TU Wien
Institut für Logic and Computation
Forschungsbereich Algorithms and Complexity

186.866 Algorithmen und Datenstrukturen VU

Programmieraufgabe P5

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 `P5.zip` aus TUWEL herunter.
2. Entpacken Sie `P5.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 wird ein Branch-and-Bound Solver für ein exemplarisches kombinatorisches Optimierungsproblem, das Maximum Independent Set Problem (MISP), implementiert. Insbesondere werden die Auswirkungen auf die praktische Laufzeit des Solvers abhängig von folgenden Aspekten betrachtet:

- Vergleich unterschiedlicher Auswahlstrategien für das nächste Teilproblem.
- Vergleich unterschiedlicher Branching-Strategien.
- Vergleich unterschiedlich starker Dualheuristiken.
- Variation eines Strukturparameters der Instanzen mit zugehörigen Phasenübergängen.

Vieles davon wird bereits vorgeben sein, d.h. lassen Sie sich nicht abschrecken!

4 Theorie

Die notwendige Theorie kann in den Vorlesungsfolien „Optimierung - Branch-and-Bound“ gefunden werden.

5 Implementierung

Wir betrachten nun im Detail das gegebene Optimierungsproblem und dessen Motivation, die für die Aufgabe betrachteten Instanzen, die konkret zu implementierende Branch-and-Bound Ausprägung und die bereits zur Verfügung stehenden Hilfsmethoden.

5.1 Maximum Independent Set Problem

Gegeben sei ein ungerichteter, schlichter Graph $G = (V, E)$ mit $|V| = n$ Knoten und $|E| = m$ Kanten. Ein *Independent Set* I ist eine Untermenge

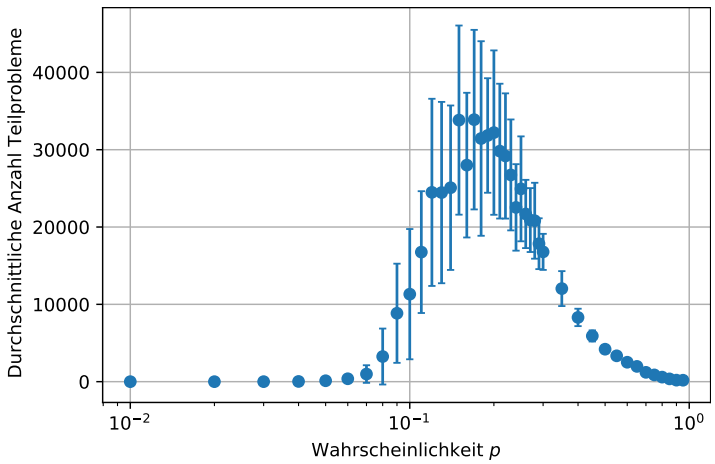

Abbildung 1: Durchschnittliche Anzahl mit $\pm 1\sigma$ von expandierten Teilproblemen in State-of-the-Art Branch-and-Bound Solver angewandt auf das MISP variiert über den Strukturparameter p, die Wahrscheinlichkeit für eine Kante im Zufallsgraphen mit jeweils 20 Zufallsgraphen mit $n = 100$.

von Knoten $I \subset V$, welche paarweise nicht benachbart sind, d.h. es muss gelten $\forall v_1, v_2 \in I \colon \{v_1, v_2\} \notin E$. Sei nun $\mathcal{I}_G$ die Menge aller Independent Sets zum Graphen G. Das Maximum Independent Set Problem (MISP) fragt nach einem Independent Set von größter Kardinalität, d.h. zu finden ist ein $I^* \in \mathcal{I}_G$, sodass $\forall I \in \mathcal{I}_G \colon |I^*| \geq |I|$.

Als praktische Motivation können wir uns einen *Konfliktengraphen* vorstellen. Dieser beinhaltet gleichwertige Elemente (Knoten), welche zum Teil miteinander in Konflikt stehen (Kanten). Ziel ist es, eine größtmögliche, konfliktfreie Menge auszuwählen. Elemente könnten beispielsweise Kurse in einem Semester sein und Konflikte, ob Paare von Kursen überlappende Zeiten haben.

5.2 Probleminstanzen

Für unsere Programmieraufgabe widmen wir uns *Zufallsgraphen* nach dem Erdős–Rényi $G(n, p)$-Modell [1]. In diesem wird die Anzahl der Knoten n vorgegeben und eine Wahrscheinlichkeit p mit der jeweils eine der $n(n-1)/2$ potenziellen Kanten auftritt. Die jeweilige Dichte $d(G) := 2m/n(n-1)$ folgt somit einer Binomialverteilung $\mathcal{B}(n, p)$, womit für den Erwartungswert der Dichte $\mathbb{E}[d(G)] = p$ gilt. Wir variieren $p \in \{0.00, 0.05, \ldots, 0.95, 1.0\}$ und erzeugen jeweils einen entsprechende Zufallsgraphen mit $n = 40$.

Zum Vergleich lösen wir zur bewiesenen Optimalität vorab etwas schwierigere Instanzen ($n = 100$, mehr Werte für p, jeweils 20 Grahpen je p) mit einem State-of-the-Art Optimierungstool[1] und betrachten den Verlauf der im Mittel expandierten Branch-and-Bound Teilprobleme (Branch-and-Bound Knoten) als Metrik für die praktische Schwierigkeit über die unterschiedlichen Kantenwahrscheinlichkeiten. Das Ergebnis ist in Abbildung 1 zu sehen, wobei die Interpretation auch Teil der Aufgabenstellung ist. Dieses soll als Referenz und zur Überprüfung für die Ergebnisse mit unserem Solver dienen, welchen wir nun genauer besprechen wollen.

5.3 Branch-and-Bound Ansatz

Zunächst modellieren wir das Problem als *Binary Integer Program*. Dazu geben wir je Knoten $v \in V$ eine binäre Entscheidungsvariable $x_v \in \{0, 1\}\ \forall v \in V$ vor. Das MISP lässt sich dann schreiben als

$$\max \sum_{v \in V} x_v \tag{1}$$

$$\text{sodass } x_v + x_w \leq 1 \quad \forall \{v, w\} \in E \tag{2}$$

$$x_v \in \{0, 1\} \quad \forall v \in V \tag{3}$$

Wir definieren nun Teilprobleme $P_U(D)$ bei dem über eine Menge von Knoten $U = \{u_1, \ldots, u_k\} \subseteq V$ bereits enschieden wurde, durch Festlegung der zugehörigen Entscheidungsvariablen auf $D = \{d_{u_1}, \ldots, d_{u_k}\} \subseteq \{0, 1\}^k$, wobei immer alle Nachbarn eines Knotens, der auf 1 gesetzt wurde, notwendigerweise auf 0 gesetzt werden. Das Optimierungsziel von $P_U(D)$ ist nun $\max \left(\sum_{u \in U} d_u + \sum_{v \in V \setminus U} x_v \right)$ mit den beschriebenen Constraints (2) und (3) auf dem durch $V \setminus U$ induzierten Subgraphen. NB: durch diese Konstruktion existiert keine Kante von einem $u \in U$ zu einem $v \in V \setminus U$ mit $d_u = 1$, womit für den durch $V \setminus U$ induzierten Subgraphen ein unabhängiges MISP betrachtet werden kann.

Bei Branch-and-Bound gilt es ausgehend vom Wurzelproblem $P_\emptyset$, das Problem in disjunkte Teilprobleme zu zerlegen und diese rekursiv zu lösen. Zur Beschleunigung des Lösungensverfahrens, versucht man bestimmte Teilprobleme möglich früh durch Schlussfolgerungen, dass dort nicht die global optimale Lösung zu finden sein kann, "abzuschneiden".

[1] Gurobi, ohne Presolving und Cuts

5.3.1 Obere und untere Schranken

Wir betrachten nun obere und untere Schranken des MISP auf dem durch $V \setminus U$ induzierten Subgraphen $G' = (V \setminus U, \{\{v, w\} \in E \mid v, w \in V \setminus U\})$ mit n' Knoten und m' Kanten. Schranken für ein zugehöriges Teilproblem $P_U(D)$ lassen sich dann durch einfaches Addieren von $\sum_{u \in U} d_u$ (Anzahl der bisher ins Independent Set gewählten Knoten) berechnen. Folgende Schrankenfunktionen sind zu implementieren:

- `int trivialUpperBound(SubGraph g)`: eine triviale obere Schranke ist die Anzahl der Knoten im Subgraphen n':

$$b_{\mathrm{u}}^{\mathrm{trivial}}(n') = n'. \tag{4}$$

- `int hansenUpperBound(SubGraph g)`: eine etwas stärkere obere Schranke ist die sogenannte *Hansen-Bound* [2], welche auch die Anzahl der Kanten des induzierten Subgraphen m' mitberücksichtigt:

$$b_{\mathrm{u}}^{\mathrm{hansen}}(n', m') = \left\lfloor \frac{1}{2} + \sqrt{\frac{1}{4} + n'^2 - n' - 2m'} \right\rfloor \tag{5}$$

- `int vertexCoverBasedHeuristic(SubGraph g)`: Überlegen Sie sich, wie Sie aus einem (beliebigen) greedy Vertex Cover eine untere Schranke für das MISP ableiten können.

Zur Berechnung der Schranken stehen Ihnen folgende Methoden der Subgraph-Klasse zur Verfügung:

- `int numberOfVertices()`: gibt die Anzahl der Knoten n' im induzierten Subgraphen zurück.
- `int numberOfEdges()`: gibt die Anzahl der Kanten m' im induzierten Subgraphen zurück.
- `int greedyVertexCoverSolutionValue()`: erzeugt ein greedy Vertex Cover nach dem in der Vorlesung präsentierten Greedy-Verfahren und gibt dessen Kardinalität zurück.

Die von Ihnen implementierten Schrankenfunktionen werden von der Klasse `class SubProblem` im Weiteren verwendet. Sie können diese getrennt über die Testinanzen `small-misp-root.csv` testen, ohne dass der vollständige Branch-And-Bound Solver schon vorhanden sein muss, welchen wir als nächstes implementieren werden.

5.3.2 Implementation Branch-and-Bound Solver

Implementieren Sie nun einen in der Vorlesungen präsentierten Pseudocode folgenden Branch-and-Bound Algorithmus für das MISP. Als eine wesentlichen Datenstruktur steht Ihnen schon ein Branch-and-Bound Teilproblem `class SubProblem` zur Verfügung. Dieses bietet folgende für Sie relevante Methoden:

- `int lowerBound()`, gibt eine lokale untere Schranke zurück.
- `int upperBound()`, gibt eine lokale obere Schranke zurück.
- `void branch(SubProblemQueue Q)`, erzeugt zwei neue disjunkte Teilprobleme und fügt diese der `SubProblemQueue Q` hinzu.

Ihr Solver wird über die Methode `int maximumIndependentSetBranchAndBoundSolver(SubProblem rootProblem, SubProblemQueue Q)` aufgerufen, wobei `rootProblem` dem Wurzelproblem $P_\emptyset$ entspricht und `SubProblemQueue Q` die anfänglich leere Queue zur Abarbeitung der Teilprobleme darstellt. Für letztere stehen Ihnen folgende Methoden zur Verfügung:

- `void add(SubProblem p)`: fügt ein Teilproblem der Queue hinzu.
- `SubProblem poll()`: entfernt und gibt das nächste Teilproblem zurück.
- `boolean isEmpty()`: fragt, ob die Queue leer ist.

Das bewiesene Optimum (globale untere Schranke) soll zurückgegeben werden. Unterschiedliche Strategien zur Teilproblem-Selektion (Breitensuche `fifo` vs. Best-First-Search geordnet nach der obere Schranke `best`) und für das Branching (lexikographisch `lex` vs. ein Knoten mit maximalem Grad in G' `maxdegree`) sind bereits vorgegeben und werden nun auch, neben den Dualheuristiken `trivial` und `hansen`, mit in den Experimenten verglichen. Vorab können Sie kleinere Tests Ihres Solvers mit `small-misp.csv` durchführen.

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.

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

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 die Anzahl der expandierten Branch-and-Bound Teilprobleme 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 theoretische und praktische Fragestellungen:

1. Ist das MISP ein schwieriges Problem? Was können Sie zu dessen Komplexität sagen? Überlegen Sie sich Klassen von Graphen, für welche das MISP besonders einfach wird.

2. Erkennen Sie Phasen (Schwierigkeitsbereiche) in Abbildung 1? Können Sie Übergänge ausmachen? Wie erklären Sie sich das Zustandekommen von Abbildung 1 konkret? *Hinweis: Überlegen Sie sich, was es bedeutet, wenn wir sehr wenige bzw. sehr viele Kanten in unserem Zufallsgraphen haben.*

3. Erstellen Sie einen Screenshot Ihres Plots mit allen acht Konfigurationen. Erkennen Sie ähnliche Phasenübergänge wie in Abbildung 1?

4. Erstellen Sie einen Screenshot, bei der nur die Ihrer Meinung nach beste und eine schlechteste Konfiguration hinsichtlich unserer Metrik angezeigt werden. Um bis zu welcher Größenordnung unterscheiden sich diese im schwierigen Bereich?

5. Konkret für unsere Implementierung und betrachteten Instanzen, ordnen Sie die drei Dimensionen (Teilproblem-Selektion, Branching, Dualheuristik) nach Einfluss auf die Anzahl der expandierten Teilproblem. Wie erklären Sie sich den Unterschied bei den beiden Branching-Strategien?

6. Wählen Sie die vier Konfiguration `rand-n40-best-*` aus und lassen Sie sich die Werte der vier Datenpunkte beim Maximum anzeigen und machen einen Screenshot davon. Lässt sich aus der Anzahl der Knotenexpansionen unmittelbar etwas über die Laufzeit aussagen?

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 den drei erstellten Bildern der Visualisierungen der Testinstanz `abgabe.csv` zusammen.

9 Abgabe

Laden Sie die Datei `src/main/java/exercise/StudentSolutionImplementation.java` in der

TUWEL-Aktivität *Hochladen Source-Code P5* 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.

10 Nachwort

Das Branch-and-Bound Paradigma im Algorithmenentwurf steht für eine sehr weit verbreitete Klasse von Algorithmen zur Lösung von NP-schweren kombinatorischen Optimierungsproblemen. Man kann damit einerseits auf möglichst geschickte Weise (eben durch Verzweigung der Suche und Verwerfen von unrentablen Lösungsästen) versuchen exakte Lösungen in akzeptabler Laufzeit zu finden, oder eben auch heuristisch in kürzerer Zeit Lösungen finden, die gut genug sind, sobald untere und oberere Schranken sich hinreichend angenähert haben. Beispiele von NP-schweren Optimierungsproblemen, die sich mit Branch-and-Bound oftmals gut lösen lassen, sind Tourenplanungsprobleme wie das klassische TSP Problem, Planung von Personaleinsatz oder Ablaufpläne in der industriellen Produktion.

Branch-and-Bound Algorithmen sind also sehr flexibel einsetzbare Verfahren, die sowohl in der Industrie zur Lösung von komplexen diskreten Optimierungsproblemen verwendet werden, als auch in aktuellen Forschungsfragen in der Algorithmik. Außerdem liefert die Branch-and-Bound Idee auch clevere Lösungsverfahren für die ganzzahlige lineare Programmierung in der diskreten/kombinatorischen Optimierung. Solche Techniken werden dann später im Master beispielsweise in 186.814 VU Algorithmics oder 186.835 Mathematical Programming behandelt. In Ihrem weiteren Studium können Ihnen Branch-and-Bound Algorithmen außerdem in Seminaren begegnen oder aber auch ganz konkret zu eigenen erfolgreichen Implementierungen von Algorithmen im Rahmen von Projektarbeiten oder Ihrer Bachelor- bzw. Masterarbeit beitragen. Für jede Informatikerin und jeden Informatiker sollten sie einen festen Platz im algorithmischen Werkzeugkasten haben.

Literatur

[1] Paul Erdős and Alfréd Rényi. On the evolution of random graphs. *Publ. Math. Inst. Hung. Acad. Sci*, 5(1):17–60, 1960. `https://www.renyi.hu/`

~p_erdos/1960-10.pdf.

[2] Pierre Hansen and Maolin Zheng. Sharp bounds on the order, size, and stability number of graphs. *Networks*, 23(2):99–102, 1993.