü an und verarbeitet die Benutzereingabe.
- **anzeigen():**
    - Druckt die Karte in der Konsole aus.
- **findPath():**
    - Führt den Algorithmus aus, bis der Zielpunkt erreicht ist.
- **step(int iteration):**
    - Führt einen Schritt des Algorithmus aus.
- **checkIfIn(int[][] neighbours, int[][] openValues, int j, boolean added):**
    - Überprüft, ob ein Nachbar bereits in der Liste der offenen Punkte enthalten ist.
- **getNeighbours(int[] pos):**
    - Gibt die Nachbarn eines Punktes zurück.
- **heuristic(int[] start, int[] finish):**
    - Berechnet die Heuristik zwischen zwei Punkten.
- **addToMap(int[] pos):**
    - Fügt einen Punkt der Karte hinzu.
- **retracePath():**
    - Verfolgt den Pfad vom Zielpunkt zum Startpunkt und gibt ihn zurück.