TU Wien
Institut für Logic and Computation
Forschungsbereich Algorithms and Complexity

186.866 Algorithmen und Datenstrukturen VU

Programmieraufgabe P4

PDF erstellt am: 1. Mai 2024

1 Vorbereitung

Um diese Programmieraufgabe erfolgreich durchführen zu können, müssen folgende Schritte umgesetzt werden:

1. Laden Sie das Framework `P4.zip` aus TUWEL herunter.
2. Entpacken Sie `P4.zip` und öffnen Sie das entstehende Verzeichnis als Projekt in IntelliJ (nicht importieren, sondern öffnen).
3. Öffnen Sie die nachfolgend angeführte Datei im Projekt in IntelliJ. In dieser Datei sind sämtliche Programmiertätigkeiten durchzuführen. Ändern Sie keine anderen Dateien im Framework und fügen Sie auch keine neuen hinzu.
   `src/main/java/exercise/StudentSolutionImplementation.java`
4. Füllen Sie Vorname, Nachname und Matrikelnummer in der Methode `StudentInformation provideStudentInformation()` aus.

2 Hinweise

Einige Hinweise, die Sie während der Umsetzung dieser Aufgabe beachten müssen:

- Lösen Sie die Aufgaben selbst und nutzen Sie keine Bibliotheken, die diese Aufgaben abnehmen.
- Sie dürfen beliebig viele Hilfsmethoden schreiben und benutzen. Beachten Sie aber, dass Sie nur die oben geöffnete Datei abgeben und diese Datei mit dem zur Verfügung gestellten Framework lauffähig sein muss.

3 Übersicht

In dieser Programmieraufgabe werden Sie mit einem praktischen Anwendungsfall einer Reduktion konfrontiert. Die Problemstellung ist, in einem gegebenen sozialen Netzwerk eine größte Clique zu finden. Sie werden allerdings keinen eigenen Lösungsalgorithmus entwerfen, sondern das Problem auf SAT reduzieren und mithilfe eines bereits existierenden SAT-Solvers lösen.

4 Theorie

Die Theorie für diese Programmieraufgabe befindet sich in den Vorlesungsfolien „Polynomialzeitreduktion".

5 Implementierung

Gegeben sei ein Netzwerk, in dem Personen paarweise eine symmetrische Eigenschaft teilen können, beispielsweise Freundschaft (repräsentiert durch Kanten). Wir wollen nun eine größtmögliche Menge von Personen finden, in der paarweise alle diese Eigenschaft teilen. Wir modellieren dies als ungerichteten, schlichten Graphen $G = (V, E)$ mit $|V|$ Knoten (die Personen) und $|E|$ Kanten (die symmetrischen Eigenschaften). Sei nun $C \subset V$ genau dann eine *Clique* der Größe $|C| = k$, wenn für alle $v, w \in C, v \neq w$ gilt, dass $\{v, w\} \in E$. Dies entspricht dann beispielsweise einem Freundkreis bestehend aus k Personen. Betrachten Sie ein Beispielnetzwerk in Abbildung 1, Zachary's berühmt-berüchtigter Karate Club [2].

Vergleichen Sie die Problemstellung mit der Definition für das Entscheidungsproblem CLIQUE. Dieses ist für ein bestimmtes k definiert. Es gilt die Frage zu beantworten, ob für einen bestimmten ungerichteten Eingabe-Graphen eine Clique mit mindestens der Größe k existiert. Führen Sie eine Polynomialzeitreduktion dieses Problems auf SAT durch, indem Sie die Informationen aus Knoten, Kanten und dem Wert k zusammenführen, um eine aussagenlogische Formel in konjunktiver Normalform zu erstellen. Bilden Sie diese Formel im DIMACS-Format als `String` ab.

Hinweis: Verwenden Sie als Entscheidungsvariablen x_{iv}, ob ein Knoten das i-te Element einer k-Clique ist, Sie können sich an gegebenen Internetquellen

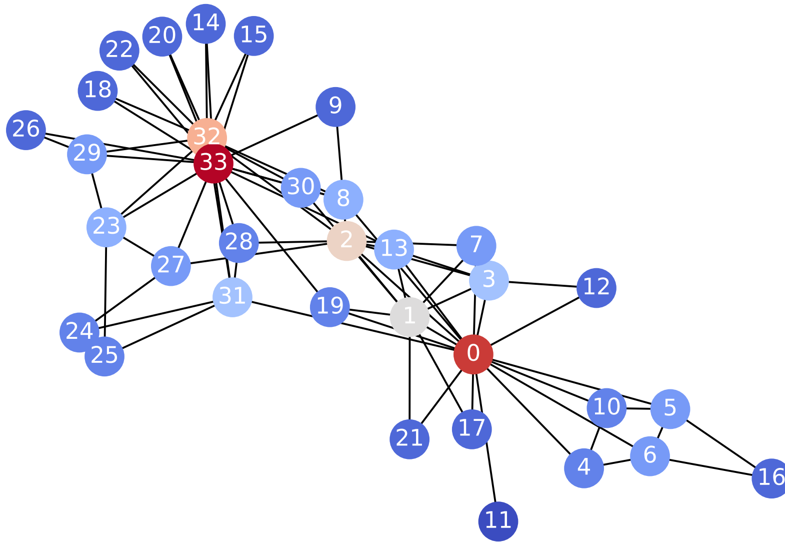

Abbildung 1: Zachary's Karate Club [2], Knoten mit hohem Grad in rot, mit geringem in blau. Ein klassisches Beispiel für Community-Strukturen in einem Netzwerk (der Club zerfällt in zwei Unterclubs), wir verwenden ihn in unserem Kontext als Testinstanz zur Cliquen-Findung. Seine Cliquenzahl (die Größe einer größten Clique) ist 5.

orientieren[1] [2]. *Wandeln Sie diese Adressierung über zwei Indizes in eine über einen Index* $l = 1, \ldots, k \cdot |V|$ *für die aussagenlogische Formel um, dies gelingt beispielsweise über die Transformationen* $l = |V| \cdot (i-1) + v$ *bzw.* $v = 1 + ((l-1) \bmod |V|)$.

Die erste Zeile besteht aus `"p cnf n_var n_claus"`, wobei `n_var` durch die Anzahl der Variablen und `n_claus` durch die Anzahl der Klauseln von Ihnen ersetzt werden muss. Vergessen Sie nicht, nach jeder Zeile einen Zeilenumbruch `"\n"` anzuhängen. Jede weitere Zeile entspricht nun einer Klausel. Innerhalb einer Zeile halten Sie mit Abständen getrennt die in der Klausel vorkommenden Literale fest. Literale sind negierte oder nicht-negierte Variablen. Eine Variable können Sie durch eine beliebige natürliche Zahl, die größer als 0 ist, repräsentieren. Entspricht ein Literal einer negierten Variable, können Sie dies durch ein vorangestelltes `"-"` ausdrücken. Jede Zeile einer Klausel muss mit `" 0"` beendet werden.

Die aussagenlogische Formel $(x_1 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_4)$ mit $n_{\text{var}} = 4$ Variablen

[1] `https://cs.stackexchange.com/questions/70531/reduction-3sat-and-clique`

[2] `https://blog.computationalcomplexity.org/2006/12/reductions-to-sat.html`

und $n_{\text{claus}} = 2$ Klauseln wird im DIMACS-Format wie folgt dargestellt:

```
p cnf 4 2
1 -3 0
-1 2 4 0
```

Abbildung 2: Aussagenlogische Formel im DIMACS-Format

Implementieren Sie die Methode `int findMaxClique(Graph g, TimedSolver solver, boolean[] chosenVertices)`. Der erste Parameter `Graph g` entspricht dem soeben diskutierten Eingabe-Graphen G. Dieser bietet folgende Methoden an:

- `int numberOfVertices()`: gibt die Anzahl der Knoten in G zurück, also $|V|$. Die Knoten werden durch ihre ID angesprochen, welche von 1 bis $|V|$ läuft.
- `boolean containsEdge(int v, int w)`: gibt zurück, ob die ungerichtete Kante $\{v, w\}$ im Graphen G existiert.

Der zweite Parameter `TimedSolver solver` beinhaltet den SAT-Solver. Dieser bietet eine Methode an:

- `String solve(String dimacs)`: Wenn Sie an diese Methode Ihre DIMACS encodierte aussagenlogische Formel übergeben, versucht der SAT-Solver, eine erfüllende Variablenbelegung zu finden.

  Wurde eine gefunden, dann erhalten Sie einen `String` mit durch Leerzeichen getrennten Literalen. Negierte Variablen bedeuten, dass in der gefundenen Variablenbelegung diese Variablen *falsch* sind. Nicht-negierte Variablen implizieren *wahr*. Wie auch zuvor bei den Klauseln, wird auch dieser `String` durch `" 0"` beendet. Ein Resultat für die Formel in Abbildung 2 könnte z.B. `"-1 -2 -3 -4 0"` sein.

  Wurde keine Variablenbelegung gefunden, dann erhalten Sie einen `String` der Länge 0 als Rückgabe. Sollte es Probleme mit der DIMACS-Encodierung geben, dann erhalten Sie eine Fehlermeldung, die mit `"Parsing error in solver: "` beginnt.

Achten Sie nun darauf, dass Sie eine größte Clique finden möchten. Führen Sie deshalb die Reduktion und das Lösen mit dem SAT-Solver im Rahmen

einer **binären Suche** über k durch, die das größtmögliche k finden soll, d.h. man fängt mit k in der Mitte von $\{1, \ldots, |V|\}$ an und setzt dann entweder links/kleiner (wenn die Formel unerfüllbar ist) oder rechts/größer (wenn die Formel erfüllbar ist) im selben Sinne fort. **Wichtig:** Verwenden Sie entweder die Java-Klassen `StringBuilder`[3] oder `StringBuffer`[4] zum stückweisen Aufbau des DIMACS-Strings, und wandeln erst **am Ende** in einen `String` um, da sonst die Reduktion viel zu langsam wird.

Haben Sie eine Lösung gefunden, die die größtmögliche Anzahl an Knoten darstellt, die alle paarweise durch eine Kante verbunden sind, so speichern Sie diese im Parameter `boolean[] chosenVertices` ab. Rekonstruieren Sie diese aus dem Ergebnis des SAT-Solvers. Der Parameter `boolean[] chosenVertices` ist ein Array, dessen Länge der Anzahl der Knoten im Graphen $|V|$ entspricht. Sollten Sie den Knoten v auswählen, dann setzen Sie auch den zugehörigen Wert `chosenVertices[v-1]` (der erste Knoten ist bei Index 0) auf `true`. Sollte es mehrere Lösungen mit der größtmöglichen Anzahl an Knoten geben, wählen Sie eine beliebige aus.

Als Rückgabewert wird die Cliquenzahl, also die Größe einer größten gefundenen Clique, erwartet.

6 Testen

Führen Sie zunächst die `main`-Methode in der Datei `src/main/java/framework/Exercise.java` aus.

Anschließend wird Ihnen in der Konsole eine Auswahl an Testinstanzen angeboten, darunter befindet sich zumindest `abgabe.csv`:

```
Select an instance set or exit:
[1] abgabe.csv
[0] Exit
```

Durch die Eingabe der entsprechenden Ziffer kann entweder eine Testinstanz ausgewählt werden oder das Programm (mittels der Eingabe von `0`) verlassen werden. Wird eine Testinstanz gewählt, dann wird der von Ihnen implementierte Programmcode ausgeführt. Kommt es dabei zu einem Fehler, wird ein Hinweis in der Konsole ausgegeben.

[3] `https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/StringBuilder.html`

[4] `https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/StringBuffer.html`

Relevant für die Abgabe ist das Ausführen der Testinstanz `abgabe.csv`.

Die weitere Testinstanz `small-clique.csv` beinhaltet eine kleine Problemstellung und Zachary's Karate Club zum anfänglichen Testen.

Die Abgabeinstanzen sind Zufallsgraphen nach dem Barabási-Albert-Modell [1] (BA-Modell). Bei diesem lässt man einen Graphen knotenweise wachsen, und verknüpft jeden hinzugefügten Knoten zufällig mit m anderen Knoten. Dabei werden solchen andere Knoten bevorzugt, die selbst schon einen höheren Knotengrad haben (*preferential attachment*). Dadurch entsteht asymptotisch ein *skalenfreies* Netzwerk, eine Eigenschaft die in manchen realen Netzwerken zu finden ist (z.B. Zitationsnetzwerke). Grob bedeuetet es, dass die Knotengrad-Verteilung nicht exponentiell abfällt und Knoten mit hohem Grad (sehr zentrale Knoten) wesentlich wahrscheinlicher sind.

Die Testinstanz `abgabe.csv` sollte in höchstens ein paar Minuten durchlaufen, andernfalls ist etwas ineffizient implementiert, mögliche Fehlerquellen dafür sind:

- Reduktion per langsamen Aufbau über `String`-Verknüpfungen anstatt `StringBuffer`
- Reduktion exponentiell anstatt polynomiell
- Lineare Suche anstatt binärer Suche

In diesen Testfällen ist jeweils ein Graph mit $|V| \in \{25, 30, \ldots, 50\}$, $m \in \{2, 4, 8\}$ zu lösen. m ist der Parameter aus dem BA-Modell, je größer desto dichter werden die Graphen.

Freiwillige Challenge: Als weitere Instanz steht Ihnen `abgabe-challenge.csv` zur Verfügung mit $|V| \in \{25, 30, \ldots, 80\}$, $m \in \{2, 4, 8, 16\}$. Für diese müssen Sie dem SAT-Solver voraussichtlich helfen, ein Hinweis dazu findet sich in der letzten Frage.

7 Evaluierung

Wenn der von Ihnen implementierte Programmcode mit der Testinstanz `abgabe.csv` ohne Fehler ausgeführt werden kann, dann wird nach dem Beenden des Programms im Ordner `results` eine Ergebnis-Datei mit dem Namen `solution-abgabe.csv` erzeugt.

Die Datei solution-abgabe.csv beinhaltet Zeitmessungen der Ausführung der Testinstanz abgabe.csv, welche in einem Web-Browser visualisiert werden können. (Auch Ergebnis-Dateien anderer Testinstanzen können zu Testzwecken visualisiert werden.) Öffnen Sie dazu die Datei visualization.html in Ihrem Web-Browser und klicken Sie rechts oben auf den Knopf *Ergebnis-Datei auswählen*, um solution-abgabe.csv auszuwählen.

Beantworten Sie basierend auf der Visualisierung die Fragestellungen aus dem folgenden Abschnitt.

8 Fragestellungen

Öffnen Sie solution-abgabe.csv und bearbeiten Sie folgende Aufgaben- und Fragestellungen:

1. Beschreiben Sie die von Ihnen implementierte Reduktion auf SAT und argumentieren Sie informell ihre Korrektheit.

2. Wie viele Klauseln und wie viele Variablen entstehen in Ihrer aussagenlogischen Formel in Abhängigkeit von $|V|$, $|E|$ und k? Sind diese Abhängigkeiten wie gewünscht polynomiell?

3. Betrachten Sie die Plots sämtlicher Gruppen, wobei jeweils auch getrennt die reinen Reduktionszeiten ohne die des SAT-Solvers visualisiert werden. Von welchen Faktoren scheint die Laufzeit abhängig zu sein? Diskutieren Sie auch mögliche Gründe. Erstellen Sie im Anschluss Screenshots der Plots.

4. Um welchen Faktor haben wir die Anzahl der gültigen Lösungen für den SAT-Solver, und somit den Suchraum, durch unsere polynomielle Reduktion künstlich aufgeblasen?

5. *Freiwillige Challenge:* Schaffen Sie es auch, die abgabe-challenge.csv Instanzen in vernünftiger Zeit zu lösen? Wenn ja, präsentieren Sie den entsprechenden Plot und diskutieren was nötig war, um dies zu erreichen. *Hinweis: reduzieren Sie den Suchraum für den Solver durch zusätzliche sogenannte Symmetry-Breaking-Constraints. Wenn Sie eine aufsteigende Reihenfolge der Knoten erzwingen, dann ist beispielsweise nur noch die Lösung* $(1, 5, 7)$ *für* $k = 3$ *(*x_{11}, x_{25} *und* x_{37} *sind auf „wahr“*

gesetzt) gültig und nicht mehr die äquivalenten Lösungen $(1, 7, 5), (7, 5, 1), (7, 1, 5), (5, 7, 1), (5, 1, 7)$.

Um Plots als Bild zu speichern, drücken Sie in der Menüleiste rechts über dem Plot auf den Fotoapparat. Falls sich im Zuge der Evaluierung die Darstellung der Plots auf ungewünschte Weise verändert (z.B. durch die Auswahl eines zu kleinen Ausschnitts), können Sie mittels Doppelklick auf den Plot oder Klick auf das Haus in der Menüleiste die Darstellung zurücksetzen.

Fügen Sie Ihre Antworten in einem Bericht gemeinsam mit dem erstellten Bildern der Visualisierung der Laufzeiten der Testinstanz `abgabe.csv` (bzw. auch `abgabe-challenge.csv`) zusammen.

9 Abgabe

Laden Sie die Datei `src/main/java/exercise/StudentSolutionImplementation.java` in der TUWEL-Aktivität *Hochladen Source-Code P4* hoch. Fassen Sie diesen Bericht mit den anderen für das zugehörige Abgabegespräch relevanten Berichten in einem PDF zusammen und geben Sie dieses in der TUWEL-Aktivität *Hochladen Bericht Abgabegespräch 2* ab.

Literatur

[1] Réka Albert and Albert-László Barabási. Statistical mechanics of complex networks. *Reviews of modern physics*, 74(1):47, 2002.

[2] Wayne W Zachary. An information flow model for conflict and fission in small groups. *Journal of anthropological research*, 33(4):452–473, 1977.

10 Nachwort

Polynomialzeitreduktionen sind wichtige Fundamente sowohl der theoretischen als auch der praktischen Informatik. Wenn wir ein Problem A auf ein Problem B in Polynomialzeit reduzieren können, wissen wir, dass „A nicht schwerer als B" sein kann. Somit übertragen sich komplexitätstheoretische untere Schranken (wie zB. NP-Schwere) von A

nach B. Umgekehrt, können wir effiziente Algorithmen für das Problem B auf A übertragen. Zum Beispiel, falls B das Erfüllbarkeitsproblem ist, und wir A mittels eines SAT-Solvers lösen wollen.

Es gibt mehrere Varianten von Reduktionen. Die Reduktionen, die in der Vorlesung besprochen wurden, wo genau einmal eine Instanz von A in eine Instanz von B übersetzt wird, werden *Karp-Reduktionen* genannt (weil sie von Richard Karp in seinem berühmten Artikel von 1972 verwendet wurden, siehe auch Vorlesungsfolien). Eine andere wichtige Variante sind die *Turing-Reduktionen*, auch *Cook-Reduktionen* genannt (nach Alan Turing und Stephen Cook). Hier verwendet ein Algorithmus, der das Problem A löst, immer wieder Aufrufe zum Problem B; d.h. der Algorithmus für Problem A generiert mehrmals Instanzen des Problems B und löst sie mittels eines „black box solvers“ (auch *Orakel* genannt). Diese Reduktionsvariante wird ebenfalls sowohl theoretisch für untere Schranken als auch praktisch zur Problemlösung verwendet. Falls sich bei Turing-Reduktionen die verschiedenen Instanzen des Problems B nur geringfügig unterscheiden, kann die Effizienz gesteigert werden, indem der Solver für B *inkrementell* arbeitet, d.h., nicht immer von vorne beginnt, sondern gewisse Teillösungen wieder verwendet.

Das Thema Reduktionen wird in zahlreichen (Master-)Lehrveranstaltungen näher behandelt. Eine unvollständige Liste ist: , 181.142 VU Complexity Theory, 184.090 VU SAT Solving and Extensions, 192.122 VU Algorithmic Meta-Theorems, 186.861 VU Modeling and Solving Constrained Optimization Problems, 184.215 VU Complexity Analysis.