IMS
Institut für Mikroelektronische Systeme
Leibniz Universität Hannover

Programmierpraktikum Technische Informatik (C++)
Aufgabe 06

Hinweise
Abgabe: Stand des Git-Repositories am 5.6.2018 um 9 Uhr.
Die Dateien zur Bearbeitung dieser Aufgabe erhalten Sie, indem Sie die neue Aufgabe aus dem Aufgabenrepository in Ihr lokales mergen. Dies geschieht mit `git pull cpp2018 master` innerhalb Ihres Repositories. Die Lösungen committen Sie bitte in Ihr lokales Repository (`git commit -a` oder `git add` gefolgt von `git commit`) und pushen sie in Ihr Repository auf dem git-Server des Instituts (`git push`).

Teilaufgabe 1 (3 Punkte)

Im Verzeichnis `Labyrinth` soll ein Programm vervollständigt werden, das in einem Labyrinth einen kürzesten Weg von einem Startpunkt zu einem Zielpunkt findet.

Ein Labyrinth liegt in Form einer speziell formatierten Textdatei vor. Diese ist unterteilt in Zeilen, die direkt den Zeilen des Labyrinthgitters entsprechen. Die einzelnen Zeilen bestehen aus einer Abfolge von Symbolen, wobei jedes Symbol den Status der durch die entsprechende Zeile und Spalte identifizierten Kachel angibt. Mögliche Symbole sind:

- '`-`' für leere Kacheln (`Empty`)
- '`x`' für durch Wände blockierte Kacheln (`Barrier`)
- '`O`' für den Startpunkt (`Origin`)
- '`D`' für den Zielpunkt (`Destination`)

In der Ausgabe sind zusätzlich zwei weitere Symbol möglich:

- '`p`' für Kacheln, die auf dem gefundenen Pfad liegen (`Path`)
- '`v`' für besuchte Kacheln (`visited`, sofern die entsprechende Option bei der Ausgabe verwendet wird

Beachten Sie, dass Start- und Zielpunkt zwar auf dem Pfad liegen und besucht sein können, diese aber trotzdem mit '`O`' bzw. '`D`' statt mit '`p`' bzw. '`v`' ausgegeben werden.

Appelstraße 4 | 30167 Hannover
fon 0511 762-19691 | fax 0511 762-19694
www.ims.uni-hannover.de | info@ims.uni-hannover.de

Die Funktionalität zum Einlesen der Daten ist bereits in der Bibliothek `liblabyrinth.a` implementiert, die vom Makefile automatisch mit eingebunden wird.

Für die Suche nach dem kürzesten Weg wird eine Breitensuche verwendet. Diese Suche geht vom Startpunkt aus und fügt bei der Verarbeitung einer Kachel alle seine bisher noch nicht besuchten Nachbarn einer Queue hinzu. Zur späteren Nachverfolgbarkeit des gefundenen Pfades wird weiterhin die aktuell bearbeitete Kachel in einer Datenstruktur als Vorgänger für die hinzugefügten Nachbarn abgelegt. Danach wird das erste Element aus der Queue genommen und das Vorgehen wiederholt. Der Algorithmus bricht ab, wenn die Queue leer ist, also alle erreichbare Kacheln besucht wurden. Vom Endpunkt ausgehend kann nun mithilfe der Vorgängerinformationen der kürzeste Weg rekonstruiert werden.

a) Implementieren Sie in `Labyrinth/src/labyrinth.cpp` die Methode `getOrigin`! Sie soll einen Pointer auf die Ursprungskachel (`Origin`) des Labyrinths zurückgeben.

b) Implementieren Sie in `Labyrinth/src/labyrinth.cpp` die Hilfsmethode `emplaceNeighboor`! Handelt es sich bei der Kachel an der übergebenen Koordinate nicht um eine Barriere (`Barrier`) oder um eine bereits besuchte Kachel (`Visited`), so soll ein Pointer auf diese Kachel in die Queue `pending` eingefügt werden. Außerdem muss in diesem Fall der Wert an der übergebenen Koordinate im Vektor `predecessors` auf die als `current` übergebene Kachel gesetzt werden.

   Hinweise:

   - Sie können davon ausgehen, dass das Labyrinth rechteckig ist, also alle Zeilen die gleiche Länge haben.
   - Der Vektor `predecessors` soll später an jeder Koordinate einen Pointer auf den im Ablauf des Wegfindungsalgorithmus zuerst ermittelten Vorgänger oder einen `nullptr` enthalten.
   - Der Container `std::queue` stellt eine Warteschlange dar, die für Elemente sowohl das Einfügen am Ende als auch das Entfernen am Anfang besonders effizient ermöglicht. Sie werden im Rahmen dieses Aufgabenblatts lediglich die Methoden `emplace`, `front` und `pop` benötigen. Die Dokumentation dieser Klasse können Sie unter `http://en.cppreference.com/w/cpp/container/queue` nachlesen.

c) Vervollständigen Sie in `Labyrinth/src/labyrinth.cpp` die Methode `searchShortestPath`! Diese soll auf Basis der bereits beschriebenen Breitensuche den kürzesten Pfad durch das Labyrinth finden.

   Hinweise:

   - Pfade dürfen in diesem Labyrinth nur rechtwinklig verlaufen, diagonale Pfade, also solche, in denen sich zwei hintereinander überquerte Kacheln nur in einem

Eckpunkt berühren, sind nicht zulässig.

- Für die erfolgreiche Implementierung des Algorithmus ist es zielführend, zunächst den Ursprung des Labyrinths zu finden und dessen Nachbarn zu bearbeiten. Anschließend sollten die Nachbarn der Nachbarn behandelt werden und so weiter.
- Das erste Auffinden des Zielpunktes ist bei diesem Vorgehen automatisch über den kürzest möglichen Weg erreicht.
- Zur Abarbeitung aller Kacheln in der richtigen Reihenfolge könnten sich die Nutzung der Warteschlange `pending` und die Hilfsmethode `emplaceNeighboor` als überaus hilfreich erweisen.
- Die Methode `.visit()` der Klasse `Tile` kann verwendet werden, um Kacheln als bereits besucht zu markieren.
- Mit der Methode `.addToPath()` werden Kacheln als zum Pfad gehörig markiert.

Bonusaufgabe (1 Punkt)

`searchShortestPath` findet garantiert den kürzesten Weg, überprüft dafür allerdings möglicherweise auch sehr viele Wege, die nicht zum Ziel führen. Implementieren Sie eine Methode `fastSearchShortestPath`, die dies effizienter gestaltet! Verwenden Sie dafür den A*-Algorithmus (`http://de.wikipedia.org/wiki/A*-Algorithmus`)! Optional können Sie auch weitere Optimierungen vornehmen, beispielsweise indem Sie die Suche an Start- und Zielpunkt parallel starten lassen oder indem Sie die Verwendung von nichtexakten Heuristiken (vergleichen Sie dazu `http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html`) erlauben. Verwenden Sie den optionalen Parameter der `print`-Methode, um sich anzeigen zu lassen, welche Zellen Sie besuchen!

Teilaufgabe 2 (2 Punkte)

Im Verzeichnis `blackJack` soll ein Programm vervollständigt werden, welches das in Casinos meistgespielte Karten-Glücksspiel Black Jack (Siebzehn und vier) umsetzt. Ein Aufruf des Programms ohne Parameter soll interaktive Partien eines Spielers (Mensch) gegen den Dealer (Computer) unter Verwendung eines unbegrenzten Decks ermöglichen. Wird das Programm mit einer Zahl n als Parameter aufgerufen, so simuliert es n Partien des Spiels an einem vollständig besetzen Spieltisch mit 7 Spielern (Computer) gegen den Dealer (Computer) unter Verwendung eines begrenzten Decks, bestehend aus 6 Kartenpaketen, und gibt die Ergebnisse jeder Runde auf der Konsole aus.

Programmablauf, Spieler und Spielablauf liegen bereits vollständig implementiert vor.

Für die Bearbeitung der Aufgabe ist kein Vorwissen über die Spielregeln vonnöten. Falls weiteres Interesse oder Unklarheiten bestehen, können die Regeln unter `http://de.wikipedia.org/wiki/Black_Jack` nachgelesen werden. Dabei können Sie die Unterabschnitte Siebener-Drilling, Insurance, Split und Double ignorieren, weil diese unübliche Regeln oder Nebenwetten behandeln, die nicht umgesetzt sind.

a) Ein Kartenpaket enthält 52 Karten. Jeweils 4 der Zahlenkarten von 2 bis 10 und jeweils 4 der Bildkarten Bube, Dame, König und Ass. Die Spielkarten werden nicht als Objekte behandelt, sondern durch ihren Wert im Spiel repräsentiert. Zahlenkarten haben den Wert der abgebildeten Zahl, das Ass hat den Wert 11 und die übrigen Bildkarten den Wert 10.
Die von der Klasse `Deck` abgeleitete Klasse `InfiniteDeck` stellt ein Deck dar, bei dem die Wahrscheinlichkeit, eine bestimmte Karte zu ziehen, von den bereits gezogenen Karten unabhängig ist. Bei der Klasse `LimitedDeck` ist sie von vorher gezogenen Karten abhängig.

   Programmieren Sie in `blackJack/src/deck.cpp` den Konstruktor der Klasse `LimitedDeck`! Dieser soll das erstellte Deck mit den Spielkarten für die übergebene Anzahl Kartenpakete befüllen und die Karten **zufällig** mischen.

   Hinweise:

   - Die bereits vorhandene Methode `Deck::fill` könnte sich als hilfreich erweisen.
   - Zum Mischen der Karten ist die Funktion std::shuffle (s. `http://www.cplusplus.com/reference/algorithm/shuffle`) in Verbindung mit der bereits vorhandenen Random Engine `Deck::re` hilfreich.

b) Ergänzen Sie die Methode `getRandomCard` für `InfiniteDeck` und `LimitedDeck`! Diese Methode implementiert das Ziehen einer Karte aus dem Deck. Verwenden Sie für das `InfiniteDeck` zur zufälligen Auswahl einer Karte die bereits in der Klasse vorhandene diskrete Gleichverteilung (s. `http://en.cppreference.com/w/cpp/numeric/random/uniform_int_distribution`) und die Random Engine `re`. Im Falle des `LimitedDeck` soll die hinterste Karte des Vektors `cards` aus diesem entfernt und zurückgegeben werden.

   Hinweis: Für die nächste Teilaufgabe kann es sinnvoll sein, aus dem `LimitedDeck` entfernte Karten in einer für diesen Zweck hinzugefügten Membervariable zu speichern.

c) Implementieren Sie die Methode `LimitedDeck::resetCards`!
Diese repräsentiert die Vorbereitung einer neuen Runde unter Verwendung einer automatischen Mischmaschine, wie sie in den meisten Casinos zum Einsatz kommt. Sie soll die durch `getRandomCard` entfernten Karten wieder in den Vektor `cards` einfügen und anschließend das Deck neu mischen.

d) Warum ist es – abgesehen von einem höheren Grad an Realismus – sinnvoll, den `std::vector` vollständig zu durchmischen und Karten vom Ende zu entnehmen, anstatt das

Mischen zu sparen und Karten an zufälligen Indizes zu entnehmen?
Warum sind `std::random_device rd` und `std::default_random_engine re` static?