Bundeswettbewerb Informatik 39 Runde 2 Aufgabe 1

Flohmarkt

Dokumentation

Jakov David Wallbrecher
19.4.2021

Inhaltsverzeichnis

Lösungsidee

Meine erste Lösungsidee war es, allen Voranmeldungen zufällig eine Position zuzuteilen und daraufhin mit dem Optimierungsverfahren simulated Annealing die Lösung zu verbessern. Da ich aber auf diesem Weg keine ausreichend gute Lösungen gefunden habe, habe ich die initiale Positionierung sowie die möglichen Positionsveränderungen der Anmeldungen in mehreren Stufen eingeschränkt bzw. präzisiert. Dazu habe ich eine Heuristik entwickelt, die die Anmeldungen so positioniert, dass sehr gute Lösungen herauskommen. Diese Heuristik lässt sich mit mehreren Parametern anpassen. Je nach Problem führen andere Parameter zu den Besten Ergebnissen.

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 Programmvarianten, die ich in der Lösungsidee schon angeschnitten habe, wechseln zu können. Elementar ist das Tuple aus zwei Listen von angenommenen und abgelehnten Anmeldungen (`{accepted} ; {rejected}`). Jede Anmeldung (bzw. Vorregistrierung) ist Element der Klasse `Registration` und beinhaltet die Parameter `id` (Zeilennummer aus Datei), `rentStart` (Mietbeginn), `rentEnd` (Mietende), `rentDuration` (Mietdauer), `rentLength` (Anzahl der Tische) und `position`.

Positionierung		Simulated Annealing				Eigenschaften
		Energieberechnung		Veränderung		
`setRandomPositions()`	->	`energy()/energy2()`	->	`move()`		- Überschneidungen zugelassen
`setPositions(unsorted, randomPosition)`	->	`energy()/energy2()`	->	`move2()`		- Überschneidungen nicht möglich
`setPositions(sorted, bestPosition)`						- deterministische Positionierung

Programmvarianten (horizontal)

In der Main-Funktion kann man zuerst die Rahmenbedingungen, wie z.B. 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 (siehe Erweiterung `validateData()`). Nun wird ein Objekt des Typs `solver` erstellt, einer Klasse, die für die gesamte Verarbeitung der Daten (inklusive der Problemlösung) zuständig ist. Danach kann eine Methode gewählt werden, mit der die Anmeldungen auf die Positionen verteilt werden. Es gibt einmal den randomisierten Ansatz `setRandomPositions()` und eine durch mitgegebene Argumente auf Wunsch angepassten besseren Ansatz `setPositions()`. Jetzt kann das Optimieren durch simulated Annealing stattfinden. Auch hier gibt es verschiedene Varianten auf die ich später noch ausführlich eingehe. Zum Schluss werden die Ergebnisse ausgegeben und

auf Wunsch gespeichert. Außerdem können die Erweiterungen `analyseResults()` und `findFreePositionsInRange()` 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

setRandomPositions()

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

setPositions()

```
public void setPositions(int sorted, bool optimalPos) {
            (List<Registration> accepted, List<Registration> rejected) registrationsLoc = cloneLists(registrations);
            registrations.accepted.Clear(); registrations.rejected.Clear();

            //sortiere registrations nach Sperrigkeit, wenn gewünscht
            if (sorted!=0) {
                if (sorted==1) { registrationsLoc.accepted.Sort(compareByRent); }
                else if(sorted==2){ registrationsLoc.accepted.Sort(compareByDuration); }
                else if(sorted == 3) { registrationsLoc.accepted.Sort(compareByLength); ; }
            }
            //setze Positionen der Anmeldungen
            foreach (Registration reg in registrationsLoc.accepted) {
                List<int> freePositions = findFreePositions(unoccupiedFields, reg); //alle Positionen auf denen keine
Überschneidungen auftreten
                if (freePositions.Count > 0) {
                    //wenn gewünscht optimale Position, sonst zufällige
                    if (optimalPos) {
                        reg.position = findBestPosition(unoccupiedFields, reg, freePositions)[0];
                    }
                    else { reg.position = freePositions[rnd.Next(freePositions.Count)]; }
                    registrations.accepted.Add(reg);
                    unoccupiedFields = setRegUnoccupiedFields(unoccupiedFields, reg, false); //unoccupiedFields updaten
                }
                else {
                    reg.position = -1;
                    registrations.rejected.Add(reg);
                }
            }
            output = (energy3(registrations.accepted), new List<int>(), new List<int>(), registrations, registrations);
        }
```

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 (`compareByRent`) oder anhand der Mietdauer (`compareByDuration`). 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.

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 `findFreePositions()` 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 `findBestPosition()`). Sollte keine Position mehr frei sein, wird die Anmeldung auf die Liste der abgelehnten Anmeldungen gesetzt.

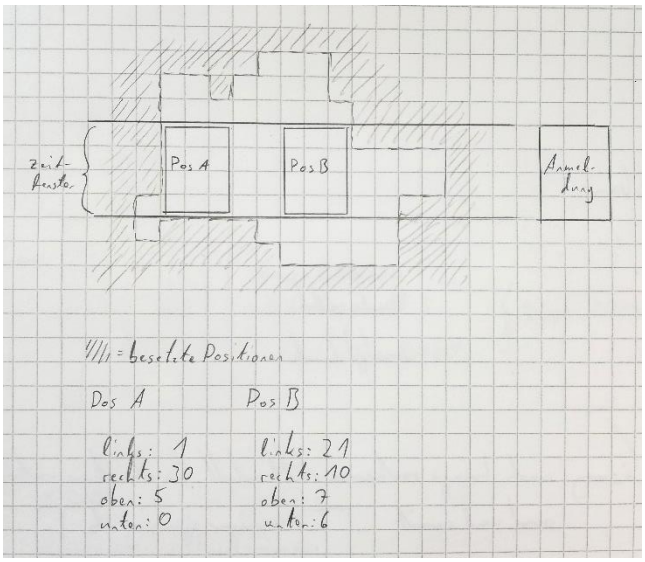
Das zweidimensionale Boolean Array `unoccupiedFields` dient primär der Laufzeitverkürzung der Funktion `findFreePositions()`. Da in dem Array für jeden Tisch zu jeder Stunde abgespeichert ist, ob er belegt ist, muss die Funktion nur durch das Array gehen und nicht zusätzlich bei allen Anmeldungen überprüfen ob sie in die fragliche Zeit fallen.

findFreePositions()

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, bzw. ob sie an dieser Position negative Felder/Einträge von `unoccupiedFields` überschneidet.

findBestPosition()

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. Die Fläche an jeder Seite wird von der Funktion `getSpaceAround()` berechnet.

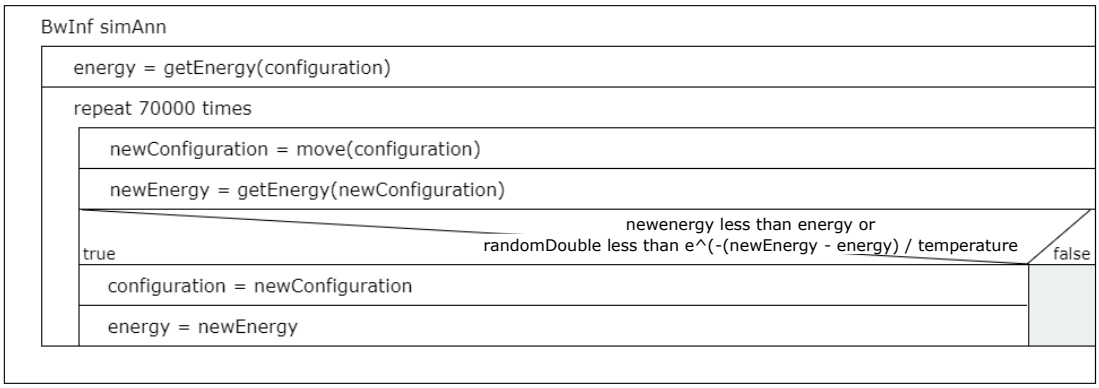
Im Bild sieht man eine Tisch-Zeit Tabelle (ein Kästchen entspricht einer Zeiteinheit x x ein Tisch), auf der zwei Positionen einer Anmeldung miteinander verglichen werden. Der

schraffierte Bereich kennzeichnet dabei die Felder, die schon von anderen Anmeldungen belegt sind. Intuitiv würden die meisten Menschen die Position A bevorzugen. Ich habe hier eine Methode entwickelt, die diese intuitive Wahl auf den Computer überträgt. Position A hat im Vergleich zu Position B mehr Kontaktfläche, bzw. möglichst viel freie Fläche an einer Seite statt auf beide Seiten verteilt. Der vorher beschriebene Term ist bei Position A 1, während er bei Position B 16 ergibt.

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 Veränderung diese Bedingung enthalten (siehe `move()`), während die Variante ohne Überschneidung eine andere Funktion braucht (siehe `move2()`). Die Energieberechnungen (`energy()` und `energy2()`) sind jeweils für beide Varianten einsetzbar. Grundsätzlich ist `energy` aber eher auf simulated Annealing ausgelegt, daher werde ich es zusammen mit `move()` erklären.

Energieberechnung und Veränderung (`energy()` und `move()`)

`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 `move()` 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“.

Energieberechnung und Veränderung Variante (`energy2()` und `move2()`)

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

Auch beim `move2` 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 `findFreePositions()` alle Positionen für die Anfrage, an denen keine Überschneidung auftreten würde. Im Gegensatz zur Positionierung am Anfang (`setPositions()`) wird nicht die optimale Position ausgewählt, sondern eine Zufällige (simulated Annealing ist nicht-deterministisch, weshalb nicht bei jeder Veränderung automatisch die beste Position ausgewählt werden soll).

Erweiterungen der Aufgabenstellung

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 (`validateData()`)

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.

Beispiel:

Eingabe:

```
3
13 9 20
7 11 5
12 18 1002
```

Ausgabe:

```
invalid data (rent start must be smaller than rent end) at line 1
  rentStart=13, rentEnd=9
invalid data (rent starts to early) at line 2
  rentStart=7, earliestStart=8
invalid data (length to big) at line 3
  länge=1002, streetLength=1000
remove invalid data (r) or exit program (e)?
```

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:

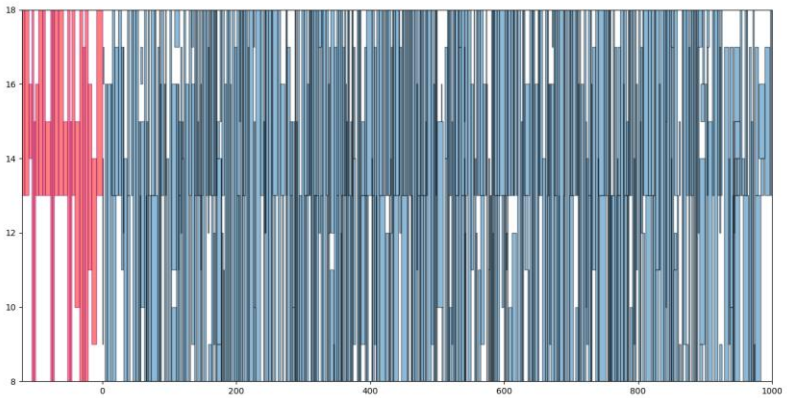
- für die erste Programmvariante mit mehr Freiheiten (zufällige Positionierung; Überschneidungen werden über Energiefunktion minimiert) reicht es, in der Energiefunktion zu bestrafen, sollte eine Anmeldung eine Grenze überschreiten (ähnlich den Überschneidungen).
- Für die präzisere/eingeschränktere Programmvariante muss man die Grenzen in der Funktion `findFreePositions` prüfen. Nur die freien Positionen werden zurückgegeben, die keine Grenze überschreiten. Damit wird die Positionierung (`setPositions`) und auch die Veränderung beim Optimieren (`move2`) abgedeckt, da diese Methoden jeweils auf `findFreePositions` 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.

Ergebnisausgabe, Visualisierungen

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 (`analyseResults`)

Die Funktion `analyseResults` 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

Beispiel (Datensatz 1)

```
double carThreshold = 5; //Anzahl an Tischen auf die ein Auto kommt
int hoursBetweenToiletUse = 2; //durchschnittliche Stundenzahl zwischen zwei WC-
Besuchen eines Austellers

ANALYSE RESULT:
  REGISTRATIONS PER HOUR:
        8       9       10      11      12      13      14      15      16      17
        252     261     261     269     271     344     324     298     261     231
  PARKING SPOTS PER HOUR:
        8       9       10      11      12      13      14      15      16      17
        274     286     286     296     297     377     356     326     282     249
  NEW CARS PER HOUR:
        8       9       10      11      12      13      14      15      16      17
        274     21      11      17      15      174     9       9       3       10
  TOILET USES PER HOUR:
        8       9       10      11      12      13      14      15      16      17
        252     17      261     32      276     188     283     195     286     204
  EARNINGS PER HOUR:
        8       9       10      11      12      13      14      15      16      17
        718     754     755     782     780     995     954     869     752     669
```

Lücken finden (`findFreePositionsInRange`)

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 und andererseits 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 `unoccupiedFields` könnten sich gleich aktualisieren. Ich habe mich allerdings auf die Programmlogik dahinter konzentriert und keine GUI implementiert.

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

Nun will ich die (maximale) Laufzeit des Programms analysieren. Da sie von vielen Parametern abhängt (Anzahl an Anmeldungen, durchschnittliche Dauer und Länge einer Anmeldung, …) werde ich die einzelnen Programmteile gesondert analysieren und jeweils erst am Schluss die O-Notation vereinfachen. Um besser die tatsächliche Laufzeit abschätzen zu können, habe Ich die Laufzeit der komplexeren Funktionen erst allgemein formuliert (mit Verweisen auf andere Funktionen), dann die Laufzeiten der verwiesenen Funktionen eingesetzt und vereinfacht, und erst zum Schluss für Variablen deren Obergrenzen eingesetzt.

Grundsätzlich lässt sich aber feststellen, dass die Positionierung (je nach Funktion: `setPos(unsorted, randomPos)` / `setPos(unsorted, optimalPos)` / `setPos(sorted, optimalPos)`) eine Laufzeitobergrenze von $O(n)$ beziehungsweise $O(n^2)$ hat. Dabei erzielen die Positionierungen mit höherer Laufzeit auch bessere Ergebnisse. Die Laufzeitobergrenze des simulated Annealing Optimierungsalgorithmus liegt bei $O(n^2)$. Allerdings lässt sich dabei nur bedingt die faktische Dauer eines Programms ableiten, da in der faktischen Laufzeit große Multiplikatoren dazukommen und n bei „realistischen“ Anmeldungswerten niemals groß genug wird um größeren Einfluss auf die Laufzeit zu haben als diese Multiplikatoren.

Begriffsklärung:

`registrationNum` : Anzahl der Anmeldungen : n

`avRegDuration` : durchschnittliche Dauer einer Anmeldung

`avRegLength` : durchschnittliche Länge einer Anmeldung

findClosestValue(list)	log(list.count)
getSpaceAround()	avRegDuration* streetLength + avRegLength* duration
findFreePositions()	streetLength* (borderNum + avRegLength* avRegDuration)
findBestPosition()	freePositions()* getSpaceAround()
sumOverlap()	registrationNum^2
borderNum, freePositions	Worst case: streetLength

setRandomPositions()	registrationNum + registrationNum^2
setPos	
setPos(unsorted, randomPos)	registrationNum + registrationNum* (findFreePositions() + avRegDuration* avRegLength) = registrationNum* streetLength* (borderNum + avRegLength* avRegDuration) = registrationNum* streetLength^2* avRegDuration = registrationNum* 10^7
setPos(unsorted, bestPos)	registrationNum + registrationNum* (findFreePositions() + findBestPos() + avRegDuration* avRegLength) = registrationNum* streetLength* (borderNum + avRegLength* avRegDuration + freePositions* duration) = registrationNum* streetLength^2* duration* 3 = registrationNum* 30^7
setPos(sorted, bestPos)	registrationNum + registrationNum* log(registrationNum) + registrationNum* (findFreePositions() + findBestPos() + avRegDuration* avRegLength) = registrationNum* log(registrationNum) + registrationNum* streetLength^2* duration* 3 = registrationNum* log(registrationNum) + registrationNum* 30^7
Simulate(allg)	registrationNum+energy() + runs* (move() + energy() + cloneList() + sumOverlap()) = runs* (move() + 2* registrationNum^2 + registrationNum* streetLength)
Simulate(move2())	runs* (streetLength^2 + streetLength* avRegLength* avRegDuration + 2* registrationNum^2) = runs* (10^6 + 10^7 + registrationNum^2)
Simulate(move())	runs* (2* registrationNum^2 + registrationNum* streetLength) = runs* (registrationNum^2 + registrationNum*10^3)
energy()	registrationNum + sumOverlap() + registrationNum* borders
move	
move()	registration.Count
move2()	findFreePositions() + avRegLength* avRegDuration + log(freePositions.count) + registrationNum = streetLength^2 + streetLength* avRegLength* avRegDuration + log(streetLength) + registrationNum

Ergebnisse

Hier möchte ich nur auf die Ergebnisse selbst eingehen. Für jedes Ergebnis steht die Anzahl der abgelehnten Anmeldungen und die daraus resultierenden Mieteinnahmen. Eine volle Programmausgabe ist im Kapitel „Programmausgabe“.

Simulated Annealing

Alle hier durchgeführten simulated-Annealing Vorgänge entsprechen der ersten Programmvariante. Also werden grundsätzlich Überschneidungen zwischen Anmeldungen zugelassen, die allerdings durch die Energiefunktion bestraft werden.

Die initiale Positionierung hat bei allen Durchläufen die Funktion `setPositions()` übernommen. Dabei wurden zufällige 20 Prozent der Anmeldungen auf die abgelehnt-Liste gesetzt.

Veränderungen wurden von der Funktion `move()` und Energieberechnung von `energy2()` übernommen.

Datensatz 1

```
SIMANN META
Anzahl Durchläufe: 70000
Starttemperatur: 25
Verkleinerungsrate: 0,99995

BESTE VERTEILUNG
Anzahl Anmeldungen: 490
   davon abgelehnt: 24
Anzahl Überschneidungen: 0
Energie/-Mieteinnahmen: -7365
```

Datensatz 2

```
SIMANN META
Anzahl Durchläufe: 70000
Starttemperatur: 25
Verkleinerungsrate: 0,99995

BESTE VERTEILUNG
Anzahl Anmeldungen: 603
   davon abgelehnt: 104
Anzahl Überschneidungen: 0
Energie/-Mieteinnahmen: -7885
```

Datensatz 3

```
SIMANN META
Anzahl Durchläufe: 70000
Starttemperatur: 25
Verkleinerungsrate: 0,99995

BESTE VERTEILUNG
Anzahl Anmeldungen: 735
   davon abgelehnt: 191
Anzahl Überschneidungen: 0
Energie/-Mieteinnahmen: -6594
```

Datensatz 4

```
SIMANN META
Anzahl Durchläufe: 70000
Starttemperatur: 25
Verkleinerungsrate: 0,99995

BESTE VERTEILUNG
Anzahl Anmeldungen: 7
   davon abgelehnt: 2
Anzahl Überschneidungen: 0
Energie/-Mieteinnahmen: -6068
```

Datensatz 5

```
SIMANN META
Anzahl Durchläufe: 70000
Starttemperatur: 25
Verkleinerungsrate: 0,99995

BESTE VERTEILUNG
Anzahl Anmeldungen: 25
   davon abgelehnt: 17
Anzahl Überschneidungen: 0
Energie/-Mieteinnahmen: -6353
```

Datensatz 6

```
SIMANN META
Anzahl Durchläufe: 70000
Starttemperatur: 350
Verkleinerungsrate: 0,99999

BESTE VERTEILUNG
Anzahl Anmeldungen: 9
   davon abgelehnt: 3
Anzahl Überschneidungen: 0
Energie/-Mieteinnahmen: -7883
```

Datensatz 7 SIMANN META Anzahl Durchläufe: 70000 Starttemperatur: 25 Verkleinerungsrate: 0,99995 BESTE VERTEILUNG Anzahl Anmeldungen: 566 davon abgelehnt: 90 Anzahl Überschneidungen: 0 **Energie/-Mieteinnahmen: -7957**	

setPositions

Für jedes Datenset habe ich die Konfiguration der Parameter gewählt, mit denen das beste Ergebnis erzielt wurde. Grundsätzlich sind die hier geführten Ergebnisse aber alle aus der Programmvariante 3 (deterministisch).

Datensatz 1

setPositions (sorted by compareByRent; optimalPosition)
Anzahl Anmeldungen: 490
davon abgelehnt: 0
Anzahl Überschneidungen: 0
Energie/-Mieteinnahmen: -8028

Datensatz 2

setPositions (sorted by compareByDuration; optimalPosition)
Anzahl Anmeldungen: 603
davon abgelehnt: 113
Anzahl Überschneidungen: 0
Energie/-Mieteinnahmen: -9069

Datensatz 3

setPositions (sorted by compareByDuration; optimalPosition)
Anzahl Anmeldungen: 735
davon abgelehnt: 154
Anzahl Überschneidungen: 0
Energie/-Mieteinnahmen: -8776

Datensatz 4

setPositions (sorted by compareByDuration/compareByRent; optimalPosition)
Anzahl Anmeldungen: 7
davon abgelehnt: 2
Anzahl Überschneidungen: 0
Energie/-Mieteinnahmen: -7370

Datensatz 5

setPositions (sorted by compareByRent; optimalPosition)
Anzahl Anmeldungen: 25
davon abgelehnt: 21
Anzahl Überschneidungen: 0
Energie/-Mieteinnahmen: -6651

Datensatz 6

setPositions (sorted by compareByDuration/compareByRent; optimalPosition)

```
Anzahl Anmeldungen: 9
   davon abgelehnt: 0
Anzahl Überschneidungen: 0
Energie/-Mieteinnahmen: -10000
```

Datensatz 7

```
setPositions (sorted by compareByDuration; optimalPosition)
Anzahl Anmeldungen: 566
   davon abgelehnt: 34
Anzahl Überschneidungen: 0
Energie/-Mieteinnahmen: -9650
```

Code

Im Folgenden möchte ich die wichtigsten Codeblöcke abdrucken:

```
/// <summary>
/// setzt zufällige Positionen, auch auf rejected Liste
/// </summary>
/// <param name="percentageRejected">anteil in Prozent der auf die Warteliste gesetzten</param>
public void setRandomPositions(int percentageRejected) {
    foreach (Registration a in registrations.accepted) {
        a.position = rnd.Next(streetLength - a.rentLength + 1);
    }
    for (int i = 0; i < (((float)percentageRejected / 100) * (registrations.accepted.Count +
registrations.rejected.Count)); i++) {
        int rnd1 = rnd.Next(registrations.accepted.Count);
        registrations.rejected.Add(registrations.accepted[rnd1]);
        registrations.accepted.RemoveAt(rnd1);
    }
    metaToSave.Add("Positioning: setRandomPositions2(" + percentageRejected + ")");
}

/// <summary>
/// setzt Positionen, auch auf rejected Liste; keine Überschneidungen zugelassen; nach Wunsch: - fängt
bei sperrigen registrations an (nach Wunsch gilt A oder Dauer als sperrig)  - immer an Position, die am
wenigsten freie Positionen einschließt
/// </summary>
/// <param name="sorted">0: nicht sortiert; 1: sorted nach Miete; 2: sortiert nach Dauer, 3: sortiert
nach Länge</param>
/// <param name="optimalPos">true: optimale Position (findBestPosition); false: zufällige
Position</param>
public void setPositions(int sorted, bool optimalPos) {
    (List<Registration> accepted, List<Registration> rejected) registrationsLoc =
cloneLists(registrations);
    registrations.accepted.Clear(); registrations.rejected.Clear();

    //sortiere registrations nach Sperrigkeit, wenn gewünscht
    if (sorted!=0) {
        if (sorted==1) { registrationsLoc.accepted.Sort(compareByRent); }
        else if(sorted==2){ registrationsLoc.accepted.Sort(compareByDuration); }
        else if(sorted == 3) { registrationsLoc.accepted.Sort(compareByLength); ; }
    }
    //setze Positionen der Anmeldungen
    foreach (Registration reg in registrationsLoc.accepted) {
        List<int> freePositions = findFreePositions(unoccupiedFields, reg); //alle Positionen auf denen
keine Überschneidungen auftreten
        if (freePositions.Count > 0) {
            //wenn gewünscht optimale Position, sonst zufällige
            if (optimalPos) {
                reg.position = findBestPosition(unoccupiedFields, reg, freePositions)[0];
            }
            else { reg.position = freePositions[rnd.Next(freePositions.Count)]; }
            registrations.accepted.Add(reg);
            unoccupiedFields = setRegUnoccupiedFields(unoccupiedFields, reg, false); //unoccupiedFields
updaten
        }
        else {
            reg.position = -1;
            registrations.rejected.Add(reg);
        }
    }
```

```
    output = (energy3(registrations.accepted), new List<int>(), new List<int>(), registrations,
registrations);
    metaToSave.Add("Positioning: setPositions (" + (sorted!=0 ? "sorted " + (sorted==1 ? "by
compareByRent; " : (sorted == 2 ? "by compareByDuration; " : "by compareByLength; ")) : "not Sorted; ")
+ (optimalPos ? "optimalPosition)" : "randomPosition)"));
}



public void simulate() {
    double temp = startTemperature;
    int currentEnergy = energyType.del(registrations.accepted, startTemperature);//variabel
    List<int> energies = new List<int>();//Datenliste für log/meta Datei
    List<int> overlaps = new List<int>();//Datenliste für log/meta Datei
    (List<Registration> accepted, List<Registration> rejected) bestDistribution;
bestDistribution.accepted = new List<Registration>(); bestDistribution.rejected = new
List<Registration>();
    (List<Registration> accepted, List<Registration> rejected) currentRegistrations;
currentRegistrations.accepted = new List<Registration>(); currentRegistrations.rejected = new
List<Registration>();
    (List<Registration> accepted, List<Registration> rejected) changedRegistrations;
changedRegistrations.accepted = new List<Registration>(); changedRegistrations.rejected = new
List<Registration>();
    changedRegistrations = cloneLists(registrations); currentRegistrations = cloneLists(registrations);
    int bestEnergy = energy3(currentRegistrations.accepted);
    bestDistribution = currentRegistrations;

    for (int i = 0; i < runs; i++) {
        //Veränderung machen und Energie berechnen
        changedRegistrations = moveType.del(changedRegistrations, temp); //variabel
        int newEnergy = energyType.del(changedRegistrations.accepted, temp);//variabel
        logToSave[logToSave.Count - 1] += " " + newEnergy;
        if (i % 100 == 0) Console.WriteLine(i + "  " + currentEnergy);
        //überprüfen ob Änderung angenommen wird
        if (newEnergy <= currentEnergy || rnd.NextDouble() < Math.Exp(-(newEnergy - currentEnergy) /
temp)) {
            currentEnergy = newEnergy;
            logToSave[logToSave.Count - 1] += " " + sumOverlap(changedRegistrations.accepted).number;
logToSave.Add("");
            currentRegistrations = cloneLists(changedRegistrations);
            //bestDistribution neu festlegen
            if (sumOverlap(currentRegistrations.accepted).number == 0 &&
energy3(currentRegistrations.accepted) < bestEnergy) {
                bestEnergy = energy3(currentRegistrations.accepted);
                bestDistribution = cloneLists(changedRegistrations);
            }
        }
        else {
            changedRegistrations = cloneLists(currentRegistrations);
        }
        energies.Add(energy3(currentRegistrations.accepted));
        overlaps.Add(sumOverlap(currentRegistrations.accepted).number);
        //temperature verkleinern
        temp *= tempDecreaseRate;
    }

    Console.WriteLine("done");

    output = (bestEnergy, energies, overlaps, currentRegistrations, bestDistribution);
    registrations = cloneLists(currentRegistrations);
}
```

```
/// <summary>
///verschieben und swappen ohne Überschneidungen (nur an Position wo sicher keine Überschneidungen
auftreten)
/// </summary>
public (List<Registration> accepted, List<Registration> rejected) move2((List<Registration> accepted,
List<Registration> rejected) registrations, double temp) {
    int rnd1 = rnd.Next(100);
    if (rnd1 < 50 && registrations.accepted.Count > 0) { //verschiebe
        int index = rnd.Next(registrations.accepted.Count());
        int x = rnd.Next(streetLength - registrations.accepted[index].rentLength + 1) -
registrations.accepted[index].position;
        int move = x;
        move = (int)(move * (temp / startTemperature));//kleinere Veränderungsschritte bei sinkender
Temperatur
        move = (move == 0) ? ((x > 0) ? +1 : -1) : move;

        List<int> freePositions = findFreePositions(unoccupiedFields, registrations.accepted[index]);
        if (freePositions.Count > 0) {
            unoccupiedFields = setRegUnoccupiedFields(unoccupiedFields, registrations.accepted[index],
true);
            //zufälliger Wert möglicherweise nicht in freePositions; daher wird nächster Wert von
findclosestVal gesucht
            registrations.accepted[index].position =
freePositions[findClosestValue(registrations.accepted[index].position + move, freePositions)];
            unoccupiedFields = setRegUnoccupiedFields(unoccupiedFields, registrations.accepted[index],
false);
        }
        logToSave.Add("verschiebe");
    }
    else {// swappe
        int rnd2 = rnd.Next(100);
        if ((rnd2 < 50 || registrations.rejected.Count == 0) && registrations.accepted.Count > 0)
{//accepted->rejected
            int index = rnd.Next(registrations.accepted.Count);
            registrations.rejected.Add(registrations.accepted[index]);
            unoccupiedFields = setRegUnoccupiedFields(unoccupiedFields, registrations.accepted[index],
true);
            registrations.accepted.RemoveAt(index);
            logToSave.Add("swap->rejected");
        }
        else {//rejected->accepted
            int index = rnd.Next(registrations.rejected.Count);
            List<int> freePositions = findFreePositions(unoccupiedFields,
registrations.rejected[index]);
            if (freePositions.Count > 0) {
                registrations.rejected[index].position = freePositions[rnd.Next(freePositions.Count)];
                registrations.accepted.Add(registrations.rejected[index]);
                unoccupiedFields = setRegUnoccupiedFields(unoccupiedFields,
registrations.rejected[index], false);
                registrations.rejected.RemoveAt(index);
                logToSave.Add("swap->accepted");
            }
        }
    }
    return (registrations.accepted, registrations.rejected);
}

/// <summary>
///  berechnet energy:
///  - (Miete aller Anmeldungen die sich nicht überschneiden)
/// </summary>
public int energy2(List<Registration> registrationsLoc, double temperature = 0) {
    int energy = 0;
    foreach (Registration reg in registrationsLoc) {
        if (checkIfOverlap(reg, registrationsLoc) == false) {
            energy -= reg.rentLength * reg.rentDuration;
        }
        foreach (int border in borderPos) { //Borderüberschreitungen prüfen und bestrafen
            if (reg.position < border && reg.position + reg.rentLength > border) {
                energy += 2 * reg.rentLength * reg.rentDuration;
            }
        }
    }
    return energy;
}
```

```
/// <summary>
/// findet alle möglichen Positionen für eine gegebene Anmeldung, die keine Grenze überschreitet und an
denen keine Überschneidung auftritt auf Basis von occupiedFields[]
/// </summary>
/// <param name="reg">Anmeldung</param>
/// <param name="unoccupiedFieldsLoc">2D Array das die Ort-Zeit-tafel darstellt. true: frei; false:
besetzt</param>
/// <returns>Liste mit allen Positionen im Flohmarkt bei denen für die Anmeldung keine Überschneidung
auftritt</returns>
private List<int> findFreePositions(bool[,] unoccupiedFieldsLoc, Registration reg) {
    List<int> positions = new List<int>();
    for (int x = 0; x < streetLength - reg.rentLength; x++) {
        bool crossBorder = false;
        foreach (int border in borderPos) { //checkt ob aktuelle Position borders überschneiden würde
(Erweiterung)
            if (x < border && x + reg.rentLength > border) { crossBorder = true; break; }
        }
        if (crossBorder == false) {
            //geht alle Tisch|Uhrzeit Felder durch, die an dieser Position besetzt sein würden; wenn
eines dieser Felder schon besetzt wird, ist die Position nicht möglich ohne Überschneidungen
            bool horizontalPosition = true;
            for (int j = 0; j < reg.rentLength; j++) {
                bool vertikalPosition = true;
                for (int y = reg.rentStart - starttime; y < reg.rentEnd - starttime; y++) {
                    if (unoccupiedFieldsLoc[x + j, y] == false) { vertikalPosition = false; break; }
                }
                if (vertikalPosition == false) { horizontalPosition = false; x += j; break; }
            }
            if (horizontalPosition) {
                positions.Add(x);
            }
        }
    }
    return positions;
}



/// <summary>
/// findet beste Position für gegebene Anmeldung in einer Liste von freien Positionen
/// überprüft für jede freie Position, wie nah ihr Abstand zu den nächsten Anmeldungen ist
/// </summary>
private List<int> findBestPosition(bool[,] unoccupiedFieldsLoc, Registration reg, List<int>
freePositions) {
    (int smallestArea, List<int> positions) best = (int.MaxValue, new List<int>() { -2 });
    for (int i = 0; i < freePositions.Count; i++) {
        (int left, int right, int above, int below) = getSpaceAround(unoccupiedFieldsLoc, reg,
freePositions[i]);
        int area = Math.Min(left, right) + Math.Min(above, below); //Summe von (kleinste Außenfläche
rechts links) und (kleinste Außenfläche oben unten)
        if (area < best.smallestArea) {
            best = (area, new List<int>() { freePositions[i] });
        }
        else if (area == best.smallestArea) {
            best.positions.Add(freePositions[i]);
        }
    }
    return best.positions;
}
```