Übungsblatt 1

Abgabe via Moodle. Deadline Fr., den 12. Juli

Um eine Klausurzulassung zu erhalten, müssen mindestens 25% der Punkte auf diesem Blatt erreicht werden! Es wird keine Fristverlängerung gewährt.

Mit Ausnahme Testfälle/4., sind alle Regeln identisch mit dem vorherigen Blatt.

Allgemeine Regeln

- Alle Aufgaben sind in C++17 zu bearbeiten. Im Zweifelsfall muss die Abgabe mittels `x64-64 gcc 14.1` auf `https://compiler-explorer.com/` funktionieren (wenn der entsprechende Code in eine einzelne Datei zusammen kopiert wurde).
- Ändern Sie **niemals** die Signatur von existierenden Funktionen. Wir verwenden zusätzliche Tests, die die vorgegebenen Signaturen benötigen. Sie dürfen aber neue Funktionen hinzufügen.
- Wir erwarten, dass alle Abgaben ohne Warnungen und Fehler kompilieren; i.A. bewerten wir nicht kompilierende Abgaben mit 0 Punkten (beachten Sie, dass wir `-Werror` nutzen und somit Warnungen als Fehler behandelt werden)
- Geben Sie alle Quelldateien (keine Kompilate oÄ) in einem ZIP-Archiv ab. Wenn Sie die Bonusaufgabe bearbeiten, fügen Sie zusätzlich die aufgezeichneten Rohdaten und deren Auswertung der ZIP-Datei bei.

Tipps

- Nutzen Sie das beigefügte Skript `build.sh` zum Kompilieren; evtl. muss die Zeile `CXX=g++` angepasst werden, falls Sie einen anderen C++ Compiler verwenden.
- Wenn Sie *nicht* das Skript nutzen, schalten Sie Warnung in Ihrem Compiler an, z.B.

```
g++     --std=c++17 -Wall -Wextra -Wpedantic -Werror -g -O0 tests.cpp -o tests
clang++ --std=c++17 -Wall -Wextra -Wpedantic -Werror -g -O0 tests.cpp -o tests
```

- Der Vorständigkeit-halber liegt dem Projekt auch eine `cmake` Konfigurationsdatei bei. Wir empfehlen Ihnen `Debug`-Builds zu nutzen, um möglichst aussagekräftige Fehlermeldungen zu bekommen. (z.B. mittels `cmake -DCMAKE_BUILD_TYPE=Debug ..`).

Testfälle

Sie werden wiederholt gebeten, Testfälle zu schreiben. Hierfür haben wir ein kleines Rahmenwerk erstellt. Um einen neuen Testfall zu schreiben, gehen Sie wie folgt vor:

1. Öffnen Sie die Datei `tests.cpp` und erstellen eine neue Funktion `bool test_XXX()` wobei XXX durch einen geeigneten Namen ersetzt werden soll.
2. Implementieren Sie den Test. Jede Testfunktion gibt `true` zurück, falls der Test klappte, und `false` sonst. Um den Vorgang zu vereinfachen, haben wir Ihnen folgende Hilfsfunktionen erstellt:

- `fail_if(cond)`: Wenn die Bedingung `cond` als `true` ausgewertet wird, wird eine Fehlermeldung ausgegeben und die Funktion direkt mit `false` verlassen.
- `fail_if_eq(a, b)`: Verursacht einen Fehler, wenn a und b gleich sind.
- `fail_unless(cond)`, `fail_unless_eq(a, b)`: die Negationen der beiden Varianten oben. So stellt `fail_unless_eq(a, b)` sicher, dass a und b identisch sind.

  Tipp: Beachten Sie, dass bei eingeschalteten Warnung z.B. auch geprüft wird, dass keine vorzeichenbehafteten Datentypen (etwa `int`) mit vorzeichenlosen Datentypen (etwa `size_t`) verglichen werden. Das tritt z.B. auf, wenn die Länge eines Containers mit einem `int` verglichen wird; ggf. müssen Sie also explizite Casts nutzen:

```
// Vollständiger Test siehe ‘test_push_front()‘ in ‘tests.cpp‘
for(int i=0; i<10; ++i) {
    fail_unless_eq(lst.size(), static_cast<size_t>(i));
    lst.push_front(i);
}
```

3. Rufen Sie die Funktion auf. Erstellen Sie hierzu in `main()` eine neue Zeile `run_test(test_XXX);`.
4. **Neu:** Im Vergleich zum ersten Blatt gibt es nun auch noch das Makro `run_test_all_ufs(test_XXX)`. Dieses erwartet eine Testfunktion `template <typename UnionFind> bool test_XXX()` und ruft den Test mit den vier in Aufgabe 1 beschriebenen UnionFind Datenstrukturen auf. Als Beispiel können Sie etwa die Funktion `test_union_find_small_hardcoded` in `tests.cpp` lesen.
5. Bauen Sie das Projekt (`bash build.sh`) und führen Sie den Test aus (`./tests`). Tipp: Wenn Sie Bash nutzen, können Sie beide Befehle verbinden: `bash build.sh && ./tests` . Dabei sorgt `&&` dafür, dass das Programm nur ausgeführt wird, wenn des Kompilieren klappte.

Testfälle sollten immer einen sinnvollen Umfang haben; so ist es z.B. wichtig zu prüfen, ob eine leere Folge korrekt sortiert wird — wäre dies jedoch der einzige Test eines Sortieralgorithmus, wäre dies nicht ausreichend.

Aufgabe 1 (*UnionFind*, $2+2+2+2$ *Punkte*)

Vervollständigen Sie die Implementierung der UnionFind Datenstruktur in der Datei `msf.hpp`.

Es gibt einige Spezialitäten:

- Die Klasse `UnionFind` wird mittels zweier Template-Parametern `template <bool PathCompression, bool UnionByRank>` während der Kompilierung konfiguriert. Die beiden Features soll genau dann von der Implementierung genutzt werden, wenn der entsprechende Parameter auf `true` gesetzt wird. Außerhalb der Klasse können Sie die konkreten Spezialisierungen mit sprechenden Namen nutzen:

```
using UnionFindNoPCNoRank = UnionFind<false, false>;
using UnionFindPCOnly     = UnionFind<true, false>;
using UnionFindRankOnly   = UnionFind<false, true>;
using UnionFindPCAndRank  = UnionFind<true, true>;
```

- Da der Name `union` in C++ ein reserviertes Schlüsselwort ist, nennen wir die Funktion `combine`. Combine erwartet zwei Element $x \in M_i$ und $y \in M_j$ und verbindet diese Mengen dann zur Menge $M_i \cup M_j$. Die eigentliche Arbeit wird aber –analog zur Vorlesung– von der Methode `link` geleistet.

- Statt eines primitiven Elternarrays, existiert die Klasse `Parents`, die das Elternarray verwaltet und jeden Zugriff darauf zählt. In der Klasse `UnionFind` können Sie den Elternknoten p eines Knotens u mittels `parents.get_parent(u)` anfragen und mit `parents.set_parent(u, p)` setzen. Wir zählen später die Zugriffe auf das Elternarray — nehmen Sie daher an, dass `get/set_parent` teure Operationen sind und vermeiden Sie offensichtlich unnötige Mehrfachzugriffe.

```
// statt
if (parents.get_parent(123) == 234) {
   return parents.get_parent(123);
}

// schreiben Sie besser
Node parent = parents.get_parent(123);
if (parent == 234) {
   return parent;
}
```

Teilaufgaben:

1. Implementieren Sie die Funktion `find` sowohl ohne als auch mit Pfadkompression. Wie in der Vorlage bereits erkennbar, können Sie mittels `if constexpr (PfadCompression) { ... }` prüfen, ob die Pfad-Kompression verwendet werden soll.
2. Implementieren Sie die Funktion `link(root_x, root_y)` sowohl ohne als auch mit Ranking; wenn Ranking deaktiviert ist, soll immer der Teilbaum von x unterhalb von y angehängt werden.
3. Implementieren Sie mehrere Testfälle, welche die korrekte Arbeitsweise von `find` und `combine` prüfen. Insb. soll auch geprüft werden, ob die Pfadkompression funktioniert; und zwar genau dann, wenn sie aktiviert ist (d.h. prüfen Sie auch, dass bei deaktivierter Pfadkompression keine Änderung gechieht).

   Zu Testzwecken hat die Klasse `UnionFind` eine Methode `get_parent`, die Sie in Ihren Testfällen nutzen können, um die Pfadkompression zu testen. Zudem verfügen alle `UnionFind` Varianten über die statischen Konstanten `path_compression` und `union_by_rank`, die immer den Wert der entsprechenden Template-Parameter haben und es Ihnen erlauben zu prüfen, welche Datenstruktur Ihnen vorliegt:

```
template <typename UnionFind> bool test_demo() {
   if (UnionFind::path_compression) {
      // hier muessen Sie testen, ob die Pfadkompression funktioniert
   } else {
      // und hier soll sie deaktiviert sein
   }

   return true;
}
```

4. Die Klasse `UnionFind` verfügt über das Datenfeld `num_groups`, das die Anzahl der verschiedenen Mengen abspeichert. Der Wert ist anfangs n und kann beim Aufruf von `link` abnehmen. Implementieren Sie die entsprechenden Updates von `num_groups` in `link`. Geben Sie Testfälle an, welche die korrekte Implementierung prüfen; die Testfälle sollen sowohl Aufrufe zu combine/links enthalten, welche `num_groups` verändern, als auch solche, bei denen `num_groups` unverändert bleibt.

Aufgabe 2 (*Kruskal*, $2 + 6$ *Punkte*)

Vervollständigen Sie die Methode `kruskal` in der Datei `msf.hpp`. Die Methode erhält als Template-Parameter die UnionFind Implementierung, die genutzt werden soll. Gehen Sie dabei davon aus, dass der ungerichtete Eingabegraph $G = (V, E)$ durch eine Kantenliste repräsentiert ist, in der jede Kante $\{u, v\} \in E$ entweder durch (u, v) oder (v, u) enthalten ist.

1. Beginnen Sie damit, die Hilfsfunktion `Node max_node(std::vector<Edge> &edges)` zu vervollständigen. Sie soll unter allen Kanten den Knoten mit größtem Index zurück geben. Erstellen Sie entsprechende Testfälle, welche die Korrektheit Ihrer Implementierung prüfen. Kanten sind dabei wie folgt definiert:

   ```
   using Node = uint32_t;
   using Weight = float;

   struct Edge {
      Node from;
      Node to;
      Weight weight;
   };
   ```

2. Vervollständigen Sie nun die Implementierung von Kruskal. Die Funktionssignatur lautet wie folgt:

   ```
   struct KruskalResult {
      std::vector<Edge> msf_edges;

      Weight total_weight;
      bool is_spanning_tree;

      uint64_t parent_accesses;
   };

   template <typename UnionFind>
   KruskalResult kruskal(std::vector<Edge> &edges);
   ```

   Ihre Implementierung soll also sowohl die Kanten `msf_edges` des MSF berechnen, als auch dessen Gesamtgewicht `total_weight` zurückgeben. Zudem soll `is_spanning_tree` genau dann `true` sein, wenn der MSF ein MST ist (und dies soll genau dann gelten, wenn die Eingabe zusammenhängend ist). Letztlich soll `parent_access` die Anzahl der Zugriffe auf das Elternarray über die gesamte Laufzeit des Algorithmus enthalten; diese können Sie am Ende des Algorithmus mittels `UnionFind::number_of_parent_accesses()` abfragen (der entsprechende Code ist schon in der Vorlage vorhanden).

   Erstellen Sie Testfälle, bei denen (i) ein MST und (ii) ein nicht-zusämmenhängender MSF berechnet werden. Prüfen Sie das Gesamtgewicht und das Flag `is_spanning_tree`.

Info: Natürlich haben wir in unseren Experimenten das Problem, dass wir erst einmal einen Graph benötigen, auf dem wir einen Spannbaum berechnen können. Die Vorlage stellt hierfür die Funktion `generate_gilbert_graph` (in `graph.hpp`) zur Verfügung und nutzt diese in `msf.cpp`. Sie erzeugt eine zufälligen Graph $G(n, p)$ auf n Knoten, in dem jede Kante unabhängig mit Wahrscheinlichkeit p existiert. Da alle Varianten jeweils identische Graphen verwenden sollen, müssen Sie selbst *nicht* mit dieser Funktion arbeiten, sondern können allen Kruskal-Varianten jeweils dieselbe Kantenliste zur Verfügung stellen.

Aufgabe 3 (*Vermessung,* $4 + 2$ *Punke*)

1. In der Datei `msf.cpp` ist bereits eine grobe Struktur für die Evaluieren der naiven UnionFind (ohne Rank/Pathcompression) vorgegeben. Erweitern Sie das Programm, dass alle vier UnionFind Datenstrukturen vermessen werden. Dabei soll die Strukture der CSV Ausgabe erhalten bleiben, d.h.

   ```
   n,m,avg_deg,unionfind,accesses
   32,92,5,naive,640
   32,92,5,pc,597
   32,92,5,rank,429
   32,92,5,pc+rank,533
   ```

   Beachten Sie, dass die naive Implementierung (ohne Pathcompression/Rank) schlechter skaliert; daher enthält die Vorlage die Bedingung `if (n < max_n / 100)` für `naive`. Diese soll für die anderen Varianten *keine* Anwendung finden.

   Das Programm sollte eine Laufzeit von etwa 50s haben, wenn es mit Optimierungen kompiliert wurde (`build.sh` macht dies bereits; für `cmake` setzen Sie bitte `-DCMAKE_BUILD_TYPE=Release`).

   Nachdem das Programm durchgelaufen ist, finden Sie in der Datei `kruskal.csv` die Ergebnisse; geben Sie diese Datei mit ab.

2. Berechnen Sie für die größte Instanz, die von allen Varianten berechnet wurde, pro Variante die durchschnittlichen Elternarray-Zugriffe pro Kante: Beachten Sie, dass jedes Experiment fünf mal wiederholt wird — geben Sie pro Variante jeweils den Median und die Standardabweichung der Läufen an. Diese Berechnung kann mit einem Werkzeug Ihrer Wahl geschehen (auch manuell).

Aufgabe 4 (*Visualisierung, 5 Bonuspunkte*)

Diese Aufgabe ist eine Bonusaufgabe, die nicht bearbeitet werden muss; hier erhaltene Punkte können genutzt werden, um einen etwaigen Punktverlust bei früheren Aufgaben dieses Blattes abzufedern.

Visualisieren Sie Ihre Messungen aus `kruskal.csv` mit einem Werkzeug Ihrer Wahl. Sie können z.B. das Skript des letzten Blattes anpassen. Die Abbildung sollte folgenden Anforderungen genügen:

- Die X-Achse soll die Knotenanzahl n auf einer logaritmischen Skala zeigen.
- Die Y-Achse soll die durchschnittliche Zahl der Elternarray-Zugriffe pro Kante zeigen.
- Jede Variante soll im selben Plot mit einer eigenen Linie (z.B. mittels Farbe und/oder Stil unterschieden werden).
- Die Streuung der Daten soll erkennbar sein; hierfür können Sie z.B. Fehlerbalken oder Konfidenzintervalle (= Standardeinstellung des Pythonpakets `seaborn`, das auf dem letzten Blatt genutzt wurde) angeben.
- Es soll eine Legende vorhanden sein, die es ermöglicht, die Linien den UnionFind Varianten zuzuweisen.

In der Vorlesung schätzten wir die (amortisierte) Laufzeit von `PCOnly` und `RankOnly` durch $\mathrm{O}(\log n)$ ab. Argumentieren Sie kurz, weshalb die Zugriffe auf das Elternarray ein guter Proxywert sind (analog wie wir für Sortieralgorithmen nur Vergleiche zählten ...). Erzeugen Sie einen zweiten Plot, der nur die beiden Varianten enthält und die Y-Achse durch $m \log(n)$ teilt. Was beobachten Sie?