Die gierige Strategie

Die gierige Strategie (greedy strategy) haben wir im Prinzip schon in unserem einführenden Lösungsversuch kennen gelernt. Der Gedanke ist:

> Erledige immer als nächstes von den noch nicht bearbeiteten
> den fettesten Teilbrocken des Problems.
> Das bedeutet,
> nimm den größten oder kleinsten, teuersten, billigsten, …
> je nachdem, welches das Kriterium der Bewertung beim jeweiligen Problem ist.

Wie wir beim Anfangsbeispiel gesehen haben, führt dieses Vorgehen ganz schnell zu möglicherweise guten Lösungen. Allerdings sind das in vielen Fällen nicht unbedingt die besten Lösungen, da der Suchraum nicht vollständig durchsucht wird.

Einer der Hauptvorteile der gierigen Strategie ist ihre Einfachheit und Schnelligkeit. Über diesen Vorteil bei allgemeinen Problemen hinaus, mit dem man sich den Mangel erkauft, eventuell nicht die beste Lösung zu erzielen, gibt es einige Anwendungsfälle von greedy Algorithmen, bei denen man zeigen kann, das so tatsächlich auch stets die beste Lösung gefunden wird.

Das Problem: Entscheidungen sind endgültig

Erledigt man nämlich immer als nächstes den noch nicht bearbeiteten *"fettesten Teilbrocken"* des Problems, dann muss das nicht der für das Problem günstigste sein. Daher führt die gierige Strategie in vielen Fällen zu recht guten, aber nicht unbedingt optimalen Lösungen. Warum kann diese *"gierige"* Auswahl zu negativen Effekten führen?

Der Grund ist für die Frage der Einordnung in die Suchverfahren von zentraler Bedeutung: Eine einmal bei einer Verzweigung gefällte Entscheidung wird später nicht wieder rückgängig gemacht. So können Entscheidungen, die lokal günstig erschienen, die aus Sicht einer optimalen Lösung [*die man ja nicht kennt*] aber tatsächlich ungünstig waren, später nicht wieder rückgängig gemacht werden.

Es ist nicht einmal so, dass bei späteren Entscheidungen auf mögliche frühere Fehlentscheidungen hin untersucht wird! Gerade hierin, in diesem *"bedenkenlos"* Vorwärtsgehen, liegt der besondere Vorteil von greedy - Verfahren.

Untersuchen wir dafür einmal das Beispiel der Labyrinthe. Man müsste so etwas wie eine Zielheuristik haben, beispielsweise die Annahme, alle Wege mit möglichst langen Gängen seien die besten. Es zeigt sich nun sehr schnell, dass solche Heuristiken so willkürlich sind, dass ein Erfolg auf diesem Wege vernachlässigbar unwahrscheinlich ist. Die Suche des Ausgangs in einem Labyrinth ist also sicher ein Problem, dass sich für greedy nicht eignet.

Um beim Bild zu bleiben: Greedy kann ich in einem Labyrinth versuchen, bei dem es von jedem Ort zu jedem anderen viele verschiedene Wege gibt (viele Zyklen) und ich beispielsweise nicht de*n Ausgang, sondern das Ungeheuer suche und sozusagen* "immer dem Geruch nachgehe". Dann kann es mir passieren, dass ich vielleicht an der einen oder anderen Stelle eine falsche Entscheidung treffe, diese aber ohne Folgen bleibt, da ich auf einem späteren kleinen Umweg immer noch das Ziel erreichen kann: Der Weg ist dann sicher nicht optimal, das Verfahren liefert aber immer noch ein akzeptables Ergebnis bei einem vermutlich sehr viel geringeren Aufwand, als bei einer systematischen Suche.

Das eigentliche Problem aber ist: In welcher Richtung stinkts ?

minimal spanning tree

Bei der Entscheidung für eine Lösung mit greedy muss man vorher abklären, ob das Problem diesen Kriterien entspricht. Wenn ja, liegt man mit den Ergebnissen meistens recht gut.

Ein wichtiges Suchproblem läßt sich im Graphen dadurch beschreiben, dass zu einem zusammenhängenden Graphen ein immer noch zusammenhängender Teilgraph minimaler Länge gesucht wird.

Dabei wird man in der Regel einen bewerteten Graphen haben, so dass nicht allein die Zahl der auftretenden Kanten interessiert, sondern ihre Gesamtkosten. Ein typisches Beispiel ist die Aufgabe, ein ***minimales Versorgungsnetz*** (beispielsweise Wasserversorgung, für Gas oder Strom) aufzubauen. Das ist dann aber kein redundantes Netz (*eigentlich gar kein Netz!*), bei dem der Ausfall eines Netzabschnittes nicht durch die Versorgung über einen Umweg kompensiert werden kann, siehe Internet.

Kosten

Für den Aufbau eines Versorgungsnetzes fallen Kosten an, die in der Regel zumindest von der Länge der Teilverbindungen abhängig sind. Bilden wir einen Graphen zur Beschreibung dieses Netzes, dann sind den Kanten diese Kosten zuzuordnen. Allgemein wird für die Kantenbewertung in Graphen daher häufig der Begriff *Kosten* verwendet. Das Ziel in vielen Anwendungsfällen wird sein, die durch das Netz entstehenden Gesamtkosten zu minimieren.

Definition minimal spanning tree

Ein Graph heißt minimaler aufspannender Baum zu einem Graphen, wenn es keinen anderen Teilgraphen mit geringeren Kosten gibt, der auch aufspannender Baum ist.

Ein Teilbegriff in diesem Satz isti der Begriff *Baum*.

Definition Baum

Ein Graph ist ein Baum, wenn es zwischen je zwei seiner Knoten genau einen Weg gibt, in ihm also keine Zyklen auftreten.

Bei der Begriffsbildung des aufspannenden Baumes geht man zunächst von irgendeinem zusammenhängenden Graphen aus und entfernt aus ihm solche Kanten, ohne die er immer noch zusammenhängend bleibt, bis er ein Baum geworden ist. Da die Auswahl der Kanten dabei aber nicht vorgegeben ist, können aus einem vorgegebenen Graphen verschiedene Bäume entstehen.

Zwei Ansätze zur Lösung

Zu diesen vielen möglichen verschiedenen aufspannenden Bäumen werden in der Regel verschiedene Kosten gehören. Gesucht ist einer davon mit dem minimalen Wert der Kosten.

Ein Beispiel für Mehrdeutigkeit ist der Fall, bei dem von mehreren gleich langen Kanten nur eine in den Baum aufzunehmen ist. Obwohl es also zu einem vorgegebenen Graphen eventuell nicht *die eine* Lösung gibt, lassen sich bei diesem Problem einige einfache Aussagen zur Lösung angeben, auf Grund derer man relativ effektiv zu einer optimalen Lösung kommen kann.

1. Für jede gegebene Zerlegung eines Graphen in zwei Teilmengen enthält der minimal spanning tree die kürzeste der Kanten, die Knoten aus der einen Menge mit denen der anderen verbindet.
2. Man kann beim Erzeugen des minimal spanning tree mit einem beliebigen Knoten beginnen und weitere dadurch hinzufügen, dass man als nächstes immer den Knoten verwendet, der den kleinsten Kantenwert aller freien Knoten hat.

Es gibt daher nicht nur in vielen Fällen mehrere Lösungen. Es gibt sogar auch zwei leicht zu verstehende Algorithmen, mit denen das Problem des minimal spanning tree zu lösen ist. Diese Algorithmen heißen Algorithmus von Kruskal und Algorithmus von Prim.

Der Algorithmus von Kruskal

- Ordne alle Kanten nach ihren Kosten.
- Beginne mit einer Liste von Teilbäumen, die zunächst jeweils allein aus den isolierten Punkten des Graphen besteht.
- Ver binde dann jeweils immer mit der nächsten Kante mit den geringsten Kosten, die keinen Zyklus erzeugt[1], zwei Teilbäume zu einem Teilbaum.
- Brich damit ab, wenn die Liste aller Teilbäume nur noch ein Element enthält.

Man kann nun sehr leicht zeigen, dass der hier beschriebene Algorithmus sicher eine optimale Lösung findet, wenn es überhaupt Lösungen geben kann[2]. Entfernt man nämlich eine Kante, entstehen zwei Teilbäume, die mit einer *"besseren"* Kante verbunden werden müssten. Da aber jeweils kürzeste Kanten verwendet wurden, kann es bestenfalls eine gleich kurze geben, mit der die beiden Teilbäume verbunden werden.

Programm zum Algorithmus von Kruskal

Siehe dazu die Präsentation

Der Algorithmus von Prim

- Beginne mit einem Baum, der allein aus einem Knoten besteht.
- Ordne alle Kanten, die von diesem Knoten ausgehen, nach ihren Kosten. (→ Prioritätswarteschlange nach den Bewertungen)
- Füge nun jeweils immer aus dieser Prioritätswarteschlange jeweils die nächste Kante mit den geringsten Kosten ein, die keinen Zyklus erzeugt. Dann füge alle von derm freien Knoten ausgehenden zulässigen Kanten in die Prioritätswarteschlange ein.
- Brich damit ab, wenn alle Knoten des Graphen zum Baum gehören.

Auch hier kann man sehr leicht zeigen, dass der hier beschriebene Algorithmus eine Lösung findet, wenn es überhaupt Lösungen geben kann. Wir sehen uns hier ebenso einmal den Aufbau des Baumes für ein Beispiel an. Beim Algorithmus von Prim sind die beim Aufbau des Baumes entstehenden Teilgraphen stets zusammenhängend. Wird eine Kante hinzugefügt, dann hat sie stets ein Knoten mit dem derzeitigen Teilgraphen gemeinsam, setzt also einen Ast fort oder bildet einen neuen.

Bemerkenswert für unsere Betrachtung verschiedener Suchverfahren ist die Feststellung:

1 Das ist dann der Fall, wenn beide Knoten der Kante zum selben Teilgraphen gehören.
2 Dazu ist Voraussetzung, dass der ursprüngliche Graph zusammenhängend ist.

Die Datenstruktur ändert sich!

Mit dem völlig neuen Ansatz zur Lösung des Problems benötigen wir auch eine völlig neue Datenstruktur: Es wird eine Datenstruktur aufgebaut, in der die Elemente nicht wie bei stack (→ Tiefensuche) und queue (Warteschlange ; → Breitensuche) nach dem Zeitpunkt ihres Einfügens in die Datenstruktur geordnet werden, sondern sie werden nach dem ihnen zugehörigen Wert geordnet eingefügt[1].

> Die angemessene Datenstruktur zur Lösung des Problems ist die Prioritäts - Warteschlange.

Das Programm ist wegen der Behandlung der Prioritätswarteschlange etwas umfangreicher als das zum Algorithmus von Kruskal. Allerdings setzt man die Funktionen zur Verwaltung der Prioritätswarteschlange auch bei Kruskal zum anfänglichen Sortieren der Kanten ein. Es wird dann zwar nicht erneut in die Prioritätswarteschlange eingefügt – aber wozu eine unabhängige Lösung erstellen.

Programm zum Algorithmus von Prim

Siehe dazu die Präsentation

1 Alternativ kann man eine Zugriffsfunktion entwickeln, durch die der Zugriff so funktioniert, als wäre das so.