Corona Schools Matching-Algorithmus

November 11, 2020

Übersicht

1. Einleitung
2. Graph Konstruktion
3. Erstellung der Kostenfunktion
4. Berechnung des Matchings
5. Output
6. Mögliche Erweiterungen

Einleitung

Problemstellung

- **Gegeben**: Eine Menge A von Schülern und eine Menge B von Studenten mit verschiedenen Eingabedaten.
- Aus den Daten ergibt sich eine Menge möglicher Paarungen $E \subseteq A \times B$.
- **Gesucht**: Ein Matching $M \subseteq E$, dass möglichst viele Schüler/Studenten abdeckt, möglichst viele Fächerwünsche erfüllt, Wartezeiten respektiert usw.

Leitfragen

- Wie lässt sich unser Problem präzise (mathemtisch) formulieren?
- Wie können verschiedene Kriterien einbezogen werden?
- Wie lässt sich das entstehende Matching Problem algorithmisch lösen?

Mathematischer Hintergrund

Das Matching Problem ist ein bekanntes Problem der **Kombinatorischen Optimierung** und lässt sich auf einem **bipartiten Graphen** modellieren.

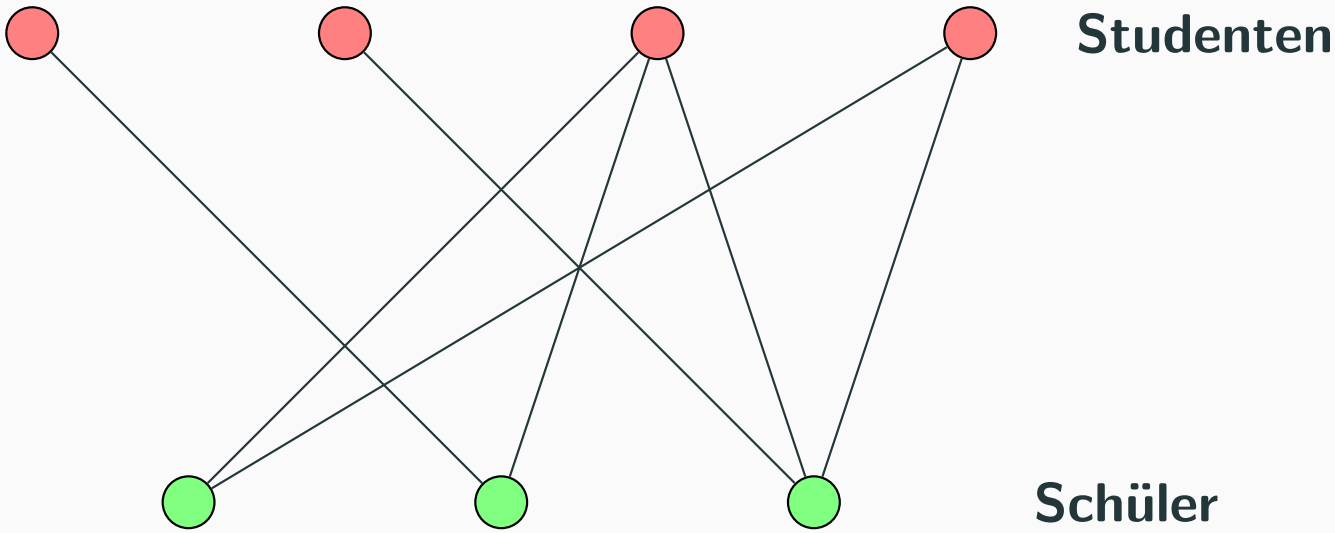
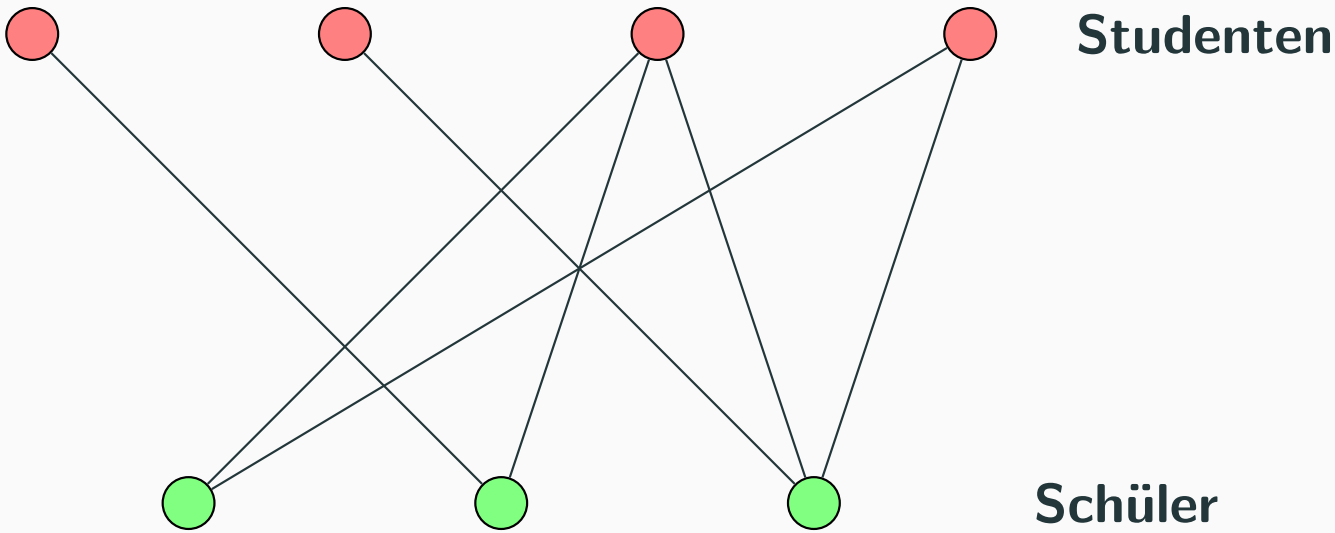


Figure 1: Kanten entsprechen möglichen Paarungen.

Mathematischer Hintergrund

Um verschiedene Kriterien zu berücksichtigen, erstellen wir eine **Kostenfunktion** $c : E \to \mathbb{R}_{\geq 0}$. Diese hängt von den Eingabedaten der Studenten und Schüler ab.

- Wir suchen ein Matching $M \subseteq E$, welches die Kostenfunktion $\sum_{e \in M} c(e)$ **maximiert**.
- Bekannt: Dieses Problem kann für jede Kostenfunktion c effizient gelöst werden.



Graph Konstruktion

Eingabeformat

Der Algorithmus erwartet drei **JSON-Eingabedateien**. (Beispiele einfügen)

1) Eine Datei mit relevanten Daten über die **Schüler**
2) Eine Datei mit relevanten Daten über die **Studenten**
3) Eine Datei mit **Gewichtungskoeffizienten** (dazu später mehr!)

Achtung! Momentan erwartet der Algorithmus alle Felder, die in den Beispieldateien existieren, mit ihren Bezeichnung in den Eingabedateien.
Abweichungen könnten aber leicht im Code verändert werden.

Datenstrukturen

- Die zentrale Klasse ist der *GraphCreator*, er kennt Kanten und Knoten, sowie die Kostenfunktion.
- Kanten haben Kosten und kennen die Id's ihrer Endpunkte.
- Die Knoten sind in einer Klasse *NodeContainer* gespeichert, dort werden Studenten und Schüler separat gehalten.
- Schüler und Studenten haben diverse Daten, die als *StudentData* oder *PupilData* verfügbar sind. Diese Daten sind Structs die von einem gemeinsamen Struct *Data* erben.

Kantenerzeugung

Jede Kante im Graphen stellt eine mögliche Paarung dar. Im Code werden diese (im Prinzip) folgendermaßen erstellt:

```
1: for student in nodes.students() do
2:     for pupil in nodes.pupils() do
3:         if student.accepts(pupil) and pupil.accepts(student) then
4:             create_edge(student,pupil)
5:         end if
6:     end for
7: end for
```

- Eigentliche Logik in accepts()-Funktion implementiert → Durch Modularisierung können Regeln schnell angepasst werden.
- Momentan wird in accepts()-Funktion überprüft ob es ein übereinstimmendes Fach gibt, bei dem der Schüler in der vom Studenten angebotenen Klassenstufe liegt.

Erstellung der Kostenfunktion

CostComponents

Um verschiedene Kriterien zu berücksichtigen, erstellen wir eine **Kostenfunktion**. Diese setzt sich als **gewichtete Summe** verschiedener *CostComponent*s zusammen

→ Im Code besteht ein *CostComponent* aus einem Gewichtungskoeffizienten (double) und einem *RawCostComponent* std::function<double>(Student,Pupil)

→ **Beispiel**: Um die Fächerübereinstimmung einzubeziehen, wählen wir (möglicherweise erst später) einen Gewichtungskoeffizienten > 0 und als RawCostComponent eine Funktion, die die Anzahl der gleichen Fächer für einen Schüler und einen Studenten zurückgibt.

Wir haben momentan Kosten für **Fächerübereinstimmungen, Bundeslandübereinstimmungen** und **Wartezeiten**.

CostComponents

Die Berechnung der Kantenkosten (im Prinzip) folgendermaßen implementiert:

```
1: for edge in edges() do
2:     edge.cost = 0
3:     for [coefficient, rawcostfunc] in cost_components() do
4:         edge.cost += coefficient *
                        rawcostfunc(edge.student(), edge.pupil())
5:     end for
6: end for
```

Vorteil: Durch Modularisierung können neue Kriterien als CostComponents leicht separat hinzugefügt werden.
Frage: Wie sollen die Koeffizienten gewählt werden?

Gewichtungskoeffizienten

In einem dritten input file können Gewichtungskoeffizienten bestimmt werden, z.B. https://github.com/corona-school/matching_new/blob/master/examples/balancing_coefficients.json

- Mit den Koeffizienten kann gesteuert werden, welches Kriterium wie wichtig ist.
- Die Summe der Gewichtungskoeffizienten sollte 1 ergeben.
- Momentan gibt es die drei Kriterien Fachübereinstimmung, Wartezeitbonus und Bundeslandbonus.
- Falls z.B. Fachübereinstimmung auf 0.8 gesetzt wird, gewichtet der Algorithmus die Kosten neu, sodass 80 % der Kosten Fächerübereinstimmungen entsprechen

→ Falls neue CostComponents hinzukommen, können neue Gewichtungskoeffizienten ohne viel Aufwand hinzugefügt werden. Anpassungen sind nur beim Einlesen der Daten nötig!

Berechnung des Matchings

Matching als Flussalgorithmus

Das Matching Problem ist äquivalent zum minimum cost maximum flow Problem auf einem modifizierten Graphen:

1) Füge zwei neue Knoten s und t hinzu.
2) Verbinde s mit allen Schülern und alle Studenten mit t.
3) Setze $k = \min\{|Schueler|, |Studenten|\}$, das wird der Wert des Flusses sein. Setze $C = \max_{e \in E} c(e)$
4) Damit wir ein **min** cost flow problem (mit nicht-negativen Gewichten) betrachten können, erstellen wir invertierte Kosten $c'(e) = C - c(e)$ (die Kanten von s und t bekommen Kosten 0).
5) Wir fügen k zusätzliche Kanten mit Kosten C von Schülern zu t oder von s zu den Studenten hinzu (je nachdem, welche Seite das minimum oben annimmt).$\rightarrow$ Dadurch wird sichergestellt, dass der maximale Flusswert k ist. Und es gibt die Möglichkeit ein nicht-maximales Matching zu berechnen, wenn ein anderes bessere Kosten hat.
6) Alle Kanten erhalten Kapazität 1.

Matching als Flussalgorithmus

Die Kosten eines maximum flows f sind gegeben als

$$\sum_{e \in E'} f(e)c'(e) = kC - \sum_{e \in E} f(e)c(e)$$

wobei E' die Kanten der modifizierten Instanz bezeichnet. Da kC konstant ist, wird dieser Wert minimiert, wenn $\sum_{e \in E} f(e)c(e)$ maximiert wird, also genau dann wenn ein kostenmaximales Matching ausgewählt wird.

- Achtung: Bereits bei 1000 Schülern und 1000 Studenten gibt es bereits mehrere hundert tausend Kanten. Die Laufzeit des Algorithmus (mit successsive shortest path) liegt in $\mathcal{O}(|V||E|)$. D.h. ab etwa 10^4 Studenten und Schülern ist der Ansatz möglicherweise nicht mehr ohne Weiteres praktikabel.

Matching als Flussalgorithmus

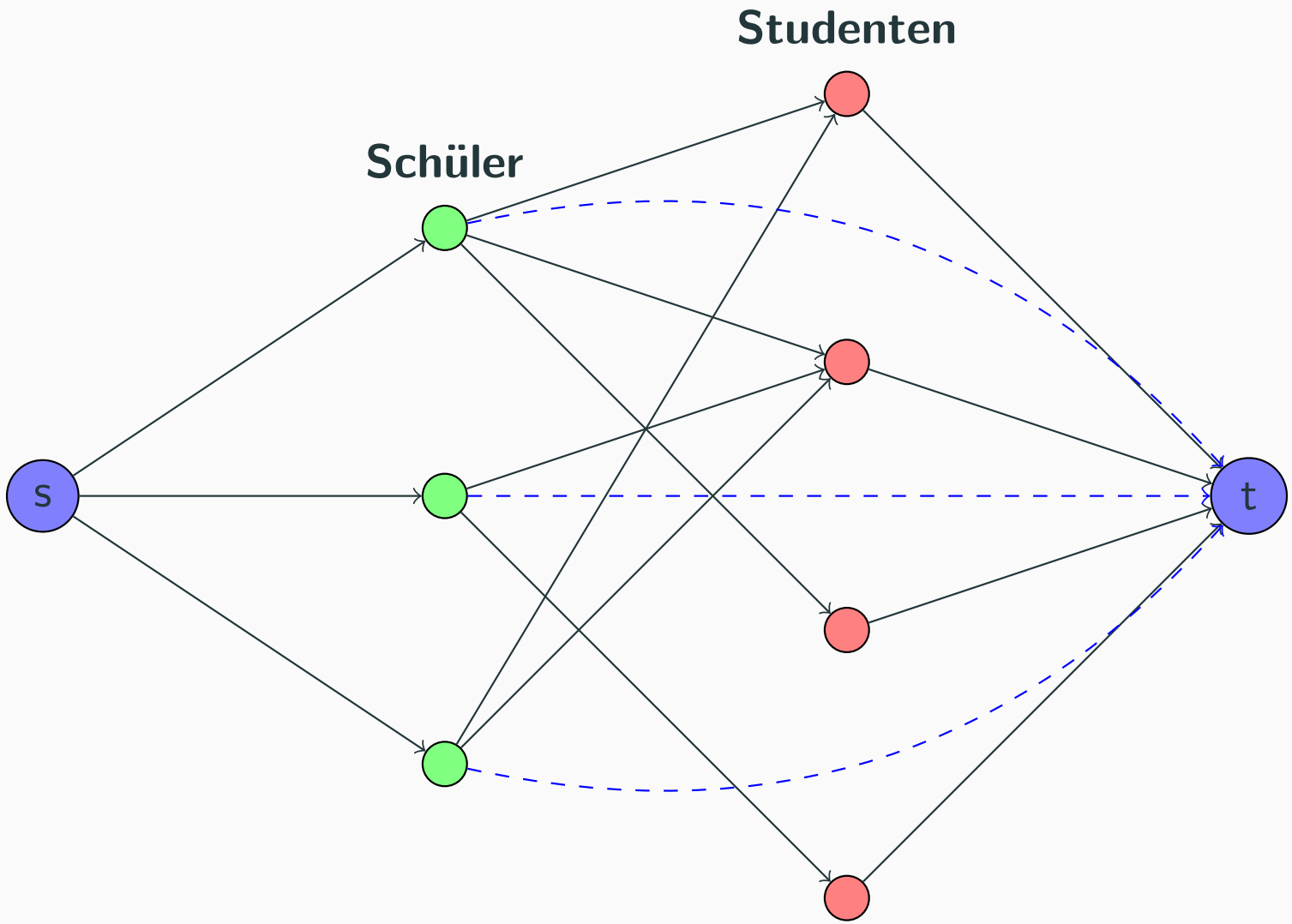


Figure 2: Die minimum cost flow Instanz (Kanten aus Schritt 5) in blau)

Matching als Flussalgorithmus

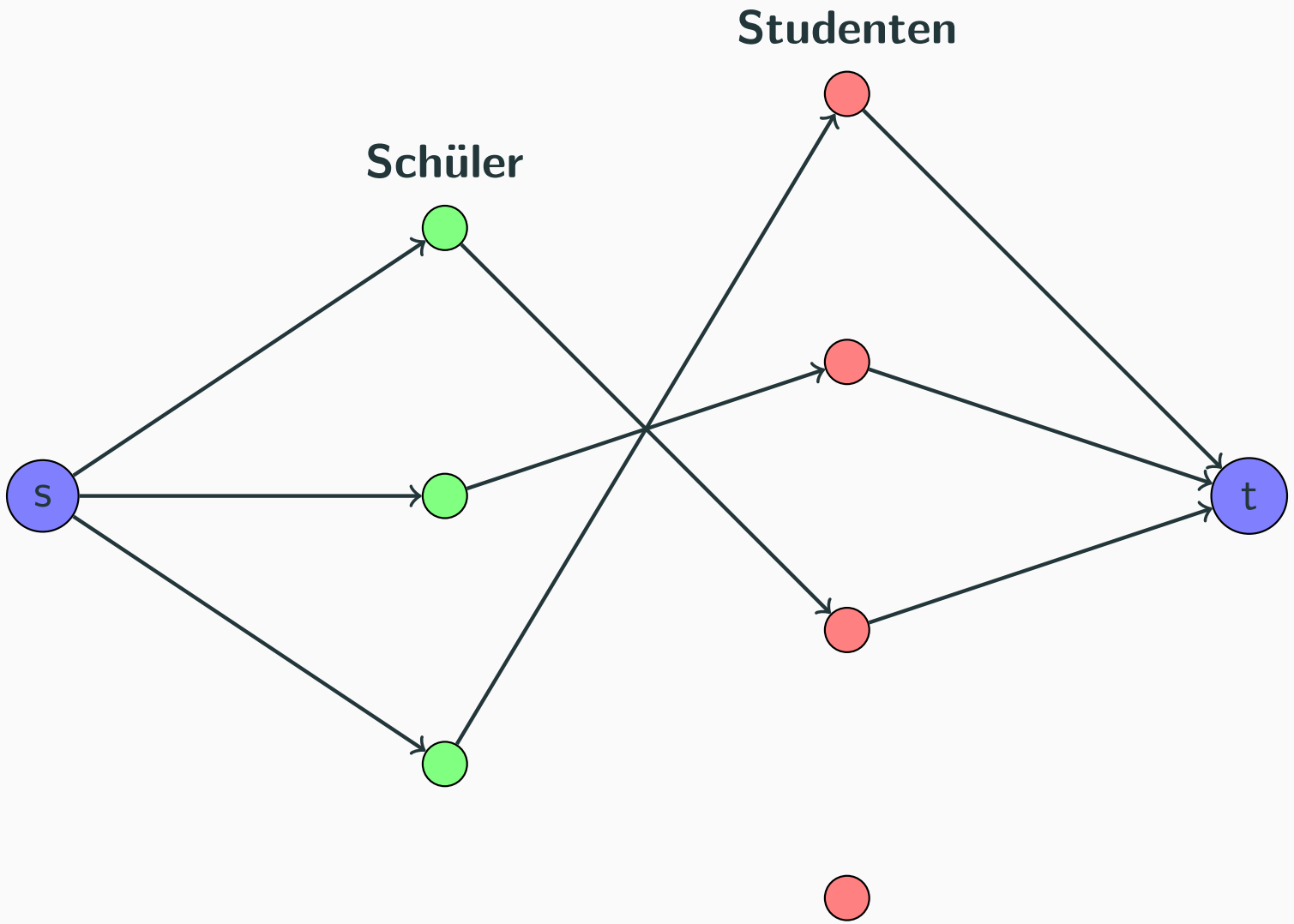


Figure 3: Kanten des berechneten minimum cost flow

Matching als Flussalgorithmus

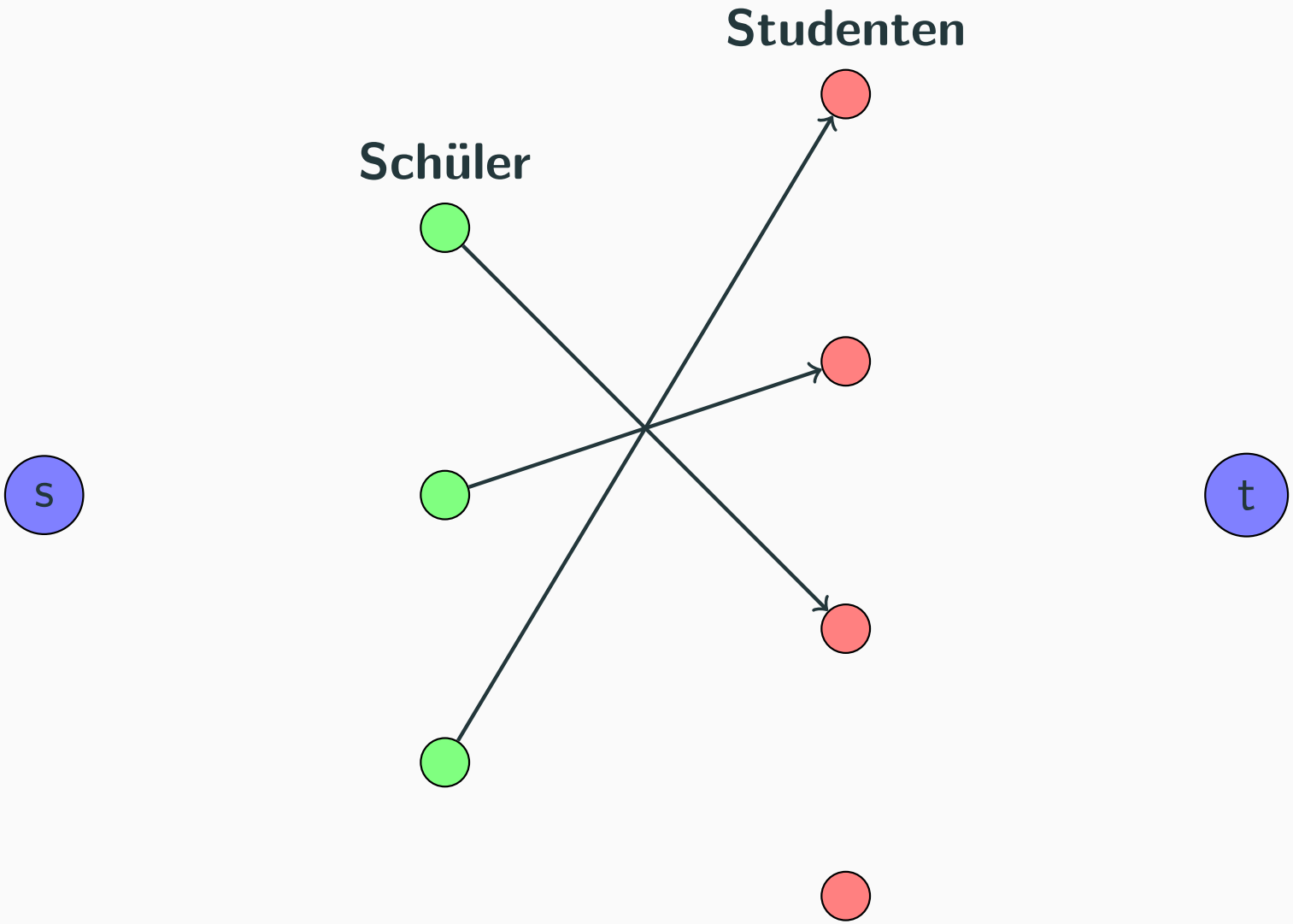


Figure 4: Die entsprechenden Kanten des Matchings

Boost Graph Library

Wir lösen das maximum cost flow problem entweder mit

- dem Successive-Shortest path Algorithm: https://www.boost.org/doc/libs/1_62_0/libs/graph/doc/successive_shortest_path_nonnegative_weights.html
- dem Cycle-Canceling Algorithm: https://www.boost.org/doc/libs/1_62_0/libs/graph/doc/cycle_canceling.html

der Boost Graph Library https://www.boost.org/doc/libs/1_62_0/libs/graph/doc/index.html

- Im Code kann man auswählen, welcher Algorithmus verwendet wird.
- Standardmäßig verwenden wir den Successive-Shortest Path Algorithmus, da dieser normalerweise schneller ist.

Output

Ausgabedateien

Der Algorithmus gibt zwei JSON-Datein aus:

1. Die gematchten Paare anhand ihrer UUID's die in den Eingabedateien spezifiziert sind als Liste mit Paaren von Elementen
   $\{$*"pupil uuid:":"UUID","student uuid:":"UUID"*$\}$
2. Statistiken über das gefundene Matching. Beispiel:
   `https://github.com/corona-school/matching_new/blob/master/examples/matching_stats.json`

Testläufe mit verschiedenen Koeffizienten

Ergebnisse eines Testlaufs mit 1000 Schülern und 1000 Studenten aus der Corona School Datenbank mit verschiedenen Gewichtungen:

(p_1, p_2, p_3)	# Matches	# FÜ	LW	# BÜ
(1,0,0)	967	2209	208	4
(0,1,0)	999	1261	15	3
(0,0,1)	60	73	221	60
(0.75, 0.2, 0.05)	991	2209	60	41

Dabei bezeichnet (p_1, p_2, p_3) die Gewichtungskoeffizienten für Fächerübereinstimmungen (FÜ), Wartezeiten und Bundeslandübereinstimmungen (BÜ). LW bezeichnet die längste Wartezeit eines ungematchten Schülers in Tagen.

$\Rightarrow$ Koeffizienten können verwendet werden, um Ergebnisse zu erzielen die in jeder Kategorie gut sind!

Mögliche Erweiterungen

Mögliche Erweiterungen

- **Multi-matchings**: z.B. Studenten könnten mehr als einen Schüler betreuen. $\rightarrow$ Modellierbar durch Anpassung der min-cost flow Instanz.
- **Andere Kostenkomponente**: z.B. Fächerpriorisierung, Berücksichtigung der Distanz, ... $\rightarrow$ Leicht modellierbar durch zusätzliche Datenpunkte und neue CostComponents.
- **Automatische Feedback Loops**: Algorithmus könnte mehrfach laufen und iterativ die Gewichtungskoeffizienten anpassen. **Problem**: Nicht klar, wie Algorithmus beurteilen kann welche Matchings "besser" sind als andere.