LEHRSTUHL FÜR RECHNERARCHITEKTUR UND PARALLELE SYSTEME

Grundlagenpraktikum: Rechnerarchitektur

Zusatzaufgabe 8: Union Find

1 Einführung

Datensätze können oft in disjunkte Gruppen Anhand verschiedener Kriterien unterteilt werden. Für diverse Algorithmen ist es notwendig, effiziente Abfragen zu diesen Gruppen auch für große Datensätze zu ermöglichen. Einer dieser Anwendungsfälle ist die Suche nach minimalen Spannbäumen auf Graphen, die für eine ganze Reihe an Anwendungsfällen interessant sind. In dieser Aufgabe untersuchen wir die Union-Find Datenstruktur, die uns erlaubt diverse Abfragen mit guter asymptotischer Laufzeit zu bearbeiten.

Sei n die Anzahl an Datenpunkten. Ein Datenpunkt wird durch einen Integer $\texttt{D}_\texttt{i} \in [0, n)$ eindeutig identifiziert. Eine Gruppe bestehend aus beliebig vielen Datenpunkten wird ebenso durch einen Integer $\texttt{G}_\texttt{i} \in [0, n)$ eindeutig identifiziert.

In der Union-Find Datenstruktur werden eine Vielzahl an Bäumen (ein sogenannter Wald an Bäumen) abgespeichert, wobei alle Knoten eines Baumes derselben Gruppe angehören.

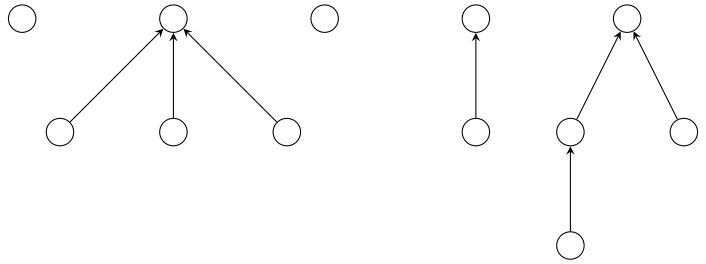


Abbildung 1: Ein Wald aus Bäumen, wie man ihn in einer Union-Find Datenstruktur finden könnte. Ein Baum kann durchaus nur aus einem Knoten bestehen.

Die Union-Find Datenstruktur unterstützt zwei Arten von Operationen: Find und Union. Im folgenden schauen wir uns diese genauer an.

Find

$\texttt{Find(D}_\texttt{i}\texttt{)} \rightarrow \texttt{G}_\texttt{i}$: Die Find Operation (Abbildung 2) berechnet die Gruppenzugehörigkeit eines Datenpunktes $\texttt{D}_\texttt{i}$. Sie gibt $\texttt{G}_\texttt{i}$ zurück, einen eindeutigen Identifier der Gruppe. Wir nennen $\texttt{G}_\texttt{i}$ den Repräsentanten von allen Datenpunkten in dieser Gruppe. Der Repräsentant der Gruppe ist äquivalent zu der Wurzel von dem Baum, der Datenpunkt $\texttt{D}_\texttt{i}$ enthält. Die Find Operation traversiert also den Baum von Knoten $\texttt{D}_\texttt{i}$ aufwärts bis zur Wurzel $\texttt{G}_\texttt{i}$ und gibt $\texttt{G}_\texttt{i}$ zurück.

Relinking: Während der Traversierung des Baumes werden alle Knoten auf dem Suchpfad, die nicht direkt unter dem Repräsentanten (der Wurzel) hängen, direkt unter dem Repräsentanten neu eingehängt. Dies hält die Bäume möglichst flach und verbessert damit die asymptotische Laufzeit. Mehr dazu im Beispiel.

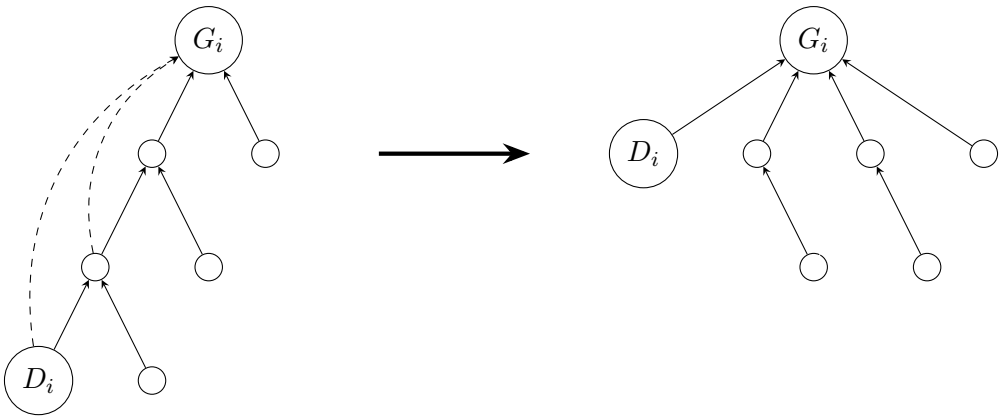


Abbildung 2: Das Relinking während einer Find Operation. Auf dem Suchpfad von $\texttt{D}_\texttt{i}$ nach $\texttt{G}_\texttt{i}$ finden wir mehrere Knoten, die nicht direkt unter $\texttt{G}_\texttt{i}$ hängen. Sobald $\texttt{G}_\texttt{i}$ gefunden wurde hängen wir diese Knoten direkt an die Wurzel (gestrichelte Linien).

Mit der Find Operation lässt sich zum Beispiel überprüfen, ob zwei Datenpunkte $\texttt{D}_1$, $\texttt{D}_2$ derselben Gruppe zugehören. Dies ist der Fall genau dann, wenn $\texttt{Find(D}_1\texttt{)} = \texttt{Find(D}_2\texttt{)}$.

Union

$\texttt{Union(D}_\texttt{i}\texttt{, D}_\texttt{j}\texttt{)} \rightarrow \texttt{G}_{\texttt{i/j}}$: Die Union Operation (Abbildung 3) vereint in der Datenstruktur die beiden Gruppen $\texttt{G}_\texttt{i} = \texttt{Find(D}_\texttt{i}\texttt{)}$ und $\texttt{G}_\texttt{j} = \texttt{Find(D}_\texttt{j}\texttt{)}$ miteinander zu einer größeren Gruppe. Dies passiert, indem einer der Repräsentanten (und damit dessen Baum) in den anderen Baum direkt unter dem anderen Repräsentanten eingehängt wird. Dabei wird immer die Gruppe mit weniger Datenpunkten in die Gruppe mit mehr Datenpunkten absorbiert. Wenn beide Gruppen dieselbe Größe aufweisen dann ist die Reihenfolge der Argumente ausschlaggebend, das heißt es wird $\texttt{G}_\texttt{j}$ in $\texttt{G}_\texttt{i}$ eingehängt. Zurückgegeben wird der Identifier der vereinten Gruppe.

Edge case: Der Fall, dass $\texttt{G}_\texttt{i}$ und $\texttt{G}_\texttt{j}$ gleich sind (das heißt $\texttt{D}_\texttt{i}$ und $\texttt{D}_\texttt{j}$ befinden sich bereits in einer Gruppe), muss gegebenenfalls gesondert behandelt werden.

2 Umsetzung & Beispiel

Für die Implementierung der Union-Find Datenstruktur verwenden wir eine Array-Darstellung des Waldes. Dafür benötigen wir zwei Arrays. Im ersten Array wird der

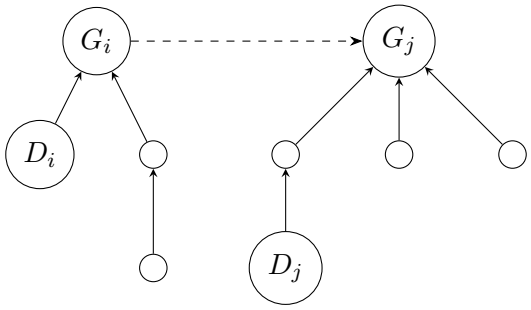
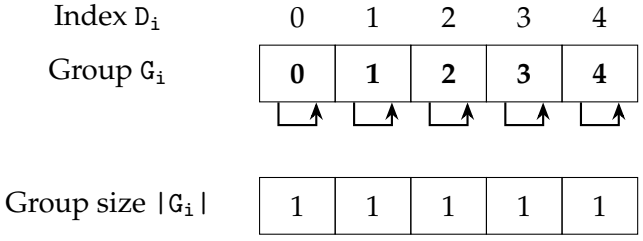
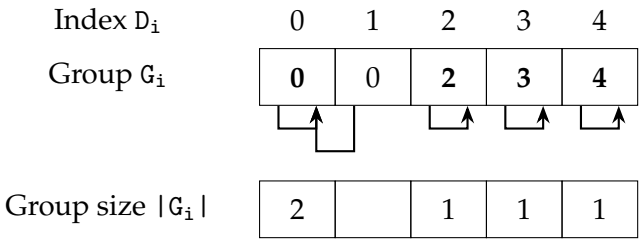


Abbildung 3: `Union(D_i, D_j)`. Weil der Baum von `G_j` größer ist als der von `G_i` wird `G_i` an `G_j` angehängt.

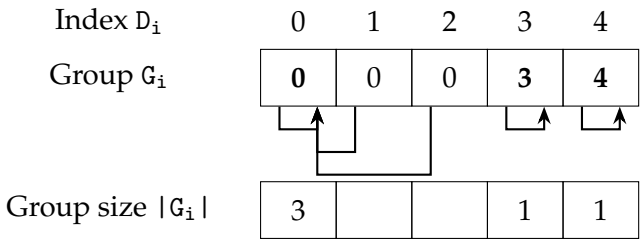
jeweilige Index des Elternknotens eines Datenpunktes im Baum gespeichert. Wenn der Knoten kein Elternknoten hat (und dementsprechend ein Repräsentant ist), dann speichern wir den eigenen Index im Array ab um die Repräsentanten identifizieren zu können. Ein Repräsentant lässt sich also finden, indem wiederholt der Elternknoten des aktuellen Knotens nachgeschlagen wird. Im zweiten Array wird die zugehörige Gruppengröße abgespeichert, die nur an den Repräsentanten relevant ist. Zu Beginn ist jeder Datenpunkt der Repräsentant seiner eigenen Gruppe.



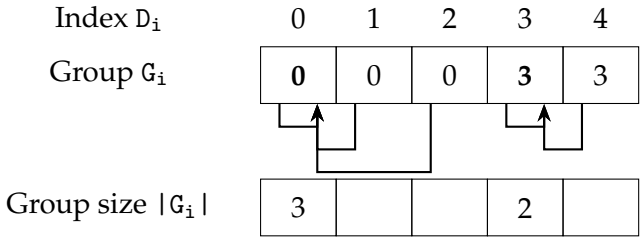
Wenn wir nun ein `Union` zwischen D_0 und D_1 ausführen, dann wird D_1 in die Gruppe von D_0 absorbiert. Die Größe der Gruppe wird beim Repräsentanten aktualisiert, beim anderen Datenpunkt ist sie nicht mehr benötigt.



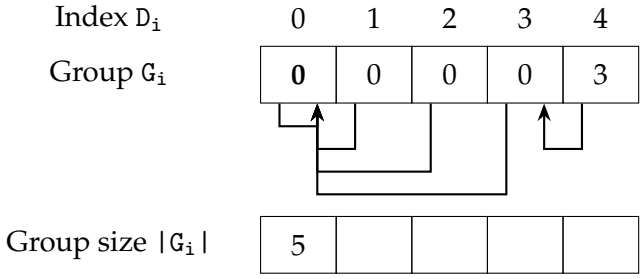
Wir können nun zum Beispiel ein `Union` zwischen D_1 und D_2 ausführen. Wir `Find`-en dafür die Repräsentanten von D_1 und D_2 und aktualisieren wieder deren Indices und Gruppengrößen.



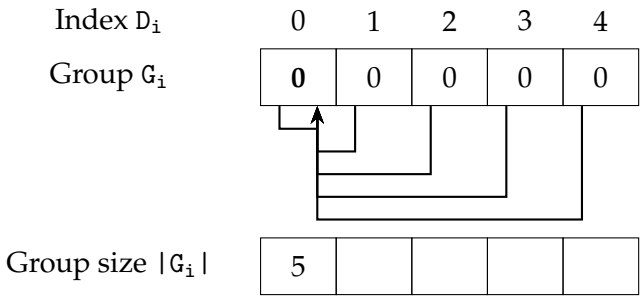
Nun können wir mit einem `Union` D_3 und D_4 diese beiden Gruppen zusammenfügen.



Schließlich können wir zum Beispiel `Union` D_4 und D_2 ausführen. Da wir immer nur die Repräsentanten der Gruppen verändern, schaut unsere Datenstruktur nun so aus:



Nun zeigt D_4 nicht mehr direkt auf den Repräsentanten, sondern nur noch indirekt. Unser Baum ist dementsprechend tiefer geworden. Wenn wir jetzt `Find`(D_4) ausführen, dann muss `Find` erst von Element 4 zu Element 3, und anschließend von Element 3 zu Element 0 suchen. Bei Element 0 wurde dann der Repräsentant gefunden, da $D_i = G_i$. Damit der Baum nicht stetig tiefer wird, aktualisieren wir alle Elemente auf dem Suchpfad von `Find` auf den aktuellen Repräsentanten. Danach schaut die Datenstruktur so aus:



Wenn wir nun erneut `Find(D4)` ausführen, dann finden wir den Repräsentanten schneller. Mit diesem neuverlinken in der Find Operation können wir auf einen Baum von n Knoten eine Sequenz von m `Union` oder `Find` Operationen mit einer Laufzeitkomplexität von $\mathcal{O}(m\alpha(n))$ ausführen, wobei α die überaus langsam wachsende inverse Ackermannfunktion ist.

Level

Neben dem tatsächlichen Resultat der Operationen `Find` und `Union` interessiert es uns auch noch, wie schnell unsere Operation durchgeführt wurde. In beiden Operationen gibt es nur eine Teiloperation, die nicht in konstanter Laufzeit durchgeführt werden kann: Die Suche nach dem Wurzelknoten und das anschließende Relinking. Daher würden wir gerne Wissen, wie viele Knoten unserer Algorithmus in jeder Operation inspizieren musste, bevor die Wurzel gefunden wurde. Diese Kennzahl nennen wir die Anzahl an traversierten Leveln. Ist der erste inspizierte Knoten direkt die Wurzel, dann ist die Anzahl an traversierten Leveln 0.

3 Aufgabe

Eure Aufgabe ist es, die Union-Find Datenstruktur in x86-64 Assembler zu implementieren. Die Funktionssignatur für eure Implementation ist wie folgt:

```
void unionfind(
    uint64_t setSize,
    char* instruction_string,
    char* solution_string);
```

Die Variable `setSize` ist n, die Anzahl der Datenpunkte D_i. Der `instructionString` ist ein ASCII String, der eine Sequenz an `Union/Find` Anweisungen (als `'U'` und `'F'` kodiert) enthält. In `solutionString` schreibt ihr einen ASCII String mit den Resultaten der Operationen.

Beispielein/ausgaben

`setSize`	`instructionString`	`solutionString`	Notiz
$n = 3$	`"F0F1F2"`	`"F0L0F1L0F2L0"`	Startkonfiguration
$n = 2$	`"U0&1F0F1"`	`"U0L0F0L0F0L1"`	`Union` + 2×`Find`
$n = 5$	`"U0&1U1&2U3&4U4&2F4F4"`	`"U0L0U0L1U3L0U0L2F0L2F0L1"`	Das Beispiel von oben
$n = 9999$	`"F1337"`	`"F1337L0"`	Ein einzelner `Find`

Tabelle 1: Diverse Beispiel Ein- und Ausgabestrings der Referenzimplementierung

Die Anweisung `'U'` (`Union`) nimmt 2 Operanden, die mit einem `'&'` getrennt sind. Die Anweisung `'F'` (`Find`) nimmt nur einen Operanden. Der `instructionString` ist

eine Aneinanderreihung von beliebig vielen dieser Anweisungen. Für jede Anweisung hängt ihr folgendes an den `solutionString`: die Art der Anweisung (`U/F`), das Resultat (der Repräsentant der Gruppe), ein Literal `'L'`, sowie die Gesamtanzahl an traversierten Leveln, die während der Ausführung der Anweisung angefallen sind[1].

Hilfsroutinen

malloc/free

In dieser Aufgabe stehen euch `malloc` und `free` zur Verfügung. Der verfügbare Speicher ist begrenzt, reicht jedoch in jedem Fall für Lösung der jeweiligen Aufgabe. Der dynamisch allozierte Speicher sollte wieder freigegeben werden.

getint

```
1 uint64_t getint(char *str)
```

Die Hilfsroutine `getint` ließt eine Zahl aus dem übergebenen String `str` und gibt sie als vorzeichenlose 64-Bit Zahl zurück. Es werden so lange Ziffern aus dem String gelesen, bis das erste Zeichen, dass nicht eine Ziffer ist, gefunden wird. Die Routine geht davon aus, dass auf die letzte Ziffer entweder ein weiteres Zeichen oder der Null-Terminator folgt.

Zusatzfeature: Obwohl es nicht ganz der Calling-Convention entspricht, dürft ihr davon ausgehen, dass nach dem ausführen von `getint` der übergebene `char* str` (bzw. dessen Register) auf das Zeichen hinter der letzten gelesenen Ziffer zeigt.

putint

```
1 char* putint(uint64_t value, char* string)
```

Die Hilfsroutine `putint` nimmt eine Zahl `value` und hängt sie Ziffer für Ziffer an den String `string` an. Das erste geschriebene Zeichen ist `string[0]`. Der Rückgabewert zeigt auf das Zeichen hinter der letzten geschriebenen Ziffer. Die Routine geht davon aus, dass in `string` immer genügend Platz vorhanden ist, um das Resultat zu schreiben. Der String wird nicht mit einem terminierenden Null-byte versehen.

Rahmenbedingungen

- $n \in [0, 65535]$, `strlen(instructionString)` $\in [0, 65535]$
- Ihr dürft davon ausgehen, dass `instructionString` valide ist. Insbesondere enthält `instructionString` keine ungültigen Zeichen oder Referenzen auf ungültige Indices.

[1] Während `Union` wird `Find` gegebenenfalls mehrere Male aufgerufen. Dies sollte in der Gesamtanzahl an traversierten Leveln berücksichtigt sein.

- Der `solutionString` enthält immer genügend Platz, um die Lösung dort abzuspeichern.
- Ihr dürft Online-Ressourcen verwenden. Wenn ihr kleine Codestücke übernehmt, verseht diese mit einer Quellenangabe im Kommentar. Der Großteil der Aufgabe ist jedoch selbstständig zu lösen.
- Achtet auf die Calling Convention, wenn ihr Hilfsfunktionen aufruft.
- Sobald die Aufgabe im Aufgabentester als korrekt gelöst markiert ist, habt ihr die Zusatzaufgabe bestanden.
- Plagiate oder die Nutzung von Compiler-generiertem Code führen zur Aberkennung der Zusatzaufgabe und damit zum nicht-bestehen des Praktikums.
- Eure Tutoren sind für euch da. Allerdings ist diese Aufgabe ein Teil eurer Prüfungsleistung. Inhaltliche Fragen können dementsprechend nur in sehr begrenztem Maße beantwortet werden.
- Abgabefrist: Auf Praktikumswebseite zu finden.

Changelog

23.05.2022 23:30 Specify range of n and fix range of $\texttt{D}_\texttt{i}$ and $\texttt{G}_\texttt{i}$.