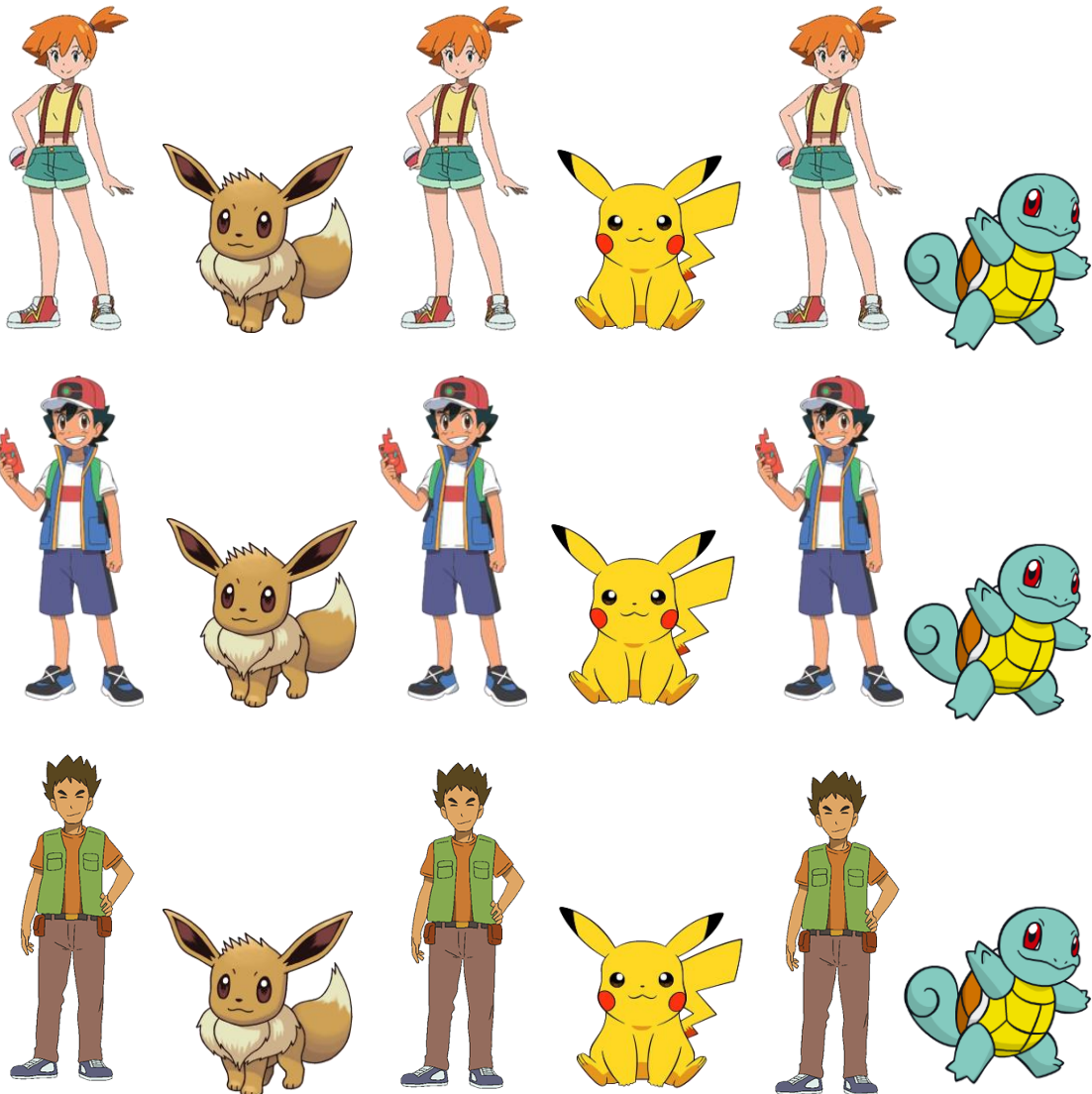


{ Gale Shapley Algorithmus und das Stable Marriage Problem }

Bericht von Nicolas Gujer zu Topic 10



{ Glossar }

Proposer/Trainer → Die Gruppe des Gale Shapley Algorithmus, die auf der “anfragenden” Seite ist. Beim Algorithmus wird diese Gruppe die andere anfragen, um Paare zu bilden.

Proposed/Pokémon → Die Gruppe des Gale Shapley Algorithmus, die auf der “angefragten” Seite ist. Beim Algorithmus wird diese Gruppe angefragt werden und muss auf diese reagieren. Also entweder ablehnen oder akzeptieren.

Entität → Steht für Proposer/Proposed ohne auf die spezifische Gruppe einzugehen

Paar/Match → Paar im Sinne des Gale Shapley Algorithmus. Gebildet aus 2 Entitäten aus 2 unterschiedlichen Gruppen.

Gruppengröße (n) → Beim Gale Shapley Algorithmus wird oft von den Gruppen geredet. Die Gruppengröße ist hierbei die Größe der einzelnen Gruppen und nicht die Anzahl aller Entitäten. Hat man also eine Gruppengröße von 50, gibt es insgesamt 100 Entitäten.

{ Stable Marriage Problem }

Das Stable Marriage Problem (oder auch Stable Matching / SMP) beschreibt ein mathematisches Problem. Es setzt sich mit der Frage auseinander, wie man sogenannte “Stable matches” zwischen zwei gleichgrossen Gruppen finden kann. Um diese zu bilden, hat jede der Entitäten aus den zwei Gruppen eine strikte Präferenzenliste von allen Entitäten der anderen Gruppe. Für zwei Gruppen existieren viele verschiedene mögliche stabile Möglichkeiten die Entitäten einander zuzuweisen.

Ein Paar/Match ist unstabil, falls folgende zwei Kriterien gegeben sind:

- Eine Entität A bevorzugt eine andere Entität B gegenüber dem jetzigen Partner
- Diese andere Entität B bevorzugt auch die Entität A gegenüber dem jetzigen Partner

Einfacher ausgedrückt: Ein stabiles Matching der Gruppen A und B ist gegeben, wenn kein Paar (a,b) existiert, dass sich gegenseitig vorzieht gegenüber ihren jetzigen Partnern.

Wichtig hierbei ist:

1. Die Matches sind strikt “Hetero”. Das bedeutet, dass nur Entitäten aus Gruppe A mit Entitäten aus Gruppe B (und umgekehrt) ein Paar bilden können.
2. Die Paargröße beträgt 2. Es sind also nur Paare zwischen zwei Entitäten möglich.

Anwendungen im echten Leben

3. In den USA werden Medizinstudenten nach ihrer Ausbildung oft Praktika zugeteilt in diversen Spitälern. Beide Seiten stellen ihre Präferenzen.
4. Bei der Verbindung von Servern und Usern sind oft beide Seiten wichtig. Der User möchte einen Server mit möglichst tiefer Latenz. Der Server möchte einen User mit möglichst tiefem Verbindungspreis.
5. Früher hat man Hochzeiten leider arrangiert. Mit stabilen Paaren könnte man die Zukunft der Paare sichern.
6. Pokémon und Pokémon-Trainer müssen am Anfang einander zugewiesen werden.

Ähnliche Algorithmen

7. Stable roommates problem → Es wird nicht mehr zwischen zwei Gruppen unterschieden. Alle Entitäten sind in einem “Pool”, es ist also eine homogene Gruppe.

- Hospitals/residents problem (Paare > 2) → Spitäler können mehrere Entitäten der anderen Gruppe aufnehmen. Die gebildeten Gruppen können also grösser als 2 werden. Zusätzlich gibt es noch die Variante bei der die Proposers angeben können mit bestimmten anderen Proposern in der Gruppe zu sein.

{ Gale Shapley Algorithmus }

Der Gale Shapley Algorithmus ist eine Lösung für das Stable Marriage Problem. Er wurde von David Gale und Lloyd Shapley entwickelt. Im Jahr 2012 wurden beiden der Nobel Memorial Prize in Economic Sciences verliehen. Leider ist Gale im Jahre 2008 schon verstorben. Der Algorithmus garantiert, dass jede Entität einen Partner zugewiesen bekommt und alle Pärchen stabil sind.

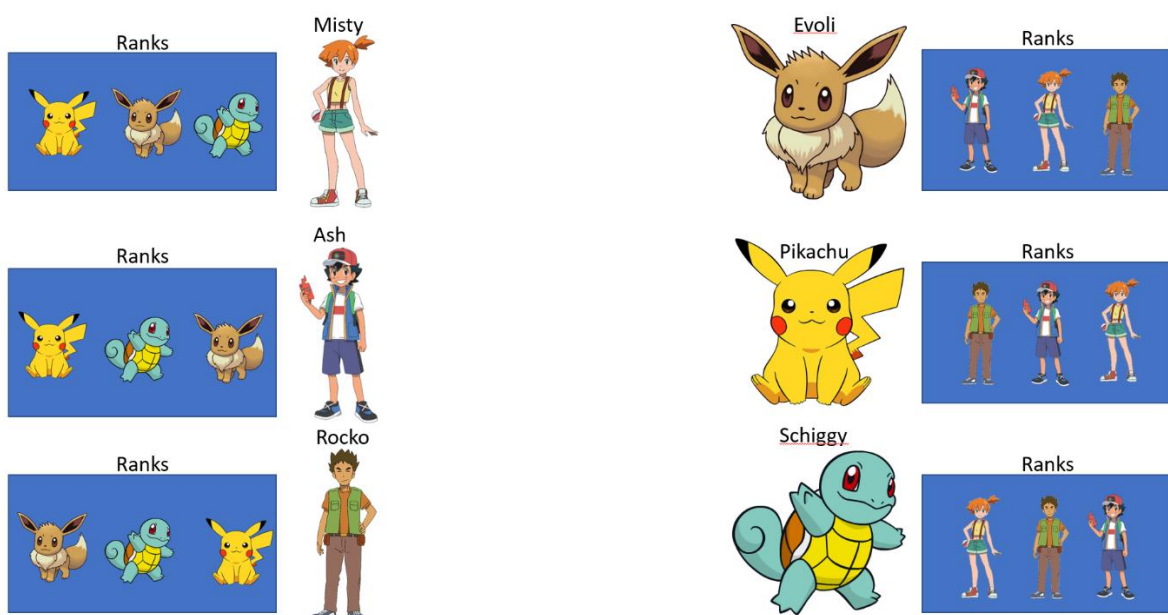
Funktionsweise

Der Gale Shapley Algorithmus ist in "Phasen/Iterationen" gegliedert.

- Jeder Proposer, der noch keinen Partner hat, fragt die ihm am liebsten, noch nicht von ihm angefragte Proposed an.
- Falls die angefragte Proposed noch in keinem Paar ist, wird ihr der Proposer zugewiesen. Falls sie schon einen Partner hat, schaut sie, ob sie den anfragenden Proposer lieber mag oder nicht. Falls ja, wird der momentane Partner freigestellt. Der Proposer und Proposed bilden ein neues Paar
- Dieser Prozess wird wiederholt bis alle Proposer einen Partner haben

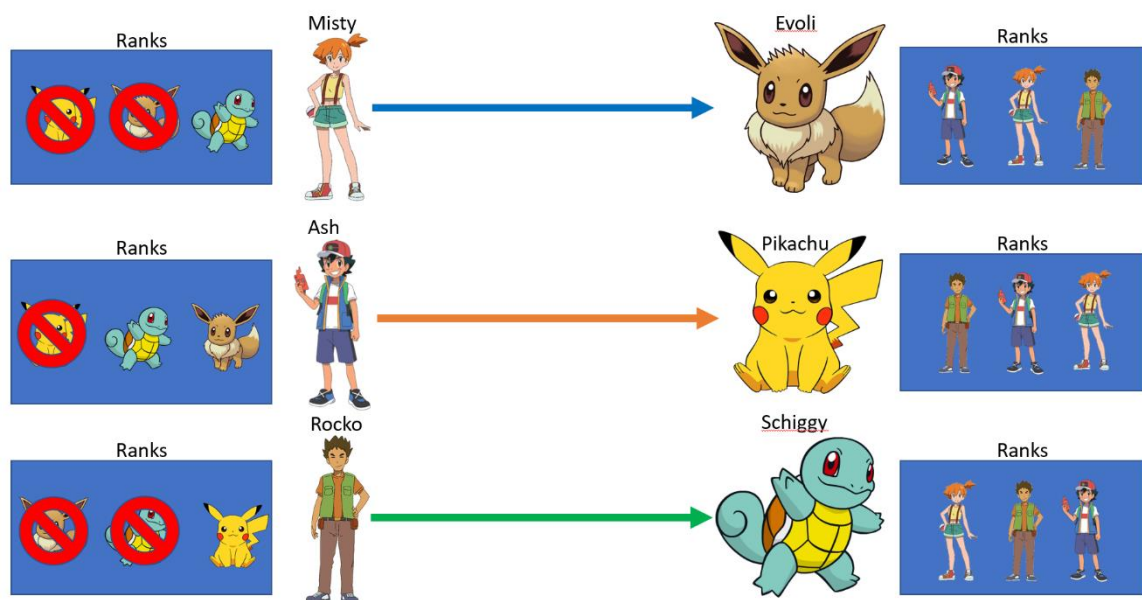
Praktisches Beispiel

Als Beispiel nehmen wir dieses Szenario. Wir haben 3 Pokémon Trainer als anfragende Gruppe und 3 Pokémon als angefragte Gruppe. Die Präferenzen sind jeweils in den blauen Boxen beschrieben. Die Rangfolge ist von links nach rechts (Ash mag Pikachu am liebsten und Evoli am wenigsten). Dieser Algorithmus verläuft etwas anders als der Java Code den ich später Zeige. Das hängt aber nur damit zusammen, dass beim Java Code eine FiFo Liste genutzt wird und hier eine LiFo. Auf das Endresultat hat dieser Unterschied aber keinen Einfluss.



Ablauf:

1. Misty ist an der Reihe und fragt Pikachu an. Da Pikachu noch keinen Partner hat, werden die beiden gematched.
2. Ash ist an der Reihe. Er fragt auch Pikachu an da es seine höchste Präferenz ist. Da Pikachu schon eine Partnerin hat, muss geschaut werden, wer Pikachu präferiert. Da Pikachu Ash lieber mag wird die Beziehung zu Misty aufgelöst. Ash und Pikachu sind nun ein Match.
3. Misty ist wieder an der Reihe, da sie nun wieder allein ist. Sie fragt Evoli an. Evoli ist allein, deshalb sind Misty und Evoli nun ein Match.
4. Rocko ist an der Reihe. Er fragt Evoli an, dass aber schon einen Trainer hat. Evoli präferiert Misty gegenüber Rocko, er wird abgelehnt.
5. Rocko fragt seine zweite Präferenz (Schiggy). Da Schiggy noch keinen Trainer hat, sind die beiden nun ein Match.
6. Alle Trainer haben ein Pokémon. Der Algorithmus ist beendet mit folgenden Paaren:



Bemerkungen

1. Der Algorithmus ist deterministisch. Das bedeutet, es ist egal in welcher Reihenfolge die Proposer abgearbeitet werden, die Paare bleiben die gleichen.
2. Der Algorithmus ist immer optimal für die anfragende/Proposer Gruppe. Die gebildeten Paare sind optimal für die Proposer Gruppe. Das heisst aber nicht, dass jeder Proposer mit seiner ersten Präferenz zusammenkommt. Im Umkehrschluss ist der Algorithmus pessimistisch für die Proposed Gruppe
3. Der Algorithmus ist für die Proposed Seite ein sogenannter "Truthful mechanism"
 - a. Die Proposer können das Resultat nicht beeinflussen, indem sie extra falsche Präferenzen angeben, um ein besseres Resultat zu erhalten.
 - b. Auch die ganze Gruppe der Proposer als Ganzes können das nicht
4. Der Algorithmus ist für die Proposed Seite ein "Non-Truthful mechanism"

- a. Falls eine Proposed die Präferenzen aller Entitäten kennt, könnte sie den Algorithmus mit gewissen Strategien beeinflussen.

Pseudo Code und Running Times Analyse

```
Algorithm galeShapleyMatch(trainers, trainersPreferences, pokemon, pokemonPreferences)
    unmatched(pokemon, trainers)
    while (!trainers.isEmpty())
        trainer = trainers.poll()
        for (nextPokemon in trainersPreferences of trainer)
            if(isUnmatched(nextPokemon))
                match(nextPokemon, trainer)
                trainers.remove(trainer)
                break;

        else
            otherTrainer = getMatch(nextPokemon)
            if(prefers(nextPokemon, trainer)) {
                unmatched(nextPokemon, otherTrainer)
                match(nextPokemon, trainer)
                trainers.add(otherTrainer)
                trainers.remove(trainer)
                break;
```

Abbildung 1: Pseudo Code für Gale Shapley Algorithmus

Dieser Pseudo Code wurde extra simpel gestaltet. Auf Datentypen und andere spezifische Elemente wird dann im Java Code eingegangen. Die Basis ist das Beispiel der Trainer und Pokémon. Es wird angenommen, dass alle Funktionen die verwendet werden in $O(1)$ ablaufen. Das beinhaltet zum Beispiel die Funktionen "match()", "prefers()", "unMatch()" und "nextPokemon in trainersPreferences of trainer". Gerade bei der letzten Methode versteckt sich, dass dort nur die Pokémons angefragt werden, die der Trainer noch nicht gefragt hat.

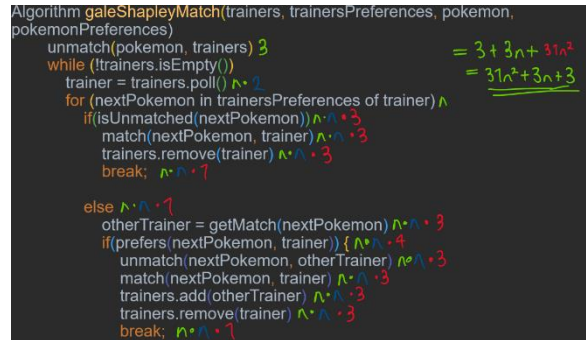
Best Case $O(N)$ → Im Best Case Szenario sind die Präferenzen so aufgebaut, dass zum Beispiel alle Trainer ein anderes Pokémon als erste Präferenz haben. Somit wäre der Algorithmus nach N Iterationen schon fertig. Jeder Trainer muss nur einmal anfragen und es gibt keine Kollisionen mit anderen Trainern.

Worst Case $O(N^2)$ → Im Worst Case Szenario sind die Präferenzen so aufgebaut, das vereinfacht gesagt jeder Trainer alle seine Präferenzen durcharbeiten muss und somit jedes Pokémon einmal anfragt. Da es N Trainer und N Pokémon in der Präferenzenliste hat, wird es eine Running time von $O(N^2)$ ergeben. Wenn man es genauer anschaut, ist die Running time $n(n-1)+1$. Ein Trainer kann ja nicht N mal abgelehnt werden, sonst hätte er kein Pokémon und der Algorithmus würde kein stable matching bilden. Das Big Oh wäre aber immer noch $O(N^2)$.

Genaue Analyse Anhand des Pseudo Codes →

Obwohl es eher unwichtig ist und auf das Big Oh keine Auswirkung hat, versuche ich noch die Time Complexity anhand des Pseudo Codes zu berechnen. Dazu schreibe ich Notizen auf das Bild. Ich komme somit auf ein Resultat von:

$$31N^2 + 3N + 3$$



```
Algorithm galeShapleyMatch(trainers, trainersPreferences, pokemon,
pokemonPreferences)
  unmatched(pokemon, trainers) 3
  while (!trainers.isEmpty())
    trainer = trainers.poll() N * 1
    for (nextPokemon in trainersPreferences of trainer) N
      if (isUnmatched(nextPokemon)) N * 3
        match(nextPokemon, trainer) N * 3
        trainers.remove(trainer) N * 3
        break; N * 1
      else N * 1
        otherTrainer = getMatch(nextPokemon) N * 3
        if (prefers(nextPokemon, trainer)) { N * 4
          unmatched(nextPokemon, otherTrainer) N * 3
          match(nextPokemon, trainer) N * 3
          trainers.add(otherTrainer) N * 3
          trainers.remove(trainer) N * 3
          break; N * 1
        }
```

Handwritten annotations on the right side of the code:

- $3 + 3n + 31n^2$
- $= 31n^2 + 3n + 3$

Abbildung 2: Berechnung Time Complexity des Pseudo Code

Java

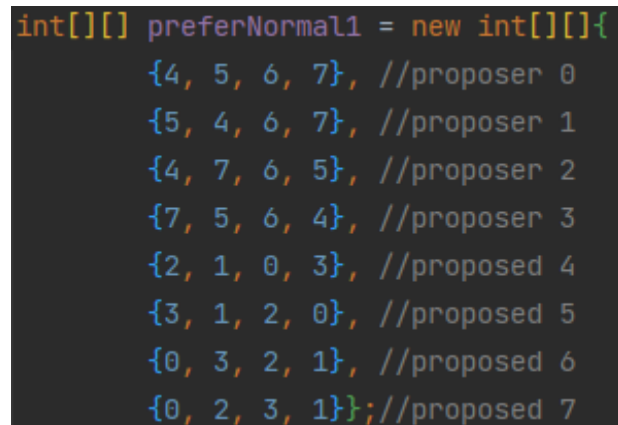
Die Implementation in Java ist der Pseudo Code Variante sehr ähnlich. Ich habe aufgrund Performance Problemen darauf verzichtet Objekte zu benutzen für die Implementation. Stattdessen besteht der Algorithmus aus mehreren Listen und Arrays die zusammen den State der Proposers/Proposed bestimmen. Damit lässt sich sehr viel Memory sparen.

Aufbau

Int[][] preferences → Dieser 2d-Array beherbergt sowohl die Entitäten selber wie auch deren Präferenzen in Form von integer Werten. Die Integer sind die Identifikatoren (id's) der einzelnen Entitäten. Er ist $\text{int}[\text{groupSize} * 2][\text{groupSize}]$ gross, da es $2 * \text{groupSize}$ Entitäten gibt, die jeweils eine Präferenzenliste so lang wie die groupSize haben. Die Proposer sind von 0 bis groupSize-1 nummeriert und die Proposed von groupSize bis $2 * \text{groupSize} - 1$. Damit der Algorithmus auch in seiner Theoretischen Time Complexity funktionieren kann, muss sowohl das iterieren durch die Präferenzen in Rangordnung der Proposed in constant time funktionieren wie auch das vergleichen der Präferenzen der Proposed Gruppe, falls sie schon in einem Match sind und angefragt werden. Deshalb sind die Präferenzenlisten der Proposer anders aufgebaut als die der Proposed

Proposer → Die Präferenzen sind so im Array abgelegt, dass die höchste Präferenz am Anfang und die tiefste am Schluss ist. Der Index ist somit gleichzeitig die Präferenz und die Zahl im Array ist der Proposer. An Index 0 ist die höchste und am letzten Index die tiefste. Das ermöglicht ein einfacher iterieren in der richtigen Reihenfolge.

Proposed → Die Präferenzen sind so angeordnet, dass in $O(1)$ die Präferenz von Proposer[i] abgefragt werden kann. Der Index stellt den Proposer dar und die Zahl den Rang der Präferenz. Wenn man also den Rang von Proposer[i] möchte, wird im Präferenzen Array einfach der Index [i] abgefragt. Der Output ist dann der Rang und nicht der Indikator des Proposer.



```
int[][] preferNormal1 = new int[][]{
    {4, 5, 6, 7}, //proposer 0
    {5, 4, 6, 7}, //proposer 1
    {4, 7, 6, 5}, //proposer 2
    {7, 5, 6, 4}, //proposer 3
    {2, 1, 0, 3}, //proposed 4
    {3, 1, 2, 0}, //proposed 5
    {0, 3, 2, 1}, //proposed 6
    {0, 2, 3, 1}}; //proposed 7
```

Abbildung 3: Preferences Array Beispiel

Als Beispiel: Proposer 2 hat Proposed 4 am liebsten und 5 am wenigsten. Proposed 7 hat Proposer 0 am liebsten und 2 am wenigsten.

proposerQueue → Eine FiFo LinkedList. Sie speichert alle Proposer, die noch in keinem Match sind. Die Liste ist von 0 bis groupSize-1 nummeriert.

proposedPartners → Speichert die Partner der Proposed Entitäten. Die Nummern im Array selbst stellen die einzelnen Proposer dar. Am Anfang ist der Array gefüllt mit -1, was bedeutet, dass keine der Proposed einen Partner hat. `proposedPartners[i]` ist der Partner von der Proposed Entität `groupSize+i`.

proposerState → Speichert den State der einzelnen Proposer. Dieser Array ist benötigt, um zu verhindern, dass die Proposer mehrmals die gleiche Proposed anfragen.

```
public int[] findCouples(int[][] preferences) {  
  
    if (preferences.length != this.groupSize*2 || preferences[0].length != this.groupSize){  
        throw new IllegalStateException("Group size does not match preference array size");  
    }  
    Queue<Integer> proposerQueue = new LinkedList<>();  
    for (int i = 0; i < groupSize; i++) {  
        proposerQueue.add(i);  
    }  
    int[] proposedPartners = new int[groupSize];  
    Arrays.fill(proposedPartners, -1);  
    int[] proposerState = new int[groupSize];  
    Arrays.fill(proposerState, -1);  
    while (!proposerQueue.isEmpty()) {  
        int proposer = proposerQueue.poll();  
        for (int i = proposerState[proposer]+1; i < groupSize; i++) {  
            int proposed = preferences[proposer][i];  
            if (proposedPartners[proposed - groupSize] == -1) {  
                proposedPartners[proposed - groupSize] = proposer;  
                proposerState[proposer] = i;  
                break;  
            }  
            else {  
                int currentMatch = proposedPartners[proposed - groupSize];  
                if (proposedPrefersProposerOverMatch(preferences, proposed, proposer, currentMatch)) {  
                    proposedPartners[proposed - groupSize] = proposer;  
                    proposerQueue.add(currentMatch);  
                    proposerState[proposer] = i;  
                    break;  
                }  
            }  
        }  
    }  
    return proposedPartners;  
}
```

Probleme mit Java und dem Algorithmus

Meine ersten Versuche waren in einer anderen Form als das Endprodukt. Ich wollte mit Java Objekten arbeiten. Dazu habe ich Proposer und Proposed Objekte gemacht. Diese hatten eine Liste von der jeweils anderen Gruppe als Klassenvariable. Bei den Proposed war diese eine HashMap. So konnte auch in $O(1)$ die Präferenzen verglichen werden. Das Problem bei dieser Lösung war, dass ich um einiges weniger grosse Gruppen erstellen konnte. Schon bei etwa einer Gruppengrösse von 8000 gab es eine Out of Memory Heap exception. Der jetzige Algorithmus kann über das doppelte. Dies hat damit zu tun, dass Objekte zu speichern in anderen Objekten eine sehr grosse Menge an Memory verbraucht im Vergleich zu den 4 Bytes eines Integer Wertes. Auch als ich es versucht habe mit eines HashMap und einer speziellen Liste gab es Probleme. Der Grund war, dass überall in Java wo Generics gebraucht werden, kann man nicht "int" verwenden, sondern muss die Wrapper Klasse

“Integer” verwenden. Diese verbraucht 4-mal mehr Memory als normale int werte (4 vs 16 bytes). Die jetzige Lösung ist leider nicht sehr intuitiv lesbar, aber gibt die beste Performance, die ich erreichen konnte.

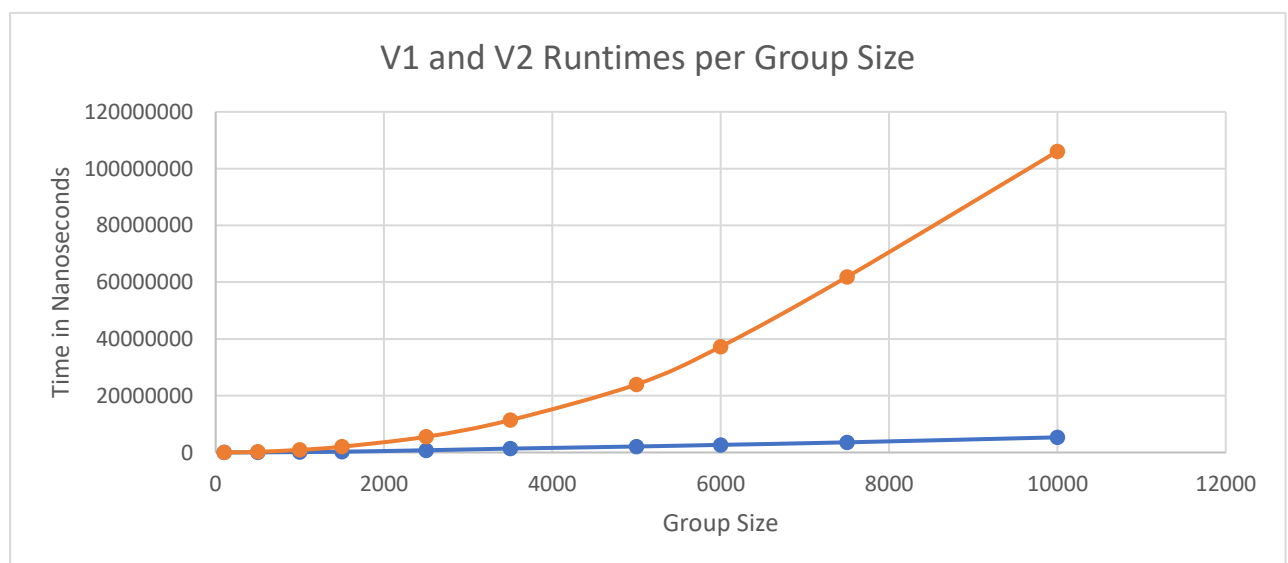
Da ein int Wert in Java etwa 4 Byte verbraucht und der Array aus “gruppenGrösse * gruppenGrösse * 2” Elementen besteht, sollte sich theoretisch gut berechnen können was die maximale Gruppengrösse bei gegebenem Heap Size ist. Bei einer Heapsize von 2GB sollte sie etwa bei 16’000 liegen ($16'000 * 32'000 * 4 = 2.048.000.000$ Bytes). Beim austesten konnte ich das so auch feststellen. Eine Gruppengrösse von 16’000 funktionierte nicht mehr aber eine von 15’000 funktionierte noch.

In einer V1 Version hatte ich dann auch dieselbe Implementation mit dem 2d-Array. Dieser war aber gleich aufgebaut für Proposer und Proposed. Damit konnte der Rang von den Proposern nicht in $O(1)$ verglichen werden. Zusätzlich wurde auch der State der Proposer nicht gespeichert. Im Algorithmus haben die Proposer mehrmals dieselbe Proposed angefragt, was auch zu Einbussen in der Performance geführt hat. Die Performance dieses alten V1 Algorithmus ist um einiges schlechter (siehe Kapitel Performance).

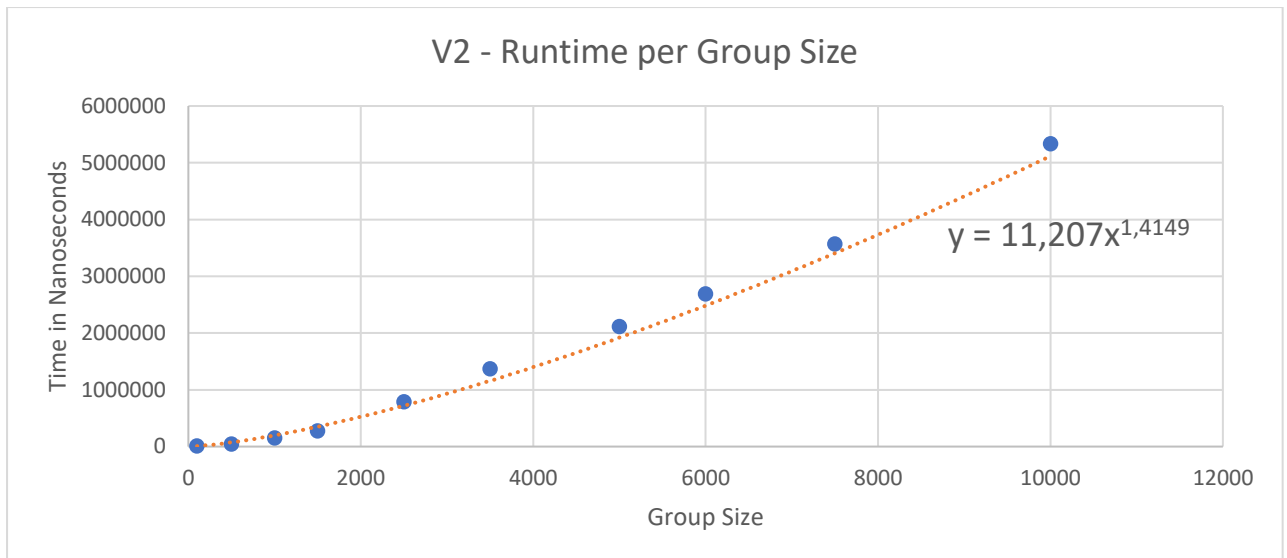
Performance

Um die Performance der Algorithmen zu messen, wurde eine kleine Testklasse entwickelt. Das Ziel war es die theoretischen Time Complexity Berechnungen mit echten Daten zu vergleichen. Dazu wurden die Algorithmen V1 und V2 in diversen Gruppengrössen getestet. Dazu wurden die generateRandomData() Methoden verwendet, die möglichst zufällige 2d-Arrays generieren. Da der 2d-Array etwas anders aufgebaut ist in V1 zu V2 waren es nicht dieselben Umstände beim Testen. Um Dagegenzuwirken wurde eine grosse Iterationsgrösse von 1000 verwendet. Beim Beobachten der Daten wurde schnell klar, dass die “Aufwärmzeit” zwischen 5 und 25 Iterationen beträgt. Ich habe diese ignoriert, da ich den Mittelwert auf 1000 Iterationen gezogen habe.

Die Testmethode hat den gleichen Algorithmus 1000-mal ausgeführt mit anderen zufälligen Daten. Die Zeit wurde in Nanosekunden gemessen. Es wurde darauf geachtet, dass der Computer immer im möglichst gleichen Zustand war. Dazu wurden alle Prozesse/Apps geschlossen ausser IntelliJ und Java. Bei Windows 10 kann man sich aber nie genau sicher sein ob nicht irgendein Hintergrundprozess am Laufen ist.

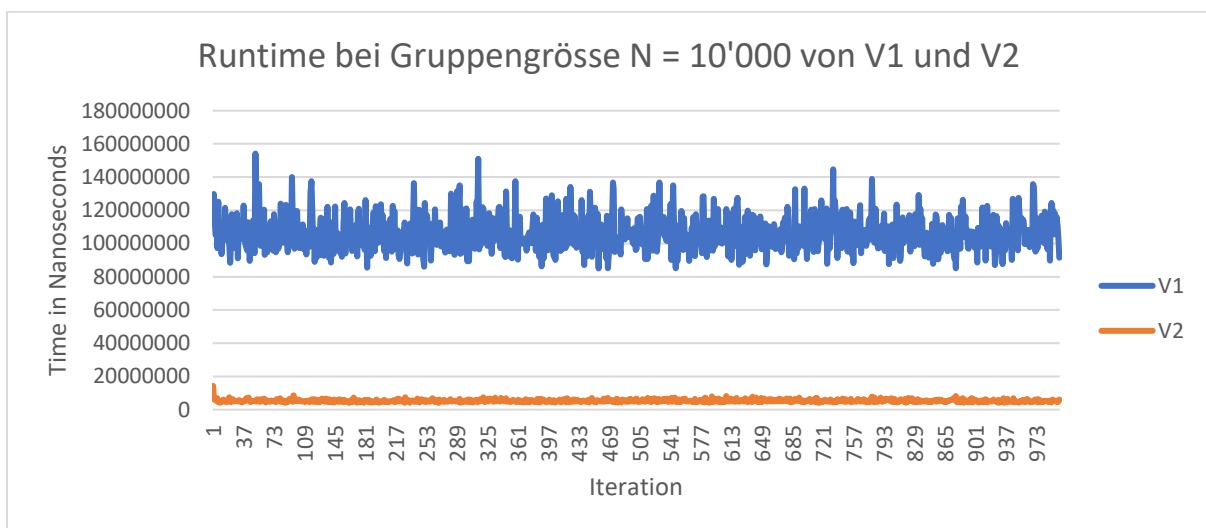


Graph 1: V1 und V2 Runtime pro Gruppengrösse



Graph 2: Runtimes Graph vom V2 Algorithmus

Man kann in Graph 2 gut erkennen, dass die Theorie nicht wirklich mit der Praxis übereinstimmt. Excel besagt bei einer Trendlinie (Exponentiell) eine Formel von $y = 11,207x^{1,4149}$ aus. Die Daten sehen aber nicht wirklich Exponential-Verteilt aus. Dies hat wohl damit zu tun, dass die meisten zufällig generierten Daten nicht annähernd ein Worst-Case waren und der Algorithmus in viel weniger Iterationsschritten fertig war. Es gibt auch kleine Ausreisser wie der Datenpunkt bei 3500, dies kann diverse Gründe haben.



Graph 2: Runtime V1 und V2 bei einer Gruppengröße von 10'000

Graph 3 und 1 zeigen gut auf wie viel schneller der V2 Algorithmus ist gegenüber dem V1. Im Durchschnitt bei einer Gruppengröße von 10'000 ist der V2 fast 20-mal schneller. Auch sieht man wie stark die Abweichungen sind vom Mittelwert. Der Zustand der Daten scheint also einen starken Einfluss auf die Runtime zu haben. V1 scheint auch eher eine exponentielle Runtime zu haben als V2. Hier schliesse ich auch wieder darauf, dass die zufälligen Daten diesen Unterschied von theoretischer und tatsächlicher Runtime ausmachen. Die vielen zusätzlichen Schritte, die der V1 Algorithmus durchgehen muss, sieht die Verteilung eher exponentiell aus. Der Worst Case bei dem V1 Algorithmus sollte theoretisch sogar noch höher als $O(N^2)$ sein. Eine genaue Berechnung lasse ich hier aus.