

Anmerkungen zu Aufgabenblatt 4

Der zu verwendende Programmrahmen steht als Zip-Archiv `uebung4.zip` im Moodle unter der Adresse <https://moodle.hpi.de/course/view.php?id=297> zum Download zur Verfügung.

Aufgabe 4.1: Game of Life (10 Punkte ¹⁺³⁺⁵⁺¹)

In dieser Aufgabe soll eine einfache Variante des Game of Life¹ implementiert werden. Zugrunde liegt dieser Simulation ein dynamisches Array konstanter Größe, das als zweidimensionales Raster verstanden wird und in dem Zellen liegen. Eine Zelle des Arrays ist entweder lebendig oder tot. Beim Übergang von einem Zeitpunkt zum nächsten verändert sich die Population (d. h. der Zustand der einzelnen Zellen), wobei die 8-Zellen-Nachbarschaft der jeweils betrachteten Zelle ausgewertet wird. Die Regeln sind wie folgt definiert:

- Leben entsteht in einer toten Zelle, wenn diese genau drei lebendige Nachbarzellen hat.
- Eine Zelle mit keiner oder nur einer lebendigen Nachbarzelle stirbt aufgrund von Einsamkeit.
- Eine Zelle mit vier oder mehr lebendigen Nachbarzellen stirbt aufgrund einer zu hohen Population in ihrem Umfeld.
- Eine Zelle mit zwei oder drei lebendigen Nachbarzellen überlebt, weil ihr Umfeld ausgewogen ist.

Im Programmrahmen übernimmt das Modul `game_of_life.cpp` die Auswertung der übergebenen Kommandozeilenargumente und initiiert die Simulation.

- a) 2D-Raster-Zugriff: Implementieren Sie die Methoden `value()` und `setValue()` der Klasse **Raster**, die die Übersetzung von einer 2D-Datenstruktur auf die interne 1D-Datenstruktur kapseln.
- b) Import und Export: Implementieren Sie in der Klasse **Raster**:
 - Die Methode `loadRasterFromFile()`, die das Raster auf Grundlage einer angegebenen Beispielgrafik (siehe hierzu die Bitmaps im Data-Ordner des Programmrahmens) befüllt. Die Beispielgrafik kann über eine Kommandozeilenoption spezifiziert werden (z. B. `game_of_life -p ./example.bmp`).
 - Die Methode `fill()`, die das Raster auf Grundlage einer Zufallsfunktion befüllen. Die Größe des Arrays sowie die Wahrscheinlichkeit, mit der eine Zelle als lebendig initialisiert wird, können über die Kommandozeile spezifiziert werden (z. B. `game_of_life -w 600 -h 400 -s 0.1`). Ein Wert von 0.4 führt beispielsweise dazu, dass 40% der Zellen als lebendig befüllt werden. Nutzen Sie für den Zufallszahlengenerator die **Mersenne Twister Engine**.
 - Die Methode `save()`, die den Export der einzelnen Simulationsschritte als Bitmaps ermöglicht. Der finale Dateiname wird dieser Methode bereits übergeben. Das Ausgabeverzeichnis kann über die Kommandozeile spezifiziert werden (z. B. `game_of_life -p ./example.bmp -o output_directory/`).
- c) Simulation: Implementieren Sie die Funktion `simulateNextState()`, um die Population zum nächsten Zeitpunkt der Simulation zu bestimmen. Ermitteln Sie hierzu die Anzahl an lebendigen Zellen in einer 3x3-Nachbarschaft um die jeweilige Zelle. Wenden Sie anschließend die oben genannten Regeln an, um den Folgezustand der Zelle zu bestimmen. Implementieren Sie die Funktion `cellValue()`, um den Wert einer angegebenen Zelle zu bestimmen. Fangen Sie ungültige Anfragen (z. B. Position außerhalb des Rasters) ab, indem Sie
 - i) die Zelle als tot angeben oder
 - ii) das Array als Torus-förmig ansehen (d. h. den Zustand der am gegenüberliegenden Rand liegenden Zelle zurückgeben).

¹https://de.wikipedia.org/wiki/Conways_Spiel_des_Lebens

Für die Simulation wird standardmäßig die erste Variante gewählt. Die zweite Variante kann jedoch über eine entsprechende Kommandozeilenoption (`-t 1`) gewählt werden. Die Anzahl an berechneten Simulationsschritten kann ebenfalls über die Kommandozeile definiert werden (z. B. `-i 20`).

- d) Inversionssimulation: Implementieren Sie als Erweiterung die Funktion `simulateInversion()`, die zu Beginn jedes Simulationsschritts den Zustand jeder Zelle mit einer angegebenen Wahrscheinlichkeit invertiert. Die Wahrscheinlichkeit, mit der eine Zelle invertiert wird, kann als Kommandozeilenargument spezifiziert werden (z. B. `-iv 0.005` für eine Wahrscheinlichkeit von 0.5 %).

Aufgabe 4.2: Iteratoren (5 Punkte ²⁺¹⁺²)

Iteratoren ermöglichen den Zugriff auf Inhalte von Datencontainern mit einer abstrakten Traversierungsfunktionalität. Dadurch können Zugriffs-, Such- und Auswertungsfunktionen implementiert werden, die vom konkret verwendeten Containertyp unabhängig sind. Implementieren Sie die folgenden Funktionen in `iterators.cpp`.

a) Umordnung von Containerelementen

Gegeben sei eine Folge $Z = \{c_0, c_1, \dots, c_{N-1}\}$ von Elementen beliebigen Typs. Erzeugen Sie eine neue Folge Z' , bei der die Elemente abwechselnd vom Anfang bzw. vom Ende von Z stammen, d. h. $Z' = \{c_0, c_{N-1}, c_1, c_{N-2}, \dots\}$. Entwickeln Sie die Funktion `front_back_pairing` generisch, so dass diese mindestens für die Containertypen `std::vector` und `std::list` funktioniert, sowie hinsichtlich des Datentyps unabhängig ist.

b) Entfernung von Duplikaten

Gegeben sei eine Folge $Z = \{c_0, c_1, \dots, c_{N-1}\}$ von Elementen beliebigen Typs, die in einem Container enthalten sind. Zu entfernen sind alle Duplikate, d. h. von mehreren Elementen mit gleichem Wert soll nur das erste Element erhalten bleiben. Beispiel: Falls $c_5 = c_7 = c_{39}$, dann werden c_7 und c_{39} aus dem Container entfernt. Entwickeln Sie eine generische Funktion `remove_duplicates`, die Duplikate entfernt. Verwenden Sie dafür temporär einen geeigneten Datencontainer, der beim Einfügen duplizierte Einträge verwirft. Die Reihenfolge der Elemente muss bei der Entfernung der Duplikate nicht erhalten werden.

c) Erweitern einer Zahlenfolge

Gegeben sei eine Zahlenfolge $Z = \{c_0, c_1, \dots, c_{N-1}\}$. Erzeugen Sie eine neue Folge Z' , bei der zu jedem Ursprungselement jeweils links und rechts ein neues Element hinzukommt; das jeweils neue linke bzw. rechte Element enthält die numerische Differenz zum Ausgangselement, d. h. das Element c_i in der Ursprungsfolge wird ersetzt durch die Teilfolge $c_{i-1} - c_i, c_i, c_{i+1} - c_i$. Die resultierende Folge Z' ist dreimal so lang wie Z . Die Folge wird zirkulär betrachtet, d. h. $c_{-1} = c_{N-1}$ und $c_N = c_0$. Implementieren Sie eine entsprechende generische Funktion `insert_differences`, die den Inhalt des übergebenen Containers C (enthält die Folge Z) durch die neue erweiterte Folge Z' ersetzt. Standardalgorithmen dürfen verwendet werden.

Aufgabe 4.3: Suchen in Datensätzen (5 Punkte ²⁺³)

Gegeben sind Flugverbindungsdaten, die im Programmrahmen `search.cpp` nach dem Einlesen einer Datei (`routes.csv`) im Speicher durch eine Struktur `Route` mit den Attributen `SourceID`, `DestinationID` und `AirlineID` dargestellt werden. Eine Flugverbindung wird als Eintrag in einem Vektor vom Typ `std::vector<Route>` abgelegt. Die gegebene Einleseroutine fügt die Daten direkt in den Vektor ein und gruppiert sie nach `AirlineID`.

a) Lineare Suche der Verbindungen zu einem Zielflughafen

Implementieren Sie eine Funktion, die zu einer gegebenen `DestinationID`, d. h. einem Zielflughafen, ermittelt, wie viele Verbindungen zu diesem Zielflughafen insgesamt existieren. Verwenden Sie dazu in dieser Teilaufgabe zunächst das Prinzip der linearen Suche.

Zum Test führen Sie Abfragen für alle möglichen `DestinationsIDs` aus und messen Sie die Gesamtlaufzeit dieser Suchanfragen. Ermitteln Sie als statistische Information die Anzahl der Zugriffe auf die Inhalte des Datenvektors.

b) **Binäre Suche der Verbindungen zu einem Zielflughafen**

Implementieren Sie die Suchfunktion zur Ermittlung aller Verbindungen zu einem Zielflughafen analog zu a) mit Hilfe eines sortierten Vektors. Überlegen Sie sich eine geeignete Sortierung der Elemente, um die Sortierreihenfolge für die Suchanfrage ausnutzen zu können. Für das Sortieren steht die Standardfunktion `std::sort` bereit.

Führen Sie wie bei Teilaufgabe a) zum Test Abfragen für alle möglichen `DestinationsIDs` aus und messen Sie die Gesamtlaufzeit dieser Suchanfragen (ohne die Sortierung). Ermitteln Sie als statistische Information wieder die Anzahl der Zugriffe auf die Inhalte des Datenvektors.

Hinweis: Nutzen Sie zur Laufzeitmessung geeignete Funktionen aus der `std::chrono`-Bibliothek.

Allgemeine Hinweise zur Bearbeitung und Abgabe

- Die Aufgaben können allein oder zu zweit bearbeitet werden. Je Gruppe ist nur eine Abgabe notwendig.
- Bonuspunkte können innerhalb des Übungsblattes (nicht nur innerhalb der spezifischen Aufgabe) verlorene Punkte ausgleichen. Sie sind nicht auf andere Übungsblätter übertragbar und werden nicht über die reguläre Punktzahl hinaus angerechnet.
- Bitte reichen Sie Ihre Lösungen bis spätestens **Dienstag, den 21. Juni um 17:00 Uhr** ein.
- Die Implementierung kann auf einer üblichen Plattform (Windows, Linux, OS X) erfolgen, darf aber keine plattformspezifischen Elemente enthalten, d. h. die Implementierung soll plattformunabhängig entwickelt werden.
- Die Lösungen von theoretischen Aufgaben können als Textdatei oder PDF abgegeben werden. Benennen Sie die Dateien im Format `<Aufgabenblatt>_<Aufgabe>_<Teilaufgabe>`, bspw. „4_4_f.pdf“ oder „4_4_g.pdf“.
- Bestehen weitere Fragen und Probleme, kontaktieren Sie den Übungsleiter oder nutzen Sie das Forum im Moodle.
- Archivieren Sie zur Abgabe Ihren bearbeiteten Programmrahmen und die Lösungen der theoretischen Aufgaben als Zip-Archiv und ergänzen Sie Ihre Namen im Bezeichner des Zip-Archivs im folgenden Format: **uebung4_Grp_MatNr1_MatNr2.zip**, bspw. **uebung4_grp1_123456_654321.zip**. Beachten Sie, dass dabei nur die vollständigen Lösungen sowie eventuelle Zusatzdaten gepackt werden (alle Dateien, die im gegebenen Programmrahmen vorhanden waren); vermeiden Sie bitte nicht notwendige Unterordner. Laden Sie keine Kompilate und temporären Dateien (*.obj, *.pdb, *.ilk, *.ncb, *.exe, etc.) hoch. Testen Sie vor dem Hochladen, ob die Abgabe fehlerfrei kompiliert und ausgeführt werden kann.
- Reichen Sie Ihr Zip-Archiv im Moodle ein:
<https://moodle.hpi.de/course/view.php?id=297>.