Schwierige Probleme in der Informatik

Informationen für die Lehrperson

Thema, Adressaten,...

Das Thema dieses Moduls sind NP-vollständige Probleme, also schwierige Probleme in der Informatik. GraphBench wird verwendet, um das abstrakte Thema Komplexitätstheorie mit einem praktischen Ansatz zu vermitteln. Wir konzentrieren uns auf Entscheidungsprobleme, die Fragen nach der Existenz einer Lösung beantworten. Die folgenden Probleme können mit GraphBench betrachtet werden:

- Colorability (Graphenfärbbarkeit)
- Vertex Cover (Knotenabdeckung)
- Clique (Komplette Subgraphen)
- Travelling Salesman (Problem des Handelsreisenden)
- Hamilton Circuit (Hamiltonkreis)
- Independent Set (Unabhängige Menge)
- Satisfiability (Erfüllbarkeit logischer Formeln)
- 3CNF-SAT (Satisfiability mit logischen Formeln in spezieller Form)

Entdeckendes Lernen

Das Unterrichtskonzept, das verwendet wird, ist *Entdeckendes Lernen*. Hier erhalten die Schüler Material und eine Lenkung, mit der sie selbstständig auf Entdeckungsreise gehen können. Was dabei herauskommt ist ungewiss, wird aber durch die Lenkung eingegrenzt.

Unterrichtsstufe

Diese Unterrichtseinheit eignet sich für Gymnasiasten mit 2 Wochenstunden Informatik oder für Informatikstudenten als Einführung in die theoretische Informatik.

Erforderliches Vorwissen

- Kenntnis des Begriffs Algorithmus
- Kenntnis des Konzeptes logischer Formeln
- Kenntnis der konjunktiven Normalform (CNF) logischer Formeln
- Englischgrundkenntnisse zur Bedienung von GraphBench

Ablauf

10 Min.	Einführung Schüler lesen Skript (ohne Erklärung seitens Lehrperson)
10 Min.	GraphBench Tutorial
60 Min.	Gelenktes Entdeckendes Lernen (mindestens 30 Minuten) Schüler bearbeiten ein NP-vollständiges Problem mit GraphBench und analysieren Algorithmen, Lösungswege, Laufzeiten, usw..
10 Min.	Besprechung in 2-er Gruppen
30 Min.	Überarbeitung Erkenntnisse Vorbereitung Kurzvortrag
45 Min.	Präsentation Kurzvorträge Einige Schüler tragen ihre Präsentation vor.

Evtl. Bewertung der Erkenntnisse und Vorträge.

Simulation möglicher Ergebnisse

Die Schüler können in mehrere Richtungen entdecken. Nachdem sie eruiert haben, um was es im Problem geht, forschen sie je nach Interesse in verschiedenen Bereichen. Die einen finden heraus, wie der Algorithmus genau abläuft. Andere interessiert, wie lange ein Algorithmus braucht um das Problem zu lösen. Es kann auch interessant sein, Fälle zu suchen, in denen Heuristiken eine falsche Antwort geben. Praktiker suchen Anwendungen in der Realität.

Mögliche Ergebnisse sind in die Kategorien Algorithmus, Laufzeit, Korrektheit und Anwendung aufgeteilt. Dies ist eine Auswahl möglicher Entdeckungen.

1) Graphfärbungsproblem

Algorithmus

- Der *Backtracking* Algorithmus testet alle Fälle, bis er eine Lösung findet oder alle Fälle durchprobiert hat.
- Der Algorithmus *Edge Optimization* loopt, falls er keine Lösung findet.
- Der Algorithmus *Edge Optimization* startet mit einer zufälligen Färbung. Er markiert alle Kanten, die zwei Knoten gleicher Farbe verbinden. Dann wählt er in jedem Schritt eine dieser Kanten und färbt einen Knoten neu, um die Kante konfliktfrei zu machen. Er fährt weiter, bis er eine Lösung findet oder gerät in einen Loopzustand und endet nie.

Laufzeit

- Es ist möglich, dass *Edge Optimization* das Problem in einem Schritt löst: Wenn die zufällige Färbung eine Lösung ist.
- *Edge Optimization* ist meistens schneller als *Backtracking*, benötigt also weniger Schritte.
- In einem Graphen mit 2 benachbarten Knoten braucht *Backtracking* 7 Schritte, um eine Antwort auf die Frage "existiert eine Färbung mit 2 Farben" zu finden. Die Heuristik benötigt auch 7 Schritte. *Edge Optimization* braucht zwischen 1 und 5 Schritten.

Korrektheit

- Heuristiken liefern nicht in jedem Fall die korrekte Lösung. Falls *Edge Optimization* im ersten Schritt eine ungünstige Färbung wählt und dann keine gleichfarbigen benachbarten Knoten umfärben kann, gerät der Algorithmus in einen Loop Zustand. Dieser Fall kann auch auftreten, wenn eine korrekte Färbung existiert.

Anwendungen

- Ob man Stände vier verschiedener Arten auf Kreuzungen so aufstellen kann, dass nicht zwei gleicher Art an benachbarten Kreuzungen stehen,

kann man herausfinden, indem man einen Algorithmus für das Graph Coloring Problem anwendet.

- In der Berner Altstadt kann man die gewünschten Bedingungen mit 4 verschiedenen Ständen erfüllen.
- Auch das Stundenplan Problem lässt sich auf das Graphfärbungsproblem abbilden. Die verschiedenen Stunden sind die Knoten. Zwei Stunden, die nicht gleichzeitig stattfinden dürfen (z.B. gleiche Lehrperson, gleiche Klasse oder dasselbe Zimmer), werden mit einer Kante verbunden.
- Das Einfärben einer Landkarte mit verschiedenen Farben. Die Länder sind die Knoten. Zwei benachbarte Länder werden mit einer Kante verbunden.

2) Vertex Cover

Algorithmus

- In einem vollständigen Graphen mit *n* Knoten existiert keine Lösung mit weniger als *n-1* Knoten.
- *Backtracking* prüft für ein Vertex Cover mit *n* Knoten alle Möglichkeiten von Knotenmengen der Grösse *n*. Der Algorithmus läuft solange, bis er eine Lösung gefunden oder alle Möglichkeiten getestet hat.
- Ansatz für einen Algorithmus: man wählt zuerst alle Knoten aus. Dann entfernt man Knoten solange bis nicht mehr alle Kanten von einem Knoten abgedeckt sind. Hat man eine Lösung gefunden ist man fertig. Ansonsten muss man durch *Backtracking* eine andere Möglichkeit durchprobieren. Je nachdem kann dies schneller sein als das normale *Backtracking*.

Laufzeit

- Falls keine Lösung existiert, probiert *Backtracking* alle Möglichkeiten aus. Das kann je nach Grösse des Graphes und der Clique sehr lange dauern.

Korrektheit

- Backtracking liefert stets das richtige Resultat. Daher werden auch Extremfälle korrekt behandelt.

Anwendung

- Die Polizei kann die günstigsten Positionen ihrer Leute ermitteln, in dem sie das Vertex Cover Problem auf dem Strassennetz löst. Die Kreuzungen entsprechen den Knoten, die Strassenabschnitte den Kanten.

3) Clique

Algorithmus

- *Backtracking* prüft im *n*-Cliquen Problem alle Untermengen der Grösse *n*.

Findet der Algorithmus eine Clique der Grösse *n*, terminiert er und liefert die Clique. Ansonsten terminiert er, wenn alle Möglichkeiten durchsucht wurden.

- Knoten, die weniger als *n*-1 Nachbarn haben, müssen bei der Suche nach einer *n*-Clique nicht beachtet werden. Die Kanten zu diesen Knoten können ebenfalls ignoriert werden.

Laufzeit

- In einem grossen Graphen mit 15 Knoten und 37 Kanten hat Backtracking eine Clique der Grösse 5 erst nach ungefähr 5 Minuten gefunden.

Korrektheit

- *Backtracking* liefert immer die richtige Lösung.

Anwendung

- Das Cliquenproblem entspricht dem Problem der Positionierung der Kameras des Regionalfernsehens.
- Die grösste Clique in der Berner Altstadt hat die Grösse 3. Das Regionalfernsehen kann die Kameras daher nicht wie gewünscht aufstellen.

4) Travelling Salesman

Algorithmus

- Die *Convex Hull* Heuristik erzeugt zuerst die konvexe Hülle der Knoten. Dann wird der Knoten ausgewählt, der der Tour am nächsten liegt und hinzugefügt. Der Algorithmus terminiert, wenn alle Knoten Teil der Tour sind.
- Die *Nearest Neighbour* Heuristik startet mit dem ersten Knoten und besucht immer den nächstliegendsten Knoten, bis alle Knoten besucht sind.
- *Two Optimization* startet mit einer vordefinierten Tour und versucht dann, diese zu optimieren. Die Heuristik versucht zwei Kanten zu finden, so dass die Tour kleiner wird, falls man deren Endknoten vertauscht. Dies macht er, bis er keine zwei solchen Kanten mehr findet.
- Travelling Salesman sucht den minimalen Hamiltonkreis in einem Graphen.

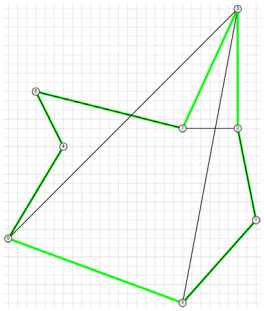
Laufzeit

- *Nearest Neighbour* berechnet *n**(*n*-1)/2 mal die Entfernung zwischen zwei Knoten. Er braucht also nicht mehr als *n*^2 Schritte.
- Die Heuristiken liefern nicht immer das korrekte Resultat, jedoch sind sie sehr schnell bzw. brauchen wenige Schritte.
- Bei meinen Versuchen hat *Convex Hull* durchschnittlich am wenigsten

Schritte benötigt.

Korrektheit

- *Nearest Neighbour* findet nicht immer die richtige Lösung. Auch für *Convex Hull* konnte ich einen Fall finden (siehe Bild), so dass diese Heuristik nicht die richtige Lösung findet. Anscheinend finden Heuristiken nicht immer die richtige Lösung. Für *Two Optimization* konnte ich keinen Fall finden, bei dem der Algorithmus eine falsche Lösung geliefert hätte.



Anwendung

- Der Zeppelin fliegt die kürzeste Route zwischen den wichtigen Plätzen: Diese Aufgabe kann mit dem Travelling Salesman Problemansatz gelöst werden.
- Ein Kurier, der verschiedene Pakete an verschiedene Orte abliefern muss, sucht den kürzesten Hamiltonkreis. Dies entspricht auch dem Travelling Salesman Problem.

5) Hamiltonkreis

Algorithmus

- *Backtracking* arbeitet mit Pfaden. Er beginnt mit einem Knoten und speichert alle Pfade, die von diesem Knoten aus erreicht werden können. Dann nimmt er stets den letzten Pfad und kreiert neue Pfade, indem er die Kante hinzufügt, die den letzten Knoten mit einem unbesuchten Knoten verbindet. Er stoppt, wenn alle Pfade durchsucht sind oder eine Lösung gefunden wird.
- Dadurch, dass *Backtracking* die unbesuchten Pfade irgendwie speichern muss, braucht er viel Speicherplatz.

Laufzeit

- Die Laufzeit ist nicht sehr schnell, da der Algorithmus viele Speicherzugriffe macht und eventuell alle Pfade durchsuchen muss.
- Je mehr Knoten und Kanten ein Graph hat, desto länger braucht der Algorithmus.
- Wenn ein Knoten und seine Kanten bei einem grossen Graphen hinzugefügt werden, wächst die Anzahl Schritte um mehr als bei einem

kleineren Graphen.

Korrektheit

- *Backtracking* liefert immer das richtige Ergebnis.

Anwendung

- Das Hamilton Circuit Problem entspricht dem Problem des Organisieren des Kinderumzuges.

6) Independent Set

Algorithmus

- *Backtracking* prüft alle möglichen Fälle.
- Knoten, die zu bereits ausgewählten Knoten benachbart sind, müssen nicht betrachtet werden.
- Wenn man den Graph invertiert, d.h. die Kanten löscht und dort Kanten macht, wo vorher keine waren, kann man das Independent Set auf das Cliquenproblem abbilden.

Laufzeit

- *Backtracking* ist sehr langsam.
- Bei kleinen Graphen könnte man das Independent Set raten und wäre damit vielleicht schneller. Im schlechtesten Fall geht die Laufzeit aber ins Unendliche.
- Sehr schwierig und zeitaufwändig ist es, das maximale Independent Set zu finden (grösste Anzahl Knoten im Set).

Korrektheit

- *Backtracking* liefert in jedem Fall das richtige Ergebnis.

Anwendung

- Independent Set entspricht dem Problem des Platzieren der Bühnen.
- Einige meiner Kollegen (Knoten) haben mit gewissen anderen Kollegen Streit (Kanten). Wieviele maximal und welche Kollegen kann ich zu meiner Geburtstagsparty einladen, ohne dass es zu einem Streit kommt?

7) Satisfiability

Algorithmus

- *Backtracking* setzt zuerst alle Variablen auf falsch. Dann sucht es Schritt für Schritt alle Möglichkeiten durch, bis eine Lösung gefunden wird oder alle Möglichkeiten erschöpft sind.

- Das physikalische Modell löst das Problem, indem es physikalische Kräfte benutzt. Jede Klausel zieht mit einer gewissen Kraft an den Variablen, die darin vorkommen.

Laufzeit

- Das physikalische Modell braucht weniger Schritte als *Backtracking*.

Korrektheit

- *Backtracking* liefert immer das korrekte Resultat.

Benotung

Möglich ist eine Bewertung der Erkenntnisse.

Für nicht triviale Erkenntnisse gibt es einen Notenpunkt. Ungültig ist z.B. die Erkenntnis "ein Graph besteht aus Knoten und Kanten".

Falls die Kurzvorträge von allen Schülern gehalten werden, können auch diese bewertet werden.

Bezugsquelle GraphBench-Software

Die Schüler benötigen in der Entdeckungsphase die Software GraphBench. Diese ist unter http://www.swisseduc.ch/informatik/graphbench/ kostenlos erhältlich. Zum Ausführen des Programms wird Java benötigt. Weitere Informationen finden Sie auf der angegebenen Homepage.