W11B01 - Pengu Survivors

Bonusaufgabe Hard

Points: 8 of 8 Bonus Assessment: automatic

Submission due: 2 months ago
Complaint due: 2 months ago

Practice Clone Repository

100% (3 months ago) GRADED

Recent results:

50% (3 months ago)	87.5% (3 months ago)	100% (3 months ago)	100% (3 months ago)	100% (3 months ago)
GRADED	GRADED	GRADED	GRADED	GRADED

Show all results

Tasks:

Pengu Survivors

Die Pingu-Studenten der PUM haben gelernt, dass es hilfreich sein könnte, Teams aus prokrastinierenden und übereifrigen Studenten zu bilden. Um möglichste viele Paare bilden zu können, möchtest du eine Simulation entwickeln, um ein optimales Matching zu finden.

Max Flow

Ein Flow Network ist ein Graph, den du dir wie eine Kanalisation vorstellen kannst. Kanten haben dabei eine Kapazität (`c`), die den maximalen Durchfluss in diesem "Rohr" beschreibt und einen Wert `f`, der beschreibt, wie viel aktuell durch dieses Rohr fließt. Für solch ein Netzwerk können wir eine Quelle (source `s`) und ein Ziel (sink/target `t`) festlegen und bestimmen, wie viele Einheiten an "Flüssigkeit" maximal von `s` nach `t` gepumpt werden kann. Das nennt man das max-flow value. Um diesen Wert zu bestimmen, müssen wir einen maximalen Flow berechnen (durch welches Rohr soll wie viel fließen?). Im Folgenden werden wir einen der vielen Algorithmen dafür Schritt für Schritt selbst implementieren.

Aufgaben

Limitierungen/Einschränkungen

Du kannst davon ausgehen, dass ...

1. ein s-t-Flow mit einem Wert, der größer ist als 0, berechnet werden kann.
2. Quelle und Ziel richtig gesetzt sind (also nicht `null` und Knoten des entsprechenden `FlowGraph`s sind).
3. der `FlowGraph` nur eine Zusammenhangskomponente enthält.
4. der `FlowGraph` keine Knotenpaare mit Hin- und Rückkante enthält (es existiert entweder eine Kante von `a` nach `b` oder von `b` nach `a` oder keine von beiden, jedoch nie beide zugleich).
5. Alle Kapazitäten des Input-Graphen positiv sind (Achtung: der von dir erstellte Residual-Graph wird Kanten mit `c=0` enthalten).

Einführung in das Template

Im Template ist bereits eine Klasse zur Modellierung des Graphen gegeben: `FlowGraph`. Dieser speichert seine Knoten (`Vertex`) in einem `Set`. Jeder `Vertex` speichert das eigentliche Netzwerk (Modell) in einer Map `neighbours`, deren Keys die benachbarten Knoten sind und die zugehörigen Values `Edge`-Objekte, die das Speichern von `f` (Flow) und `c` (Kapazität) erlauben. Für unsere Berechnung benötigen wir einen Residual Graphen. Dieser wird in der Map `residual` des Knoten gespeichert. Da im Residual Netzwerk lediglich die Kapazitäten der Kanten relevant sind, ist bei diesen Kanten der Eintrag für `f` immer `0`.

Residual Graph

Ein Residual Graph hilft uns dabei, den maximalen Fluss eines Netzwerkes zu bestimmen. Auch wenn man diesen eigentlich aus dem ursprünglichen Netzwerk herauslesen kann, ist es dennoch einfacher, diesen separat zu modellieren. Daher entscheiden wir uns für diesen etwas intuitiveren Ansatz. Die Idee des Residual Graphen ist es, eine Repräsentation von allen möglichen Flüssen zu haben, die auf den aktuellen Fluss "addiert" werden könnten. Nehmen wir an es existiert eine Kante von u nach v mit Kapazität $c(u, v)$, durch die aktuell $f(u, v)$ Einheiten fließen. Ist die Kante noch nicht vollständig saturiert ($f(u, v) < c(u, v)$), könnten wir $f(u, v)$ noch um die Differenz erhöhen. Der Residual Graph würde daher eine Kante von (u, v) mit Residualkapazität $c_f(u, v) = c(u, v) - f(u, v)$ enthalten. Andererseits könnte es auch schlau sein, den Fluss entlang einer Kante zu verringern. Daher enthält der Residual Graph zu jeder Kante auch eine Rückkante. Bei unserem Beispiel von zuvor hätte diese Kante nun die Residualkapazität $c_f(v, u) = f(u, v)$, da der Fluss maximal um $f(u, v)$ verringert werden könnte. Für ein Beispiel einer einzelnen Kante kannst du dir gerne folgendes Beispiel ansehen (Am Ende der Angabe findest du noch einmal ein Beispiel, dass den gesamten Algorithmus inklusive aller Zwischenschritte darstellt und erläutert):

1. **generateResidualGraph** 1 of 1 tests passing
   Implementiere die Methode `FlowGraph.generateResidualGraph()`. Die Methode soll den Residual Graphen initialisieren. Halte dich dabei an die Beschreibung von oben. Zusätlich kann das Beispiel (unterhalb der Angabe) beim Verständnis helfen.

Augmentierende Pfade

2. **findPathInResidual** 1 of 1 tests passing
   Implementiere die Methode `FlowGraph.findPathInResidual()`. Die Methode soll einen Pfad von `s` nach `t` im Residual Graph finden. Achte dabei darauf, dass der Pfad keine Kreise enthalten darf und nur über Kanten mit positiver Kapazität laufen soll. Existiert solch ein Pfad, soll dieser als `List<Vertex>` zurückgegeben werden. Der erste Knoten soll dabei `s` und der letzte Knoten `t` sein. Lässt sich kein solcher Pfad finden, wird der Rückgabewert `null` erwartet. (*Hinweis*: Existieren mehrere mögliche Pfade, ist es irrelevant, welcher Pfad von der Methode gefunden wird.)

Nun sind wir dazu in der Lage, einen Pfad zu finden, der den aktuellen Fluss augmentieren (vergrößern) kann. Jetzt benötigen wir eine Methode, um herauszufinden, um wie viel unser Fluss vergrößert wird.

3. **calcAugmentingFlow** 1 of 1 tests passing
   Implementiere die Methode `FlowGraph.calcAugmentingFlow(List<Vertex> path)`. Die Methode soll berechnen, wie viele Einheiten zusätzlich entlang des Pfades fließen können. Traversiere dazu den Pfad im Residual Graph. Du kannst davon ausgehen, dass der Methode nur Pfade, die nicht `null` sind und echte s-t-Pfade übergeben werden. (In anderen Worten: Du musst dir keine Gedanken über Sonderfälle machen.)

Augmentieren eines Flows

Wir sind nun dazu in der Lage, einen augementierenden Pfad und dessen Wert zu bestimmen. Nun möchten wir diesen in unser bestehendes Modell übernehmen, also den aktuellen Fluss um den augmentierenden Pfad erweitern.

4. **updateNetwork** 1 of 1 tests passing
   Implementiere die Methode `FlowGraph.updateNetwork(List<Vertex> path, int f)`. Die Methode bekommt einen augmentierenden Pfad und den entsprechenden Flusswert übergeben. Mit diesen Informationen soll das aktuelle Modell geupdatet werden (sowohl die Kanten des ursprünglichen Modells als auch des Residual Graphs). Traversiere dazu den gegebenen Pfad und passe `f` (ursprüngliches Modell) und `c` (Residual Graph) der Kanten entsprechend an. Achte dabei darauf, dass ein augmentierender Pfad auch gegen die Flussrichtung einer Kante entlang laufen kann. Stell dir vor der augmentierende Pfad schreibt einen augmentierenden Fluss von `from` nach `to` vor, im Graph des Modells gibt es jedoch nur eine Kante von `to` nach `from`. In diesem Fall möchte der augmentierende Pfad den Fluss von `to` nach `from` reduzieren.

Berechnen eines Max-Flows und seinem Wert

Mit unserem aktuellen Programm können wir augmentierende Pfade suchen und unseren aktuellen Durchfluss dadurch Schritt für Schritt ein wenig verbessern. Sieh dir die Methode `FlowGraph.computeMaxFlow()` an. Diese sollte einfach zu verstehen sein. Solange wir einen Pfad finden, um den aktuellen Durchfluss zu verbessern, updaten wir unseren Flow bis kein solcher Pfad mehr gefunden werden kann. Dann wissen wir: Wir haben eine optimale Lösung für das Max-Flow-Problem gefunden. Jetzt möchten wir noch wissen, wie groß der Wert des Flows ist.

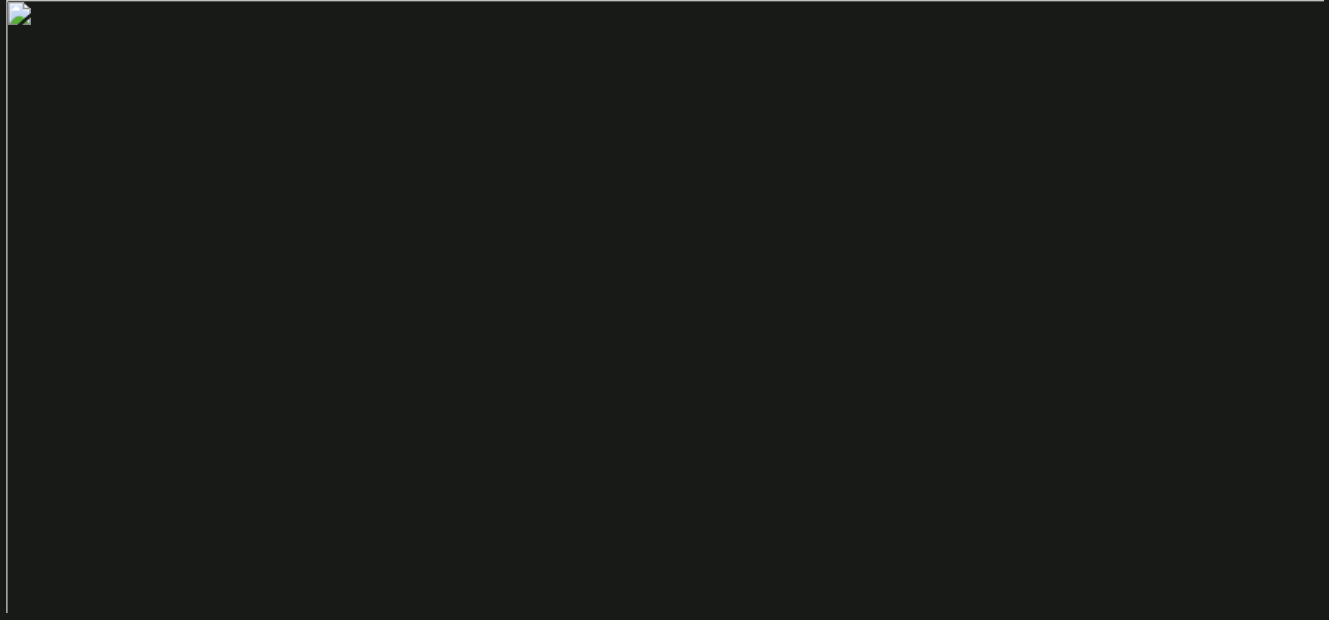
5. **computeMaxFlowValue** 1 of 1 tests passing
   Implementiere die Methode `FlowGraph.computeMaxFlowValue()`, die den Wert des Flows berechnen soll. Benutze dazu zunächst die Methode `computeMaxFlow()`, um einen maximalen Durchfluss des Netzwerks zu bestimmen (korrekte Zuweisungen der Attribute `Edge.f`). Du kannst davon ausgehen, dass der Algorithmus korrekt arbeitet (unabhängig von deiner Implementierung). Jetzt musst du nur noch bestimmen, wie groß der Wert des maximalen Durchflusses ist. *Tipp*: Dazu bieten sich vor allem die (ein/ausgehenden) Kanten der Knoten `s` und `t` an.

Bipartites Matching

6. **generateModel** 1 of 1 tests passing

Implementiere die Methode `PenguSurvivors.generateModel(int[], int[], int[][])`. Die Methode bekommt die beiden Arrays `workaholics` und `procrastinators` gegeben. Dabei handelt es sich um die Pinguine, die gematcht werden sollen. An Stelle `i` steht dabei der Workaholic/Prokrastinator mit Id `i` und symbolischen Namen `workaholics[i]` bzw. `procrastinators[i]`. (Die Namen sind für die Lösung irrelevant, können das Debuggen aber vereinfachen.) Das Array `friendships` enthält Arrays der Länge `2` wobei der erste Eintrag der Id eines `workaholics` entspricht und der zweite Eintrag der Id eines `procrastinators`. Zwei Pinguine dürfen nur gematcht werden, wenn die beiden Pinguine befreundet sind, also ein entsprechender Eintrag in `friendships` enthalten ist. Modelliere mit diesen Informationen ein Netzwerk (`FlowGraph`), dass das bipartite Matching von `workaholics` und `procrastinators` simuliert und gib dieses zurück. (Vergiss nicht, Quelle und Ziel zu definieren.) Du darfst dazu so viele Kanten und Knoten erstellen, wie du für notwendig hältst, versuche aber den Graph relativ klein zu halten, um den Test nicht unnötig lange rechnen zu lassen.

Beispiel



Every student is allowed to complain once per exercise. In total 1000 complaints are possible in this course. You still have 998 complaints left.

Complain