IMS
Institut für Mikroelektronische Systeme
Leibniz Universität Hannover

Programmierpraktikum Technische Informatik (C++)
Aufgabe 06

Hinweise
Abgabe: Stand des Git-Repositories am 16.6.2020 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 cpp2020 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 (2 Punkte)

Im Verzeichnis `MontyHall` soll ein Programm vervollständigt werden, das die Spielshow des Ziegenproblems (Monty-Hall-Problem, `https://de.wikipedia.org/wiki/Ziegenproblem`) umsetzt. Wird das Programm ohne Parameter aufgerufen, so soll es interaktiv eine solche Showsituation durchspielen. Wenn das Programm mit einer Zahl als Parameter aufgerufen wird, dann sollen entsprechend viele Showsituationen automatisch durchprobiert werden. Anschließend werden der Anteil der Versuche ausgegeben, in denen die Strategie erfolgreich gewesen war, bei der ersten gewählten Tür zu bleiben und der Anteil der Versuche, bei denen es besser gewesen wäre zu wechseln.

Der Großteil des Programms liegt bereits vor. Eine Instanz der Klasse `Show` stellt ein Ziegenproblem-Experiment dar. Die Türen sind von 1 bis 3 nummeriert. Verwenden Sie zur zufälligen Auswahl einer Tür eine diskrete Gleichverteilung (s. `http://en.cppreference.com/w/cpp/numeric/random/uniform_int_distribution`) und die bereits vorhandene Random Engine `re`.

a) Programmieren Sie in `MontyHall/src/show.cpp` die Methode `Show::showGoatDoor`! Versetzen Sie sich dazu in die Lage des Showmasters, d.h. Sie wissen, wo sich das Auto befindet. Falls der Kandidat bei seiner ersten Wahl eine Tür mit Ziege dahinter gewählt hat, so geben Sie die Nummer der Tür zurück, hinter der die andere Ziege steht. Falls der Kandidat bei seiner ersten Wahl bereits die Tür mit dem Auto gewählt hat, so geben Sie eine **zufällige** der beiden anderen Türen zurück (hinter beiden steht eine Ziege).

b) Vervollständigen Sie in `MontyHall/src/main.cpp` die Funktion

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

`montyHallExperiment`! Sie soll eine jeweils neue Show durchführen und einen Kandidaten spielen, der zunächst eine zufällige Tür wählt und dabei bleibt. Die Funktion soll zurückgeben, ob die Strategie erfolgreich war. Warum sind `std::random_device rd` und `std::default_random_engine re` static?

c) Vervollständigen Sie in `MontyHall/src/main.cpp` die Funktion `montyHallExploration`! Die Funktion soll `n` mal das `montyHallExperiment` durchführen und den Anteil (zwischen 0 und 1) der erfolgreichen Versuche zurückgeben.

Teilaufgabe 2 (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.

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!