Übungen zur Vorlesung

Algorithmen und Datenstrukturen

Übungsblatt 10

Arbeitsgruppe Kryptographie und Komplexitätstheorie
Prof. Dr. Marc Fischlin
Dr. Christian Janson
Patrick Harasser
Felix Rohrbach

Sommersemester 2019
Veröffentlicht am: 21.06.2019
Abgabe am: 05.07.2019, 12:00 Uhr

P1 (Gruppendiskussion)

Nehmen Sie sich etwas Zeit, um die folgenden Fachbegriffe in einer Kleingruppe zu besprechen, sodass sie anschließend in der Lage sind, die Begriffe dem Rest der Übungsgruppe zu erklären:

(a) Amortisierte Analyse (Account-Methode, Potential-Methode)

P2 (Amortiesierte Analyse)

Ein MultipopStack ist eine Datenstruktur, die aus einem herkömmlichen Stack durch Hinzunahme einer Operation Multipop hervorgeht. Ein MultipopStack S besitzt also die zwei üblichen Stack-Methoden S.Push und S.Pop, und dazu noch eine Methode S.Multipop, welche die k obersten Objekte vom Stack entfernt (bzw. diesen vollständig leert, wenn weniger als k Objekte vorhanden sind). Die Multipop-Methode kann in Pseudocode wie folgt beschrieben werden:

S.Multipop(k)

```
11 : while ¬StackEmpty(S) ∧ k > 0 do
12 :     S.Pop
13 :     k = k − 1
```

Hier gibt StackEmpty(S) den Wert **true** aus, wenn sich aktuell keine Elemente im Stack befinden, ansonsten **false**.

In dieser Aufgabe sollen Sie die Methoden aus der amortisierten Analyse verwenden, um die Laufzeit $T(n)$ von n beliebigen aufeinanderfolgenden Push-, Pop- und Multipop-Operationen auf einem anfangs leeren MultipopStack S abzuschätzen. Nehmen Sie einfachheitshalber an, dass die Kosten einer einzelnen Push- und Pop-Operation gleich 1 sind.

(a) Berechnen Sie die Kosten von S.Multipop(k) auf einem MultipopStack S mit s Objekten.
(b) Zeigen Sie mittels herkömmlicher Worst-Case Analyse, dass $T(n) = O(n^2)$.
(c) Benutzen Sie nun die Account-Methode und die Potential-Methode, um zu zeigen, dass tatsächlich $T(n) = O(n)$.

Im restlichen Teil dieses Übungsblattes beschäftigen wir uns mit dem *String-Matching*-Problem: Es sollen Algorithmen zum Finden von Textsegmenten in einer Zeichenkette anhand eines vorgegebenen Suchmusters erarbeitet werden.

Wir formulieren das String-Matching-Problem wie folgt. Der zu durchsuchende Text entspricht einem Array T der Länge n, und das Textmuster einem Array P der Länge $m \leq n$. Wir nehmen an, dass die Elemente von P und T Zeichen aus einem endlichen Alphabet Σ sind. Gesucht sind nun alle gültigen "Verschiebungen", mit denen P in T auftaucht. Mit anderen Worten: Es müssen alle $s \in \mathbb{N}$ bestimmt werden, sodass $T[s..s+m-1] = P[0..m-1]$, also $T[s+j] = P[j]$ für alle $0 \leq j \leq m-1$.

P3 (String-Matching 1 – Naiver Algorithmus)

Wir beginnen in dieser Übung mit dem naiven String-Matching-Algorithmus. Dieser Algorithmus bestimmt mithilfe einer Schleife alle gültigen Verschiebungen, indem für alle möglichen Werte $0 \leq s \leq n-m$ überprüft wird, ob $P[0..m-1]$ und $T[s..s+m-1]$ übereinstimmen.

(a) Geben Sie den Pseudocode des naiven String-Matching-Algorithmus NaiveStringMatching an. Der Algorithmus soll den zu durchsuchenden Text T und das Textmusterarray P als Eingabe bekommen, und die Liste aller korrekten Verschiebungen ausgeben.
(b) Beweisen Sie die Korrektheit und analysieren Sie die Laufzeit von NaiveStringMatching.
(c) Betrachten Sie den Text $T =$ "hehehhheyh" und das Muster $P =$ "heh". Benutzen Sie den naiven String-Matching-Algorithmus, um alle Vorkommen von P in T zu bestimmen.

Wie Sie in der letzen Aufgabe sicher bemerkt haben, ist der String-Matching-Algorithmus NaiveStringMatching relativ ineffizient. Grund dafür ist, dass die Informationen über den Text T, die bei der Behandlung eines bestimmten Wertes von s gewonnen werden, bei der Bearbeitung anderer Werte nicht miteinfließen. Solche Informationen können aber sehr wertvoll sein, und die Laufzeit des Algorithmus deutlich verkürzen.

In den nächsten Übungen werden Sie zwei alternative Ansätze untersuchen, mit denen versucht wird, diese Informationen einzusetzen. Beide Ansätze benötigen eine Vorverarbeitungsphase, um das Suchen des Textmusters danach zu beschleunigen.

P4 (String-Matching 2 – Rabin-Karp Algorithmus)

Dieser Algorithmus basiert auf elementaren zahlentheoretischen Begriffen, wie z.B. der Gleichheit zweier Zahlen modulo einer dritten Zahl. Deshalb ist es notwendig, das Alphabet Σ (mit $|\Sigma| = d$) zunächst mit den Zahlen $\{0, \ldots, d-1\}$ zu identifizieren. Wir nehmen hier einfachheitshalber an, dass $\Sigma = \{0, \ldots, 9\}$, sodass wir wie gewohnt im Dezimalsystem ($d = 10$) arbeiten können. Wenn z.B. Σ das deutschen Alphabet wäre, dann müsste man in Basis $d = 26$ arbeiten.

Der Rabin-Karp Algorithmus basiert auf folgender Überlegung: Für jedes $0 \leq s \leq n - m$, sei t_s der Dezimalwert des Teilstrings $T[s..s+m-1]$, und sei p der Dezimalwert von P. Offenbar ist s eine gültige Verschiebung, also $T[s..s+m-1] = P$, genau dann wenn $t_s = p$.

(a) Zeigen Sie, wie man p und t_0 in Zeit $\Theta(m)$ berechnen kann.
(b) Überlegen Sie sich, wie man t_{s+1} aus t_s in konstanter Zeit berechnen kann.
(c) Geben Sie einen Algorithmus RabinKarpMatchBasic an, der die oben erläuterte Vorgehensweise benutzt, um das String-Matching-Problem zu lösen. Der Algorithmus soll das zu durchsuchende Array T und das Musterarray P als Eingabe bekommen, und die Liste aller korrekten Verschiebungen ausgeben.

Wir haben bisher in unserer Vorgehensweise ein kleines Problem ignoriert: Mit zunehmender Länge des Suchmusters können die Zahlen p und t_s sehr groß werden. Es könnte unter Umständen nicht angemessen sein, wenn wir voraussetzen, dass jede arithmetische Operation auf dem Wert p nur "konstante Zeit" benötigt.

Dieses Problem kann wie folgt gelöst werden: Sei q eine Primzahl, sodass $10q$ in ein Computerwort passt. Der Trick ist nun, die Werte p und t_s einfach modulo q zu berechnen und zu vergleichen; damit werden die betrachteten Zahlen um einiges kleiner. Man beachte allerdings, dass diese Lösung nicht perfekt ist: Aus $t_s \equiv p \pmod{q}$ folgt nicht $t_s = p$. Auf der anderen Seite gilt mit Sicherheit $t_s \neq p$ wenn $t_s \not\equiv p \pmod{q}$.

Man kann die Kongruenz $t_s \equiv p \pmod{q}$ also als einen schnellen Test verwenden, um ungültige Verschiebungen auszuschließen. Wenn allerdings $t_s \equiv p \pmod{q}$ für eine Verschiebung s gilt, muss nochmal explizit nachgeprüft werden, ob $T[s..s+m-1] = P[0..m-1]$, also ob s auch eine gültige Verschiebung ist, oder ob ein *unechter* Treffer vorliegt.

(d) Passen Sie Ihren Algorithmus RabinKarpMatchBasic an, sodass der eben beschriebene Test verwendet wird. Der resultierende Algorithmus RabinKarpMatch soll zusätzlich zu den vorherigen Parametern auch die Primzahl q als Eingabe bekommen.
(e) Finden Sie mithilfe des Rabin-Karp Algorithmus alle Vorkommen des Integer-Arrays $P = [7, 3, 4]$ in $P = [6, 9, 1, 7, 3, 4, 5, 0, 9, 4, 6, 2, 4, 8, 7, 3, 4]$. Verwenden Sie dabei $q = 13$. Sie können folgende Tabelle als Hilfsmittel benutzen:

s	t_s	$t_s \pmod q$	$t_s == p \pmod q$	$T[s..s+m-1] == P$	Treffer?
0					
1					
2					
3					
4					
5					
6					
7					
8					
9					
10					
11					
12					
13					
14					

P5 (String-Matching 3 – Endliche Automaten)

Der letzte Ansatz den wir hier verfolgen zeigt, dass es nach einem cleveren Preprocessing des Musters P ausreicht, den Text T genau einmal von links nach rechts durchzugehen, um das String-Matching-Problem zu lösen.
Dieser Ansatz basiert auf folgender Überlegung: Das Problem wäre einfach lösbar, wenn wir beim Durchgehen von T bei jedem Zeichen wüssten, wie groß die Überlappung zwischen den zuletzt gelesenen Zeichen von T und dem Anfang von P ist (ob sich also ein 'Match' anbahnt oder nicht). Man könnte dann nämlich eine korrekte Verschiebung berechnen, sobald man auf eine lückenlose Übereinstimmung zwischen P und einem gewissen Teilstring von T stößt.
In der Vorverarbeitungsphase sollen genau diese Informationen gesammelt werden. Angenommen, $P = P[0..m-1]$ ist ein Muster der Länge m, und beim Durchgehen von T sind wir gerade auf eine Übereinstimmung der letzten j gelesenen Symbole von T und $P[0..j-1]$ $(0 \leq j \leq m)$ gestoßen. Wie verändert sich die Länge dieser Überlappung, wenn wir zum nächsten Symbol weitergehen? Diese Informationen können effektiv durch einen endlichen Automaten dargestellt werden.

Informell ist ein endlicher Automat eine Maschine, die den gegebenen String T einmal von links nach rechts durchließt. Dabei kann sich der Automat in einem von $m+1$ vielen internen Zuständen befinden (einer pro Größe der Überlappung zwischen den zuletzt gelesenen Zeichen von T und dem Anfang von P), und der Zustand kann sich nach dem Lesen jedes einzelnen Buchstaben ändern, in Anhängigkeit nur vom aktuellen Zustand und dem gerade gelesenen Symbol.

Allgemein gilt, dass sich der Automat im Zustand $0 \leq j \leq m$ befindet genau dann, wenn die letzten j gelesenen Zeichen von T mit den ersten j Symbolen von P übereinstimmen, und es kein $i > j$ mit derselben Eigenschaft gibt. Die Zustandsübergänge werden so gewählt, dass diese Eigenschaft beim Lesen jedes neuen Symbols immer erhalten bleibt: Für jeden Zustand $0 \leq j \leq m$ und jedes Zeichen $a \in \Sigma$ muss die große Überlappung zwischen den letzten Symbolen von $[P[0], \ldots, P[j-1], a]$ und dem Anfang von P bestimmt werden – Dies ist dann der Zielzustand $\delta(j, a)$, ausgehend von Zustand j mit Eingabe a.

(a) Angenommen, Sie haben für ein Textmuster P den oben beschriebenen Automaten mit Übergangsfunktion δ. Geben Sie einen Algorithmus FiniteAutomationMatch an, der den Automaten benutzt, um das String-Matching-Problem zu lösen. Der Algorithmus hat Zugriff auf die Übergangsfunktion δ und soll das zu durchsuchende Array T und die Länge m des Textmusters als Eingabe bekommen, um dann die Liste aller korrekten Verschiebungen auszugeben.

(b) Betrachten Sie nun das Alphabet $\Sigma = \{\mathrm{a}, \mathrm{g}, \mathrm{n}, \mathrm{o}\}$ und das Muster $P =$ "nano". Zeichnen Sie nach den oben erklärten Regeln den etsprechenden endlichen Automaten und benutzen Sie ihn, um den Text $T =$ "ganannanonananog" nach dem Muster P zu durchsuchen.

H1 (Travelling Salesman) (3+2+5+2+3 Punkte)

Das Travelling-Salesman-Problem ist, für eine gegebene Menge an Städten die kürzeste Rundreise zu finden, die alle Städte besucht. Zunächst sollen Sie das Problem mit Hilfe vom Backtracking lösen und anschließend zwei Optimierungen einbauen, die die Laufzeit des Algorithmus verbessert.

Bitte lesen Sie sich zunächst die allgemeinen Hinweise für die praktischen Übungen auf Moodle durch! Laden Sie sich anschließend das vorgegebene Framework herunter, um die Aufgabe implementieren zu können. Bitte entfernen Sie vor der Abgabe alle eigenen `System.out.println`s, damit Ihr Tutor die Ausgabe der Tests nicht suchen muss!

(a) Zunächst sollen Sie die Städte aus einer Datei einlesen. Die Städte sind dabei durch ihre x- und y-Koordinaten gegeben, welche durch ein Semikolon getrennt sind. Die Koordinaten sollten als Datentyp `double` eingelesen werden. Pro Zeile wird genau eine Stadt angegeben. Leere Zeilen sollen übersprungen werden. Weiterhin soll jeder Stadt eine ID vergeben werden, die bei 0 beginnt und einfach hochgezählt wird. Implementieren Sie diese Funktionalität in der Methode `readFile` in der Klasse `CityParser`.

(b) Im Backtracking-Algorithmus werden Sie sehr oft den Abstand zwischen zwei Städten berechnen müssen. Um das Backtracking zu beschleunigen, sollen Sie daher alle Abstände vorberechnen. Implementieren Sie dies in der Methode `buildDistanceMap` der Klasse `AbstractTSPSolver`. Sie sollen dabei die Abstände in dem zweidimensionalen Array `_distanceMap` speichern, sodass beispielsweise der Abstand zwischen der 3. und 10. Stadt im Array an der Stelle `_distanceMap[2][9]` (und natürlich auch an der Stelle `_distanceMap[9][2]`) steht. Da die IDs der Städte durchnummeriert sind, entspricht `City.id()` der Position im Array.

(c) Nun sollen Sie den eigendlichen Backtracking-Algorithmus implementieren. Dieser wird in einer abstrakten Klasse (`AbstractTSPSolver`) implementiert, damit Optimierungen für den Algorithmus in abgeleiteten Klassen implementiert werden können. Eine einfache abgeleitete Klasse ohne weitere Optimierungen ist `BasicTSPSolver`.
Da das Travelling-Salesman-Problem nach einer Rundreise fragt, macht es keinen Unterschied, in welcher Stadt wir beginnen. Beginnen Sie deshalb immer in der ersten Stadt (also der Stadt mit der ID 0). Implementieren Sie in der Methode `solve` einen rekursiven Backtracking-Algorithmus, der die bisherige Strecke weiterführt, indem er eine bisher unbesuchte Stadt zur Rundreise hinzufügt. Falls alle Städte besucht wurden, überprüfen Sie, ob diese Rundreise besser ist als alle bisherigen (bedenken Sie, dass Sie für die Länge die Strecke von der letzten Stadt zurück zur Stadt 0 mit einberechnen müssen!). Am Ende speichern Sie die beste Rundreise in der Variable `_solution` (dabei beginnt `_solution` immer mit der Stadt 0 und enthält jede Stadt genau einmal) und die Länge dieser Reise in der Variable `_length`.
Um später die Optimierungen entwickeln zu können, müssen Sie im Backtracking-Algorithmus noch die zwei abstrakten Funktionen aufrufen: `notifyNewBest` rufen Sie jedes Mal auf, wenn Sie eine neue bisher beste Lösung gefunden haben. Die Methode `prune` rufen Sie in jedem Rekursionsschritt mit dem aktuellen (unvollständigen) Weg auf. Falls `prune` **`true`** zurückgibt, soll dieser Weg nicht weiter verfolgt werden, da die Optimierung herausgefunden hat, dass eine Weiterführung dieses Weges nicht mehr zu einer optimalen Lösung führen kann.
Eine gute Implementierung sollte den Testfall `berlin11.txt` in unter einer Sekunde und den Testfall `berlin12.txt` in weniger als 10 Sekunden schaffen.

(d) Die beiden Optimierungen sollen in der Klasse `OptimizedTSPSolver` implementiert werden. Dabei muss bei der Initialisierung der Klasse angegeben werden, ob und welche der beiden Optimierungen benutzt werden soll (auch beide Optimierungen gleichzeitig ist eine mögliche Option).
Zunächst sollen Sie eine Optimierung implementieren, die den jeweils aktuellen Weg mit der aktuell besten (vollständigen) Lösung vergleicht. Falls der aktuelle Weg plus die Distanz vom der letzten Stadt zurück zur ersten schon größer ist als die aktuell beste Lösung, kann dieser Weg niemals zu einer optimalen Lösung führen (da alle Abstände zwischen den Städten nicht negativ sind). Daher soll in diesem Fall beim Aufruf von `prune` **`true`** zurückgegeben werden. Diese Optimierung soll aktiviert werden, wenn `useLengthPruning` auf **`true`** gesetzt wurde.
Eine gute Implementierung sollte mit dieser Optimierung auch `berlin13.txt` in unter einer Sekunde schaffen.

(e) Die zweite Optimierung nutzt aus, dass die Städte auf einer zweidimensionalen Karte angeordnet sind. In diesem Fall gilt, dass eine Rundreise, deren Wege sich kreuzen, niemals optimal sein kann. Sie sollen daher diese Fälle aussortieren.
Angenommen, Sie wollen überprüfen, dass die Strecke zwischen Stadt 1 und Stadt 2 nicht die Strecke zwischen den

Städten Stadt 3 und Stadt 4 kreuzt. Sei x_i die x-Koordinate für die i-te Stadt und y_i entsprechend die y-Koordinate der i-ten Stadt. Berechnen Sie[1] :

$$t_a = \frac{(y_3 - y_4)(x_1 - x_3) + (x_4 - x_3)(y_1 - y_3)}{(x_4 - x_3)(y_1 - y_2) - (x_1 - x_2)(y_4 - y_3)}$$
$$t_b = \frac{(y_1 - y_2)(x_1 - x_3) + (x_2 - x_1)(y_1 - y_3)}{(x_4 - x_3)(y_1 - y_2) - (x_1 - x_2)(y_4 - y_3)}.$$

Falls $0 < t_a < 1$ und $0 < t_b < 1$ sind, treffen sich die beiden Strecken in der Mitte. Überprüfen Sie in der Methode `prune`, ob die letzte hinzugefügte Strecke eine der vorigen Strecken kreuzt. Wenn ja, soll dieser Weg nicht weiter verfolgt werden. Diese Optimierung soll über `useIntersectionPruning` eingeschaltet werden.

Hinweis: In dieser Übung wird jetzt die LinkedList-Klasse aus der Java-Standardimplemetierung verwendet. Diese verwendet sich etwas anders als die selbstimplementierte LinkedList aus dem letzten Praktikum. Über den Aufruf von `listIterator()` kann man auf einen ListIterator Zugriff bekommen, der Ähnlichkeiten zur ListNode-Klasse hat. Der größte Unterschied ist, dass der Iterator beim Aufruf von `next()` und `previous()` den nächsten Eintrag ausgibt und dann intern einen Schritt weiter- bzw. zurückgeht. Lesen Sie sich am besten ein bisschen in die Dokumentation der Klasse ein.

[1] Quelle: www.cs.swan.ac.uk/~cssimon/line_intersection.html