# Zur Variablenbenennung

Vorregistrierung = Anmeldung = registration

Abgelehnt = rejected

Angenommen = accepted

# Lösungsidee

Meine erste Lösungsidee war es, allen Voranmeldungen zufällig eine Position zuzuteilen und daraufhin per simulated Annealing die Lösung zu optimieren. Da ich aber auf diesem Weg keine ausreichend gute Lösungen gefunden habe, habe ich die erst-Positionierung sowie die möglichen Positionsveränderungen der Anmeldungen in mehreren Stufen eingeschränkt bzw. präzisiert. Am Ende hat allein das geschickte Positionieren (ohne Optimierung danach) zu den Besten Lösungen geführt.

# Umsetzung

Ich habe mich für die Implementierung in der objektorientierten Programmiersprache C# entschieden.

Ich möchte den Programmablauf chronologisch beschreiben. Das Programm ist relativ modular aufgebaut, um möglichst einfach zwischen den verschiedenen „Eskalationsstufen“, die ich in der Lösungsidee schon angeschnitten habe, wechseln zu können. Elementar ist das Tuple aus zwei Listen von akzeptierten und abgelehnten Anmeldungen. Jede Anmeldung ist Element der Klasse \_\_ und beinhaltet die Parameter id (Zeilennummer aus Datei), Mietbeginn, Mietende, Mietdauer, Länge und Position.

In der Main-Funktion kann man zuerst die Rahmenbedingungen, wie zB die Flohmarktlänge und -dauer, festlegen (in dieser Hinsicht ist das Programm völlig flexibel konzipiert). Daraufhin wird die angegebene Datei eingelesen und die gelesenen Daten auf Plausibilität geprüft (sie Erweiterung \_\_). Nun wird ein Objekt des Typs \_\_ erstellt, einer Klasse, die für die gesamte Verarbeitung der Daten zuständig ist. Nun kann eine Methode gewählt werden, mit der die Anmeldungen auf die Positionen verteilt werden. Es gibt einmal den randomisierten Ansatz \_\_ und eine durch mitgegebene Argumente auf Wunsch angepassten besseren Ansatz \_\_. Jetzt kann das optimieren durch simulated Annealing stattfinden. Zum Schluss werden die Ergebnisse ausgegeben und auf Wunsch gespeichert. Außerdem können die Erweiterungen \_\_ und \_\_ durchgeführt werden.

Meine als solche beschriebenen Erweiterungen beziehen sich vor allem auf die Verarbeitung des Ergebnisses, nicht das Ergebnis an sich. Theoretisch kann man aber auch die Ergänzungen zu mehreren Programmvarianten (mehrere Energieberechnungen, mehrere Veränderungsvarianten, mehrere Positionierungsvarianten) als Erweiterungen betrachten.

Um sich das Problem visuell besser vorzustellen, greife ich in den nächsten Abschnitt immer wieder das Bild einer Tabelle auf, die in x-Richtung die Position der Tische und in y-Richtung die Zeit angetragen hat. So entspricht die Fläche einer Anmeldung ihrer Miete.

# Positionierung

## setRandomPositions2

Diese Funktion setzt die Voranmeldungen auf zufällige Positionen. Ein angegebener Anteil an den Anmeldungen wird auf die Liste der abgelehnten Anmeldungen gesetzt.

## setPositions5

Eine sehr effektive Methode um gleich beim positionieren eine sehr gute Verteilung zu erzielen war es, die Liste der Anmeldungen nach „Sperrigkeit“ zu sortieren; je unpraktischer das Format der Anmeldung auf der vorher beschriebenen Tabelle, desto mehr Möglichkeiten verbaut man sich. Die Sperrigkeit wird entweder anhand des Flächeninhalts bestimmt (\_\_) oder anhand der Mietdauer (\_\_). Theoretisch könnte auch die Länge ein Sperrigkeitsindikator sein – da aber alle Anfragen eher lange Dauern als viele Tische belegen (betrachtet im Verhältnis zur Gesamtdauer und Flohmarktlänge) ist die Dauer der sinnvollere Parameter.

public void setPositions5((bool sorted, bool comparer5, bool optimalPos) state) {

(List<Anfrage> verwendet, List<Anfrage> abgelehnt) anfragenLoc = cloneLists(anfragen);

anfragen.verwendet.Clear(); anfragen.abgelehnt.Clear();

if (state.sorted) {

if (state.comparer5) { anfragenLoc.verwendet.Sort(compareByLength5); }

else { anfragenLoc.verwendet.Sort(compareByRent4); }

}

foreach (Anfrage afr in anfragenLoc.verwendet) {

List<int> freePositions = findFreePositions5(unoccupiedFields, afr);

if (freePositions.Count > 0) {

if (state.optimalPos) {

afr.position = findBestPosition5(unoccupiedFields, afr, freePositions)[0];

}

else { afr.position = freePositions[rnd.Next(freePositions.Count)]; }

anfragen.verwendet.Add(afr);

unoccupiedFields = setAfrUnoccupiedFields(unoccupiedFields, afr, false);

}

else {

afr.position = -1;

anfragen.abgelehnt.Add(afr);

}

}

output = (energyChart(anfragen.verwendet), new List<int>(), new List<int>(), anfragen, anfragen);

}

Eine weitere Möglichkeit um die Ergebnisse zu verbessern ist es, schon beim Positionieren keine Überschneidungen zwischen Anmeldungen zuzulassen und die Bedingung nicht erst in die Energiefunktion im Optimierungsalgorithmus einfließen zu lassen. Dazu findet die Funktion \_\_ für jede Anmeldung alle möglichen Positionen, bei denen keine Überschneidung auftritt.

Aus dieser Liste der möglichen Positionen wählt das Programm entweder eine zufällige aus, oder die, die am wenigsten freie Fläche verdeckt (siehe \_\_). Sollte keine Position mehr frei sein, wird die Anmeldung auf die Liste der abgelehnten Anmeldungen gesetzt.

Das zweidimensionale Boolean Array \_\_ dient primär einer kürzeren Laufzeit der Funktion \_\_. Da in dem Array für jeden Tisch zu jeder Stunde abgespeichert ist, ob er belegt ist, muss die Funktion nur durch das Array gehen und nicht zusätzlich bei allen Anmeldungen überprüfen ob sie in die fragliche Zeit fallen.

## findFreePositions5

Diese Funktion prüft jede Position im Flohmarkt auf ihre Eignung für die aktuelle Anfrage. Relevant ist dazu, ob sie sich an dieser Position mit einer anderen, schon positionierten Anfrage überschneidet.

## findBestPosition5

Diese Methode findet die optimale Position einer Anmeldung aus einer Liste aller freien Positionen. Kriterium für die optimale Position ist hierbei, möglichst wenig freie Fläche zu verdecken. Das bedeutet im Umkehrschluss meist, möglichst viel an bereits positionierte Anmeldungen anzugrenzen.

Dazu berechnet die Funktion die Fläche zwischen den Kanten der Anmeldung (in der Tabelle betrachtet) und den frühesten nächsten besetzten Tischen in jeder Richtung. Voraussetzung für die optimale Position ist, dass Min(flächeLinks, flächeRechts) + Min(flächeOben, flächeUnten) minimal ist.

# Optimieren

## Simulate

In der Funktion Simulate wird der Optimierungsalgorithmus „simulated Annealing“ ausgeführt. Hier die Funktionsweise im Struktogramm:

Sowohl die Energieberechnung, als auch die Veränderung („move“) kann variiert werden. In der Variante des Programms, die Überschneidungen zwischen Anmeldungen während des Optimierungsvorgangs grundsätzlich zulässt, muss die Energieberechnung diese Bedingung enthalten (siehe \_\_ und \_\_), während die Variante ohne Überschneidung eine andere Funktion braucht (siehe \_\_ und \_\_).

## Energy und move2

Energy berechnet die Kosten wie folgt: - Fläche aller angenommenen Anmeldungen + alle Überschneidungen \* x. X steigt bei sinkender Temperatur. Je weiter das simulated Annealing also fortgeschritten ist, desto wichtiger wird es, dass keine Überschneidungen vorkommen.

Die Funktion move2 macht eine von drei verschiedenen Arten von Veränderungen: verschieben der Anmeldung (50%), abgelehnte Anmeldung annehmen (25%) oder angenommene Anmeldung ablehnen (25%). Die Wahrscheinlichkeitsverteilung ist eine der vielen Schrauben, an denen man drehen kann um das Endergebnis zu verbessern. Je niedriger die Temperatur ist, desto kleiner werden die Positionssprünge bei der Variante „verschieben“.

## Energy2 und move4

Energy2 berechnet die Kosten wie folgt: - (Fläche aller angenommenen Anmeldungen die sich nicht überschneiden). Diese Energieberechnung wird bevorzugt bei der Programmvariante eingesetzt, die von Haus aus keine Überschneidung zulässt. Dann entspricht es: - (Fläche aller angenommenen Anmeldungen).

Auch beim move4 gibt es die drei oben beschriebenen Veränderungen verschieben, ablehnen, annehmen. Allerdings muss hier sichergestellt werden, dass bei einer Veränderung der Position keine Überschneidungen auftreten. Sowohl beim Annehmen einer Anmeldung als auch beim Verschieben findet die Funktion \_\_ alle Positionen für die Anfrage, an denen keine Überschneidung auftreten würde. Im Gegensatz zur Positionierung am Anfang ( \_\_) wird nicht die optimale Position ausgewählt, sondern eine Zufällige (simulated Annealing ist nicht-deterministisch).

# Erweiterungen

Wie schon eingangs erwähnt, beschränke ich mich im Folgenden hauptsächlich auf die Programm- und Aufgabenerweiterungen die keinen direkten Einfluss auf das Ergebnis haben. Auch die verschiedenen Programmvarianten sind als Erweiterungen zu verstehen.

Bei diesen Erweiterungen habe ich unter anderem überlegt, welche Funktionen und Daten für den Betreiber eines solchen Flohmarkts relevant sein könnten.

## Plausibilitätsprüfung (\_\_)

Nachdem die Daten eingelesen werden, prüft das Programm sie auf Plausibilität. Folgende Punkte werden geprüft:

* Mietbeginn der Anmeldung ist nach dem Start des Flohmarkts
* Mietende der Anmeldung ist vor Ende des Flohmarkts
* Anmeldung ist nicht länger (belegt mehr Tische) als Flohmarkt
* Mietbeginn der Anmeldung ist vor deren Mietende

Jede Anmeldung, bei der einer der Punkte nicht zutrifft, wird mit einer Fehlermeldung ausgegeben. Wenn alle Anmeldungen geprüft sind und es eine oder mehrere gab, die Fehlerhaft sind, kann das Programm entweder abgebrochen oder die betreffenden Anmeldungen entfernt werden.

## Grenzen

Diese Erweiterung beeinflusst als einzige der hier Genannten das Ergebnis. Sie ermöglicht es, örtliche Grenzen festzulegen, die von keiner Anmeldung überschritten werden darf. Dies musste an mehreren Stellen im Programm berücksichtigt werden:

* für die erste Programmvariante mit mehr Freiheiten (zufällige Positionierung; Überschneidungen werden über Energiefunktion minimiert) reicht es, in der Energiefunktion zu bestrafen, sollte eine Anmeldung eine Grenze überschreiten (ähnlich den Überschneidungen).
* Für die präzisere/eingeschränktere Programmvariante muss man die Grenzen in der Funktion \_\_findfreepos prüfen. Nur die freien Positionen werden zurückgegeben, die keine Grenze überschreiten. Damit wird die Positionierung (\_\_) und auch die Veränderung beim Optimieren (\_\_) abgedeckt, da diese Methoden jeweils auf \_\_ zurückgreifen.

Die Funktion kann man zum Beispiel einsetzen, wenn der Flohmarkt nicht einreihig ist, sondern aus mehreren unverbundenen Abschnitten besteht, deren Grenze nicht von einer Anmeldung überschritten werden darf (z.B. wegen physischem Abstand der Abschnitte oder aus Brandschutzgründen). Auch ein (Indoor-)Flohmarkt der sich über mehrere Räume erstreckt oder in Tischinseln eingeteilt ist (also die Tische nicht durchgehend verbunden sind) lässt sich auf diese Weise darstellen.

## Visualisierungen, Ergebnisausgabe

Die Ergebnisse werden (wenn bei der Abfrage in der Konsole entsprechend gewünscht wird) in csv Dateien unter „…\BwInf 39.2.1 Flohmarkt\BwInf 39.2.1 Flohmarkt\bin\Debug“ abgespeichert. Es gibt immer die Ergebnisse („results“), die Metadaten des Durchlaufs und eine Log-Datei, die, wenn simulated Annealing durchgeführt wird, jeden move mit seinen energetischen Auswirkungen enthält. Die Dateinamen setzen sich aus der Nummer des Datensets, der Art der gespeicherten Daten (results/meta/log), sowie dem Datum und der Uhrzeit zu der das Programm gestartet wurde, zusammen.

Zur Besseren Illustration des Ergebnisses habe ich zwei Erweiterungen geschrieben, die einmal die Energieentwicklung beim simulated Annealing und das Endergebnis auf einer Tabelle zeigt. Während die Option zum Plotten der Energieentwicklung am Ende in der Konsole in C# abgefragt wird, muss man für das Ergebnis im python script „visualizeResult“ den Namen der unter „…\BwInf 39.2.1 Flohmarkt\BwInf 39.2.1 Flohmarkt\bin\Debug“ zu findenden savedResults Datei angeben. Die Tabelle zeigt die Uhrzeiten von unten nach oben auf der y-Achse und die Tische auf der x-Achse. Im Bereich der negativen x-Werte werden die abgelehnten Anmeldungen in zufälliger Abfolge rot markiert angezeigt.

## Ergebnisanalyse (\_\_)

Die Funktion \_\_ wertet das Ergebnis unter mehreren Gesichtspunkten aus. Sie berechnet Werte für folgende Punkte:

* Anzahl der verschiedenen Anmeldungen in jeder Stunde (relevant für Statistik)
* Mieteinnahmen pro Stunde (relevant für Statistik)
* von Austellern benötigte Parkplätze pro Stunde   
  - Prämisse: je mehr Tische ein Aussteller mietet, mit desto mehr Autos kommt er (die Konstante Tische/Auto kann angepasst werden)
* neu ankommende Autos von Austellern pro Stunde  
  - relevant zum Abschätzen des Verkehrsaufkommens (ggf. Verkehrsordner oder Polizei nötig)  
  - Prämisse: je mehr Tische ein Aussteller mietet, mit desto mehr Autos kommt er (die Konstante Tische/Auto kann angepasst werden)
* erwartete Anzahl an Toilettengängen von Austellern pro Stunde:  
  - relevant für die Anzahl an Toilettenhäuschen die der Veranstalter mietet  
  - Prämisse: Austeller geht nach dem Ankommen und dann alle n Stunden auf die Toilette; n ist abhängig von Zielgruppe (bei Frauen niedrigeres n), verfügbarem Essen/Trinken, … und kann entsprechend manuell angepasst werden

## Lücken finden (\_\_)

Diese Funktion findet freie Positionen für Anmeldungen, an denen die eingegebenen Rahmenkriterien der Anmeldung erfüllt werden. Diese Rahmenkriterien sind: früheste Startzeit, späteste Endzeit, Mindestdauer, Maximaldauer, Mindestlänge, Maximallänge und die maximalen Kosten. Berechnet wird eine Liste, die alle möglichen Positionen für diese Anfrage enthält. Dabei werden alle verschiedenen Kombinationen der angegebenen Rahmenkriterien aus Startzeit, Dauer und Länge berücksichtigt.

So eine Funktion ist relevant, damit der Veranstalter für später hinzukommende Anmeldungen schnell alle möglichen Kombinationen aus Startzeit, Dauer und Länge abrufen und prüfen kann. Er kann zum Beispiel den Austellern der abgelehnten Anmeldungen die Möglichkeit geben, zu versuchen, mit veränderten, flexibleren Parametern einen Platz zu finden. So kann der Veranstalter einerseits seinen Gewinn maximieren als auch den abgelehnten Ausstellern doch noch eine Teilnahme ermöglichen.

Diese Funktion könnte man in eine GUI einbauen, wo man die passende Kombination aus Startzeit, Dauer, Länge und Position auswählen kann. Sie würde zu der Liste aller angenommenen Anmeldungen hinzugefügt werden und die Werte aus \_\_ könnten sich gleich aktualisieren. Das habe ich aber aus Zeitgründen nicht so ausführlich umgesetzt.

## Weitere, nicht-umgesetzte Erweiterungen

* Preisabstufungen in Bereichen: Es wäre denkbar, in Kombination mit dem Grenzen-feature, den Flohmarkt in verschieden teure Bereiche einzuteilen. Dann würde ich zuerst den teuersten Bereich befüllen mit den Anmeldungen, die bereit wären, einen höheren Preis zu zahlen und dann der Reihe nach die günstigeren Bereiche auffüllen.
* Mengenrabatt: Ein Mengenrabatt würde die Aufgabe bedeutend verkomplizieren, da dann eine sehr große Anmeldung weniger Einnahmen bringt als viele kleinere, die die gleiche Fläche bedecken. Dies wäre mit der direkten Positionierung nur schwer optimal umzusetzen und erfordert wahrscheinlich simulated Annealing.

## Laufzeit

# Ergebnisse

## Simulated Annealing

## setPositions

Für jedes Datenset habe ich die Konfiguration der Parameter gewählt, mit denen das beste Ergebnis erzielt wurde. Die anderen kann man gern im Code ausprobieren.