ATM
informatiCup Aufgabe 3

Kai Fabian
Hasso-Plattner-Institut, IT-Systems Engineering, 1. Semester
Kai.Fabian@student.hpi.uni-potsdam.de

Stefan Hansch
FH Regensburg, Medizinische Informatik, 1. Semester
Stefan.Hansch@stud.hs-regensburg.de

Dominik Moritz
Hasso-Plattner-Institut, IT-Systems Engineering, 1. Semester
Dominik.Moritz@student.hpi.uni-potsdam.de

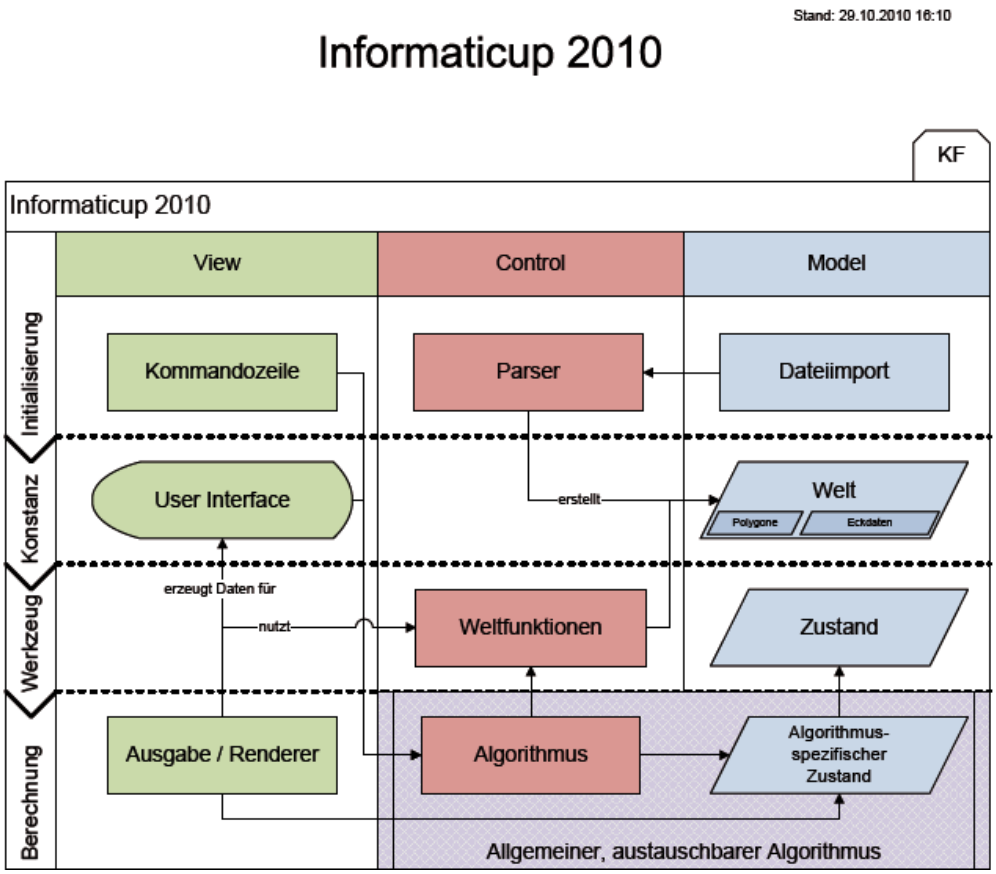
Matthias Springer
Hasso-Plattner-Institut, IT-Systems Engineering, 1. Semester
Matthias.Springer@student.hpi.uni-potsdam.de

20. März 2011

Inhaltsverzeichnis

1 Einführung

Vor der Planung des eigentlichen Programmsablauf, der Entwicklung der Algorithmen und der Implementierung des Programms und der GUI haben wir uns grundlegende Gedanken zur Aufteilung des Programms gemacht und folgendes Modell entwickelt.



Dieses Modell entspricht nicht mehr dem aktuellem Stand, war jedoch Anfangs als ein grobes Konzept sehr hilfreich.
Zur Umsetzung unserer Ideen benutzten wir die NetBeans IDE unter Verwendung der Programmiersprache Java unter Windows. Unser Projekt ist sehr modular aufgebaut und in verschiedene Packages und Klassen unterteilt. Diese modulare und objektorientierte Programmentwicklung hatte insbesondere folgende Vorteile.

- **Teamfähigkeit:** Verschiedene Mitglieder eines Teams können gleichzeitig an verschiedenen Klassen/Teilen des Projekts arbeiten. Dies war besonders wichtig, weil drei Personen in Potsdam und eine Person in Regensburg am Projekt arbeitete. Als Versionsverwaltung setzten wir Subversion ein.
- **Leichte Austauschbarkeit von Quelltext:** Änderungen, Verbesserungen und neue Algorithmen können ohne großen Aufwand implementiert werden.

Die Dokumentation wurde mit LaTeX (TeX Live 2010) erstellt.

2 Installationsanleitung

Wie in der Dokumentation beschrieben, setzten wir die Aufgabe in Java mithilfe der NetBeans IDE um. Mit NetBeans ist es sehr einfach möglich, den Code zu untersuchen und das Programm neu zu kompillieren.

Um das Programm zu starten, ist aber nur die Java Runtime Environment (JRE) oder eine vergleichbare Java VM erforderlich. Um das Programm zu starten, genügt es, die Datei `informaticup.jar` im Ordner `Programm` auszuführen. Falls das Programm so nicht gestartet werden kann (wegen einer falsch gesetzten `PATH` Variable), lässt sich das Programm auch mit dem Befehl `java -jar ‘‘informaticup.jar‘‘` starten. Wichtig ist, dass der Ordner `lib` mit den Bibliotheken immer neben der Programmdatei liegt, damit die Bibliotheken geladen werden können.

3 Bedienungsanleitung

Wir schrieben eine benutzerfreundliche, möglichst selbsterklärende Benutzeroberfläche für unser Programm. Die wichtigsten Bedienelemente sind im folgendem dokumentiert.

Im Menüpunkt Berechnung gibt es zwei Unterpunkte. Der eine startet eine komplett neue Berechnung, der andere bearbeitet eine bereits bestehende Berechnung. Beide bestehen weitgehend aus denselben Komponenten.

Wir wählen „Berechnung“ → Untermenü „Neue Berechnung“ → Es öffnet sich ein neues Fenster. Mit „Lade Datei“ kann die entsprechende Eingabedatei ausgewählt werden. Ist die Karte ausgewählt, hat der Benutzer weitere Einstellungsmöglichkeiten.

- Auf der rechten Seite finden sich die einzelnen Attribute dieser Karte. Der Nutzer kann diese bei jeder neuen Berechnung verändern. Somit können die verschiedenen Einzelattribute, die in den Eingabedatei stehen, unterschiedlich stark gewichtet werden.

- Mittig unter „keine Berechnung“ kann die Karte ohne Automaten in unser Hauptfenster eingezeichnet werden. Gewichtskarte bedeutet, dass die Karte die entsprechend gewählten Attribute bereits umsetzt.

- Auf der linken Seite, kann der Algorithmus ausgewählt werden, mit dem die Automaten auf dieser Karte gesetzt werden. Es gibt zwei Arten von Algorithmen: Die Eröffnungsalgorithmen sowie die Optimierungsalgorithmen. (Unterschiede siehe Algorithmendokumentation) Bei den Optimierungsalgorithmen, kann die Suchintensität eingestellt werden. Ab

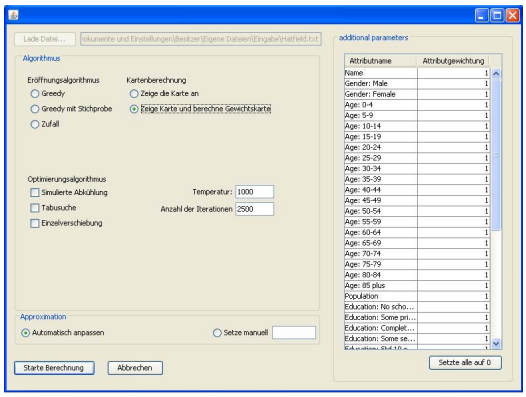
einer genügend hohen Suchintensität, empfiehlt es sich, den Eröffnungsalgorithmus „Zufall“ zu wählen, da der vorherrgehende Algorithmus keinen signifikanten Einfluss mehr auf die Güte der Lösung hat. Falls andere als die vorgegebenen oder von uns erzeugten Beispiele verwendet werden (insbesondere bei einer sehr großen Anzahl an Automaten), kann es dennoch sinnvoll sein, den Greedy-Algorithmus zu verwenden.

- Im unteren Teil kann die Approximationsrate noch verändert werden. Die Approximationsrate gibt an, wie stark die Karte verkleinert wird, damit ggf. schnellere Berechnungen erfolgen können. Zur Auswahl steht die manuelle oder automatische Auswahl der Approximationsrate. Wird sie automatisch gesetzt, so wird diese in Abhängigkeit des Arbeitsspeichers des entsprechden PCs gewählt. Durch die Auswahl per Hand kann je nach den entsprechenden Bedürfnissen eine höhere oder niedrigere Approximationsrate gewählt werden.

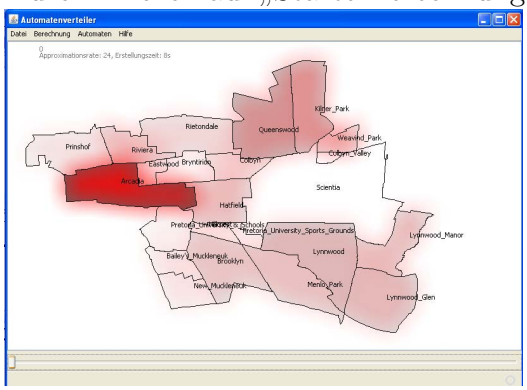
Ist die Auswahl getroffen, so wird nach erfolgreicher Eingabe wieder ins Hauptfenster gewechselt. Die angezeigte Karte kann vergrößert oder verkleinert werden. Vielmehr können die Automaten auch gänzlich entfernt werden oder zusätzliche Automaten über den Menüpunkt „Automaten“ gesetzt werden. Will man nicht alle Automaten automatisch setzen lassen, so müssen diese entsprechend markiert werden. Wählt man nun „Berechnung“ bearbeiten, so können die Standpunkte der restlichen Automaten (wie bereits zuvor erwähnt) optimiert werden. Jeder Zustand (wie die Automaten stehen und welches Gewicht sie haben) kann unter dem entsprechenden Menüpunkt gespeichert werden.

3.1 Eine Beispielberechnung

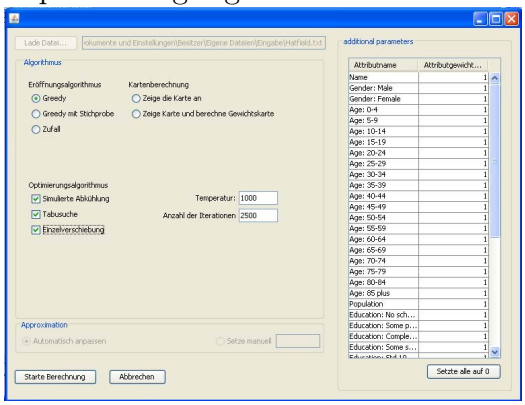
Programm Starten → „Berechnung“ → „Neue Berechnung“ → „Lade Datei“. Hier wählen wir Hatfield.txt aus. Wir belassen die Attribute alle auf 1, lassen die Approximationsrate automatisch setzen und wählen „Zeige Pixelkarte und berechne Gewichtskarte“ aus.



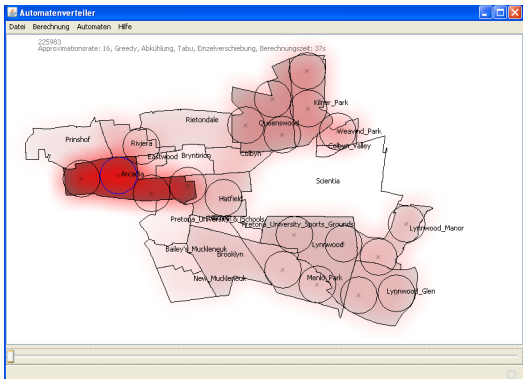
Durch Klicken auf „Starte Berechnung“ erscheint in etwa folgende Karte.



Nun setzen wir über „Automaten“ → „Neuer Automat“ einen neuen Automaten. Dieser ist fest verankert, kann aber durch entsprechende Menüauswahl entsperrt werden. Wir führen nun „Berechnung“ → „Berechnung bearbeiten “ aus. Hier wählen wir den Greedy-Algorithmus aus und zusätzlich alle Optimierungsalgorithmen.



Sieht dann folgendermaßen aus.



Zuletzt speichern wir das Ganze mit „Datei“ → „Speichern“.

4 Datenstruktur

4.1 Grundsätzliches in aller Kürze

- **Approximationsrate:** Um schneller an Ergebnisse zu kommen, werden Karten verkleinert und somit sind weniger Daten zu verarbeiten.
- **Automat, Stadtteil, Koordinate:** Klassen, in welchen die wichtigsten Daten der einzelnen Automaten, Stadtteile, Koordinaten gespeichert sind.
- **Karte:** Karte, die aus einzelnen Pixeln besteht, und für jeden Pixel einen Wert enthält.
- **Welt:** Speichert die wichtigsten Parameter, wie die einzelnen Stadtteile, die Pixelkarte, den Radius der Automaten oder die Anzahl der Automaten, jedoch keine Lösung des Problems. Die Positionen der einzelnen Automaten werden hier also *nicht* gespeichert.
- **Zustand:** Ausgehend von einer Welt wird hier gespeichert, wo die Automaten stehen und welches Gewicht sie haben.

4.2 Automat

Wie der Name schon sagt beschreibt die Klasse `infomaticup.datenstruktur.Automat` einen Automaten. Es werden keine Berechnungen durchgeführt, sondern nur Informationen über einen einzelnen Automaten gespeichert. Dazu gehören Position und Radius des Automaten, sowie ein Wahrheitswert, der angibt, ob der Automat gesperrt ist. Wenn ein Automat gesperrt ist, darf er nicht gelöscht werden und seine Position darf nicht verändert werden.

4.3 Koordinate

Die Klasse `informaticup.datenstruktur.Koordinate` wird für die Speicherung der Positionen der Automaten sowie der Punkte der Polygone der Stadtteile verwendet. Die Klasse enthält keine Algorithmen oder Berechnungsschritte.

4.4 Pixelkarte

Lösungsidee Die Pixelkarte ist eine Konvertierung der Welt als Vektorkarte in eine Pixeldarstellung. Sie enthält einzelne Stadtteile, deren Begrenzungen keine Vektoren mehr sind, sondern Striche auf einer Pixelkarte.

Bei der Erzeugung der Pixelkarte wird die Karte automatisch skaliert (approximiert). Wird beispielsweise ein Approximationsfaktor von 2 angegeben, so werden sämlichte vorkommende Koordinaten durch 2 geteilt. Die Auflösung der Karte wird also effektiv durch den Faktor 4 geteilt. Der Approximationsfaktor kann entweder manuell eingegeben werden oder automatisch ermittelt werden. Dadurch kann das Programm auch extreme Eingabedateien mit einer großen Kartenbreite oder -höhe verarbeiten. Große Eingabedateien werden einfach heruntergerechnet auf kleinere Eingabedateien. Natürlich nimmt dadurch auch die Auflösung und damit die Genauigkeit der Lösung ab. Die Geschwindigkeit der Algorithmen nimmt dafür zu. Wie genau der Zusammenhang zwischen der Geschwindigkeit der Algorithmen und der Größe der Karte aussieht, kann den Laufzeitabschätzungen der einzelnen Algorithmen entnommen werden.

Die Pixelkarte speichert für die einzelnen Pixel keine Farben, sondern Gewichte. Ein Pixel kann sich in einem bestimmten Stadtteil befinden, dem nach der Gewichtungsfunktion der Attribute ein bestimmter Gewichtswert zugewiesen wurde. Jeder Pixel in einem Stadtteil erhält das Gewicht des betreffenden Stadtteils geteilt durch die Anzahl an Pixel im Stadtteil als Wert. Summiert man nun beispielsweise nur die Hälfte der Pixel des Stadtteils auf, erhält man nur die halbe Bewertung des Stadtteils.

Programm-Dokumentation Die Koordinaten der einzelnen Stadtteile wurden zuvor vom Parser erstellt, dabei wurden einfach die Werte aus der Eingabedatei übernommen. Die Methode `informaticup.Weltfunktion.Approximation` skaliert (approximiert) die Karte um einen bestimmten Approximationsfaktor. Dazu werden einfach alle Koordinaten der Polygone aller Stadtteile durch den Approximationsfaktor geteilt. Ebenso wird der Radius der Automaten durch den Approximationsfaktor geteilt.

Nun beginnt die eigentliche Berechnung der Pixelkarte. Dazu werden Java-interne Funktionen verwendet, da diese einseits sehr schnell und andererseits bei extremen Eingabedaten (z.B. Stadtteilen, die aus keinem Pixel bestehen) nicht anfällig für Fehler sind[1]. Die Funktion `informaticup.Weltfunktion.erstellePixelkarte` erstellt die eigentliche Pixelkarte. Dazu werden die Stadtteile als gefüllte Polygone auf ein `BufferedImage` geschrieben. Danach werden die einzelnen Pixel aus dem Bild ausgelesen und in ein Array geschrieben, auf das effizienter zugegriffen werden kann. Anfängliche Probleme gab es, weil das Gewicht durch die Anzahl der Pixel geteilt wird und dadurch Werte enstanden, die kleiner als 1 sind. Das `BufferedImage` unterstützt jedoch nur Ganzzahlen. Als Lösung wird ein zweites `BufferedImage` erstellt, das angibt, durch welchen Wert jeder Pixel geteilt werden soll.

Laufzeitabschätzung Das Skalieren (Approximieren) der Karte hängt lediglich linear von der Anzahl der Koordinaten ab, die verarbeitet werden sollen. Wie schnell das Zeichnen der Polygone geht, ist uns nicht genau bekannt, da diese Aufgabe von Java-Bibliotheken übernommen wird. Das Übertragen der einzelnen Pixel-Werte in das Array hängt linear von der Anzahl der Pixel und damit der Größe der Karte ab. Die Skalierung beschleunigt diesen Prozess nur um einen konstanten Faktor.

4.5 Stadtteil

Die Klasse `informaticup.datenstruktur.Stadtteil` wird für die Speicherung der Stadtteile verwendet. Für jeden Stadtteil werden die Name und Werte der einzelnen Attribute gespeichert, sowie die Eckpunkt der Polygone, die den Stadtteil abgrenzen, wobei diese Punkte Koordinaten auf der Pixelkarte sind. Das Gewicht eines Stadtteils, das aus der Gewichtung der Attribute ergibt, wird in der Variable `_gewichtNachFkt` gespeichert.

Der einzige Berechnungsschritt in dieser Klasse ist die Berechnung der Fläche des Polygons (Methode `berechneNebeneffekte`). Diese ergibt sich nach der gaußschen Trapezformel.

$$2A = \sum_{i=1}^{n} (y_i + y_{i+1})(x_i - x_{i+1}) \tag{1}$$

Die Laufzeit dieses Berechnungsschrittes hängt linear von der Anzahl der Punkte eines Polygons ab.

[1] Diesen Teil hatten wir ursprünglich komplett selbst geschrieben.

4.6 Welt

Die Klasse `informaticup.datenstruktur.Welt` beschreibt die Probleminstanz ohne den Zustand, also ohne die Lösung (Automaten). Die Welt besteht aus den einzelnen Stadtteilen, dem Radius der Automaten, der Anzahl der zu platzierenden Automaten und der Pixelkarte. Die Lösung des Problem wird später in einen `Zustand` geschrieben. In der `Welt` wird außerdem der Approximationsfaktor gespeichert. Die `Welt` wird jedoch nur approximiert, wenn die entsprechende Methode in der `Weltfunktion` aufgerufen wird.

4.7 Gewichtskarte

Lösungsidee Da viele Algorithmen simulieren, wie Automaten an den verschiedensten Stellen bewertet würden, war eine Überlegung um die Programmeffizienz zu steigern, eine zweidimensionale Karte vorauszuberechnen, die für jede Koordinate die erwartete Bewertung beinhaltet. Da der Radius eines Automaten durch die Eingabedatei festgelegt ist, kann diese von fast jedem Algorithmus benötigte Gewichtskarte vorausberechnet werden. Ursprünglich wurde für jede Koordinate `(x, y)` das umliegende Quadrat mit der Kantenlänge des Kreisdurchmessers `[(x-r..x+r),(y-r..y+r)]` betrachtet: Mittels des Satzes des Pythagoras wurde bestimmt, ob der Punkt innerhalb des umliegenden Kreises liegt, und falls dies zutrifft, das Gewicht des darunter befindlichen Stadtteiles `getGewicht(x, y)` zu einer spezifischen Summe der Koordinate hinzuaddiert. Dieses Verfahren war äußerst ineffizient, da für jede mögliche Koordinate viele Berechnungen durchgeführt werden mussten. Um dies zu vermeiden, wurde die folgende Optimierung vorgenommen: In einer separaten Variable der Größe `short Kreis = [2r, 2r]` wurde ein derartiger Kreis mittels pythagoräischer Betrachtung vorberechnet. Sofern der Punkt im Kreis lag (also die Bedingung $(x-r)^2 + (y-r)^2 \leq r^2$ erfüllt ist) enthält das Kreisarray an der Stelle `(x,y)` den Wert `1`, ansonsten `0`. Aufsummiert wurde nun nicht mehr `getGewicht(x, y)`, sondern `getGewicht(x, y) * Kreis[x][y]`. Um die benötigte Zeit zur Berechnung der Gewichtskarte weiter zu reduzieren, wurden die so genannten **Deltakreise** eingeführt. Es ist nämlich zu erkennen, dass das Gewicht des Pixels `(x+1, y)` auch aus dem Gewicht des Punktes `(x, y)` berechnet werden kann. Selbiges gilt für `(x, y+1)` und `(x, y)`. Deshalb werden vergleichbare Arrays (`short deltakreis_rechts = [2r+1, 2r]` und `short deltakreis_unten = [2r, 2r+1]`) vorausberechnet, um die Anzahl der notwendigen Operationen weiter zu reduzieren. Dem Bildbereich dieser beiden Arrays wurde der Wert `-1` hinzugefügt. Als Berechnungsanweisung wird der gleiche Algorithmus wie für den normalen Kreis angewandt (*Multiplikation des Gewichts mit*

dem Inhalt des Arrays).

Laufzeitabschätzung Das Generieren der Gewichtskarte hngt von mehreren Faktoren ab: Von der Kartengröße `Weltfunktion.getMaximaleBreite-Hoehe(true) = A`, den Automatenradien `Welt.getRadiusAutomaten() = r` und dem Approximationsfaktor `Welt.getApproxrate() = a`. Für jeden Punkt der approximierten Karte, also insgesamt $\frac{A}{a}$ Punkte, müssen die Referenz-/Deltakreis-Arrays durchlaufen werden, welche widerum eine Größe von $4{*}r^2$, bzw. vernachlässigbar $4 * r * (r + 1)$ besitzen. Daraus folgt, dass je Koordinate $\frac{A{*}4{*}r^2}{a}$ Vergleichsoperationen durchgeführt werden müssen. Das Ergebnis der Vergleichsoperation bestimmt, ob ein Wert addiert (subtrahiert) werden muss. Durchschnittlich findet dies alle $\frac{2{*}2}{\pi} = \frac{4}{\pi}$ Koordinaten statt.

4.8 Zustand

Die Klasse `informaticup.datenstruktur.Zustand` beschreibt die Lösung einer Probleminstanz, also wo die einzelnen Automaten platziert worden sind. Dazu gibt es ein Array `_automaten`. Im Zustand werden außerdem die Bewertung der aktuellen Lösung und ein Zeiger auf die Gewichtskarte gespeichert. Die Gewichtskarte ist nur über den Zustand erreichbar und wird beim ersten Zugriff erstellt und gespeichert. Erfolgen später weitere Zugriffe auf die Gewichtskarte, muss diese nicht neu berechnet werden. In dieser Klasse befinden sich deshalb auch der Algorithmus zur Berechnung der Gewichtskarte.

Vor der Berechnung der Automatenpositionen durch die Algorithmen wird bereits ein Automaten-Array mit der korrekten Größe erstellt, welches jedoch nur `null`-Zeiger enthält. Somit kann bereits vor Ausführung eines Algorithmus durch die GUI auf das Automaten-Array zugegriffen werden, um einzelne Automaten per Hand zu erstellen.

5 Typischer Programmablauf

Ein typischer Programmablauf mit Anwendung der Algorithmen und Generieren einer Ausgabedatei sieht wie folgt aus.

1. Benutzer lädt eine Eingabedatei durch Auswählen einer Datei im Dateiauswahl-Dialog.
   (a) Parser liest die Eingabedatei ein und erzeugt eine `Welt` (`informaticup.createCalcFrame.startParsing`).
   (b) Attribute werden aus der `Welt` ausgelesen und in die Tabelle hinzugefügt (`informaticup.createCalcFrame.loadFile`)

2. Benutzer gewichtet die Attribute in der Tabelle (optional).
3. Benutzer startet den Berechnungvorgang (`informaticup.InformaticupView.startCalculation`).
   (a) Gewichtung der einzelnen Attribute wird ausgewertet und die Stadtteile erhalten ihr Gewicht `_gewichtNachFkt` (`informaticup.createCalcFrame.bereiteBerechnungVor`).
   (b) Parameter und ausgewählte Algorithmen werden ausgewertet, Variablen werden gesetzt (`informaticup.InformaticupView.startCalculation`), neuer Task für die Berechnung wird erstellt (`informaticup.InformaticupView.StartCalculationTask.doInBackground`).
   (c) Koordinaten werden approximiert/skaliert (`informaticup.Weltfunktion.Approximation`).
   (d) Wenn kein Algorithmus ausgewählt wurde und die Karte nur gezeichnet werden soll: Neuer `Zustand` wird erstellt, das heißt, bereits gesetzte Automaten werden verworfen.
   (e) Pixelkarte wird erstellt (`informaticup.Weltfunktion.erstellePixelkarte`).
   (f) Wenn bisher noch kein `Zustand` existiert: Erstelle neuen `Zustand`. Bestehende Automaten werden nicht gelöscht.
   (g) Gewichtskarte wird erzeugt (`informaticup.datenstruktur.Zustand.erzeugeGewichtskarte`).
   (h) Führe Eröffnungsalgorithmus aus (Greedy, Greedy mit Strichproben, Zufall oder Backtracking).
   (i) Führe Optimierungsalgorithmus aus (Simulierte Abkühlung, Tabu-Suche, und/oder Einzelverschiebung).
   (j) Zeichne die Lösung (Klasse `informaticup.drawingPanel`).
4. Benutzer speichert die Lösung (`informaticup.Dateiexport.dateiAusgeben`).

5.1 Dateieingabe und Dateiausgabe

Eine Datei wird sofort geladen, wenn der Benutzer die Datei im Dateiauswahl-Dialog öffnet. Die Klasse `informaticup.Dateiimport` ist dafür zuständig, die komplette Datei erst einmal in eine Zeichenfolge zu lesen. Diese kann dann vom Parser `informaticup.Parser` zu einer `Welt` verarbeitet werden.

Dabei werden die einzelnen Stadtteile erzeugt, die Attribute ausgelesen und Radius und Anzahl der Automaten ermittelt. Schlägt dieser Vorgang fehl, wird eine `ParserException` geworfen.

Wenn alle Berechnung abgeschlossen sind, liegt ein `Zustand` vor, der vom Benutzer auch nachträglich noch verändert werden kann. Er kann beispielsweise Automaten verschieben. Ein solcher `Zustand` kann durch die Methoden in der Klasse `informaticup.Dateiexport` dann in eine Ausgabedatei geschrieben werden, die die in der Aufgabenstellung geforderte Form hat.

5.2 Weltfunktion

Die Klasse `informaticup.Weltfunktion` enthält Operationen und Funktionen, die auf die `Welt` angewandt werden können. Dabei handelt es sich um das Erstellen der Pixelkarte (siehe Kapitel *Zustand*) und das Approximieren der Koordinaten.

Außerdem existiert eine Funktion, die die maximale Breite bzw. Höhe der Karte ermittelt. Dieser Wert ist notwendig, um automatisch einen guten Wert für die Approximation zu finden. Die Karte wird dann so approximiert, dass bei jeder Karte etwa gleich große Pixelkarten entstehen, die in vernünftiger Zeit verarbeitet werden können. Große Karten werden also stärker verkleinert als kleine Karten.

6 Algorithmen

6.1 Modularer Aufbau

Aufgrund einer Vielzahl von möglichen Algorithmen, haben wir uns dazu entschlossen, eine vom Algorithmus unabhängige Datenstruktur und Benutzeroberfläche zu entwerfen. Diese Datenstruktur ermöglicht es, viele Algorithmen zu testen und somit den *besten* Algorithmus zu finden. Hinzu kommt, dass durch viele Algorithmen auch Spezialfälle leichter betrachtet werden können.

6.2 Güte einer Lösung

Bei dem gegebenen Problem, Automaten möglichst effizient auf einer Landkarte zu verteilen, handelt es sich um ein Optimierungsproblem. Damit eine Lösung als gültig (zulässig) klassifiziert wird, müssen die folgenden Kriterien erfüllt sein.

- Es müssen alle Automaten auf der Landkarte platziert sein.

- Jeder Automat muss auf der Landkarte und darf nicht außerhalb der Karte platziert sein.
- Die Bereiche (Kreise) der Automaten dürfen sich nicht überschneiden.

Für gültige Lösungen kann dann die *Güte* berechnet werden. Je höher dieser Wert ist, desto besser ist die Lösung (Maximierungsproblem). Jedem Pixel wurde zuvor bereits ein Gewicht zugewiesen. Die Güte einer Lösung ergibt sich aus der Summe der Gewichte aller Pixel, an denen sich ein Automat befindet. Da die einzelnen Pixel in der Pixelkarte bereits Gewichtswerte enthalten, die durch die Anzahl der Pixel im Stadtteil geteilt wurden, ergibt sich eine anteilsmäßige der Güte an den Stadtteilen. Wenn ein Stadtteil von einem Automaten nur zu 40 Prozent abgedeckt wird, gibt es für diesen Automaten nur 40 Prozent der Bewertung des Stadtteiles.

6.3 Brute-Force-Suche

Eine erste Idee war ein Brute-Force-Algorithmus. Das würde in unserem Programm bedeuten, dass jeder einzelne Punkt der Karte auf seinen Automatenwert untersucht wird. Ein solcher Algorithmus würde zu einer optimalen Lösung führen. Allerdings ist dieser Algorithmus nicht performant genug.

Wenn man annimmt, dass die Landkarte aus p Pixel besteht, so gibt es für jeden Automaten p Möglichkeiten, ihn zu platzieren. Natürlich dürfen sich die Automatenradius nicht überschneiden, sodass sich etwas weniger mögliche Positionen ergeben. Da die abgedeckte Fläche eines Automaten aber in der Regel viel kleiner als die Landkarte ist, kann man diesen Effekt bei der Laufzeitbetrachtung vernachlässigen. Sollen a Automaten platziert werden, ergeben sich $\mathcal{O}(p^a)$ mögliche Zustände (exponentielle Laufzeit), die überprüft werden müssen. Mit einem Zustand eine mögliche Platzierung aller a Automaten gemeint. Bereits bei $a = 2$ Automaten können die vorgegebenen Beispieleingaben nicht mehr in sinnvoller Zeit berechnet werden.

6.4 Zufällige Verteilung der Automaten

Vor allem als Eröffnungsverfahren für Metaheuristiken kann es interessant sein, eine zufällige gültige Lösung zu erzeugen. Dabei werden einfach alle Automaten per Zufall auf der Karte verteilt, wobei sich Automaten nicht überschneiden dürfen. Der Algorithmus befindet sich in der Klasse `algorithmen.Zufall` und die Laufzeit hängt quadratisch von der Anzahl der Automaten ab, da für jeden Automaten überprüft werden muss, ob er sich mit irgendeinem anderen Automaten überschneidet.

6.5 Heuristiken

Heuristiken bezeichnen in der Informatik solche Vorgehensweisen, bei denen ein Kompromiss zwischen dem Rechenaufwand und der Güte der gefundenen Lösung eingangen wird. Mit sehr großer Wahrscheinlichkeit findet die Heuristik also nicht die/eine optimale Lösung, dafür wird versucht, eine gute Annäherung in akzeptabler Zeit zu ermitteln.

6.5.1 Greedy-Algorithmus mit vollständiger Suche

Grundprinzip Der Greedy-Algorithmus (engl. *greedy* = gierig) wählt zu jedem Zeitpunkt den Schritt, der momentan am meisten Erfolg verspricht. Dieses Verfahren führt in der Regel nicht dazu, dass eine optimale Lösung gefunden wird. Ebenso kann nicht garantiert werden, dass die gefundene Lösung eine bestimmte Mindestgüte aufweist, die gefundene Lösung kann beliebig schlecht sein. Andererseits funktioniert das Verfahren relativ schnell.

Pseudocode Der Algorithmus geht bei der Berechnung nach folgendem Prinzip vor.

for $i = 1$ **to** Automaten **do**
  Beste Position $\leftarrow -1$
  for $x = 1$ **to** max_x **do**
    for $y = 1$ **to** max_y **do**
      if Bewertung(x, y) > Beste Position **then**
        Beste Position $\leftarrow$ Bewertung(x, y)
        $\text{best}_x \leftarrow x$
        $\text{best}_y \leftarrow y$
      end if
    end for
  end for
  Setzte Automat(best_x, best_y)
end for
return best_x, best_y

Der erste Schritt besteht darin, den Pixel mit dem maximalen Gewicht zu finden. Hat man diesen gefunden, platziert man an dieser Stelle den ersten Automaten. Dieser Schritt wird nun so lange wiederholt, bis alle Automaten platziert wurden. Es muss jedoch darauf geachtet werden, dass sich die Automatenkreise nicht überschneiden. Bevor ein Automat platziert wird, muss deshalb sichergestellt sein, dass der Abstand zu jedem anderen Automaten

mindestens den doppelten Automatenradius beträgt[2]. Im allgemeinen kann der erste Pixel der Landkarte, anders als im Pseudocode angegeben, auch eine andere Koordinate als (1|1) aufweisen.

Laufzeitkomplexität Besteht die Landkarte aus $p = \max_x \cdot \max_y$ Pixeln, so müssen für jeden Automaten p Pixel untersucht werden. Für jeden interessanten Pixel muss nun noch der Abstand zu allen anderen Pixel überprüft werden. Sollen a Automaten platziert werden, sind $\theta(a \cdot p)$ Vergleiche der Pixelgewichte und $\mathcal{O}(a^2 \cdot p)$ Abstandsberechnungen notwendig. Normalerweise ist a im Gegensatz zu p sehr klein und kann vernachlässigt werden.

Programm-Dokumentation Der Greedy-Algorithmus befindet sich in der Klasse `algorithmen.Greedy` und implementiert das Interface `informaticup.Algorithmus`. Der eigentliche Algorithmus startet, wenn die Methode `Berechne` aufgerufen wird. Die Methode `SetzeNaechstenAutomaten` platziert den nächsten Automaten und verwendet dazu die Methode `FindeBestePositionVollstaendigeSuche`, die über alle Pixel iteriert. Die Funktion `PunktZulaessig` ermittelt, ob an der angegebenen Position ein Automat platziert werden darf, indem der Abstand zu allen bereits gesetzten Automaten berechnet wird.

6.5.2 Greedy-Algorithmus mit Stichproben-Suche

Grundprinzip Der vorgestellte Algorithmus kann beschleunigt werden, wenn nicht alle Pixel auf ihr Gewicht überprüft werden. Stattdessen wird die Karte horizontal oder vertikal in zwei gleich große Teile (Kartenausschnitte) aufgeteilt. Dann wird aus beiden Teilen eine Stichprobe von s Pixeln entnommen. Für jeden Teil werden die Gewichte der ausgewählten Pixel aufsummiert. Dann wird der Teil mit der größeren Summe erneut aufgeteilt. Eine horizontale Teilung erfolgt, wenn der betrachtete Kartenausschnitt eine größere Breite als Höhe aufweist. Ansonsten wird vertikal geteilt. Wenn ein Kartenausschnitt aus nur noch einem Pixel besteht, endet das Verfahren. Unter allen insgesamt ausgewählten Pixeln wird der Pixel mit dem maximalen Gewicht als neuer Automatenstandort festgelegt. Natürlich muss noch sichergestellt werden, dass kein Pixel ausgewählt wird, sodass sich die Automatenkreise überschneiden.

Dieses Verfahren ist vor allem dann sinnvoll, wenn man annimmt, dass sich Stadtteile mit einer guten Gewichtung nebeneinander befinden. Der Algorithmus betrachtet die Karte zunächst sehr grob. Die Stellen, die besonders

[2]Diese Überprüfung auf Gültigkeit fehlt der Übersicht halber im Pseudocode.

gut aussehen, betrachtet er dann näher. Außerdem kann man davon ausgehen, dass Stadtteile relativ groß sind und viele Pixel umfassen. In allen Pixeln eines Stadtteils ist das Gewicht gleich. Deshalb kann man bereits bei einer relativ kleinen Anzahl an Stichproben mit großer Sicherheit einen guten Pixel finden.

Pseudocode Der Algorithmus geht bei der Berechnung nach folgendem Prinzip vor.

```
Punkt 1 ← (1|1)
Punkt 2 ← (max_x|max_y)
Bester Punkt ← −1
while Bereich groß genug do
  Teil A/B Punkt 1/2 ← Teile Bereich()
  Stichproben_{A/B} ← 0
  for i = 1 to Stichprobengröße do
    Punkt_A ← Stichprobe(Teil A Punkt 1, Teil A Punkt 2)
    Punkt_B ← Stichprobe(Teil B Punkt 1, Teil B Punkt 2)
    Stichproben_A ← Stichproben_A + Bewertung(Punkt_A)
    Stichproben_B ← Stichproben_B + Bewertung(Punkt_B)
    Bester Punkt ← max (Punkt_A, Punkt_B, Bester Punkt)
  end for
  if Stichproben_A > Stichproben_B then
    Punkt 1 ← Teil A Punkt 1
    Punkt 2 ← Teil A Punkt 2
  else
    Punkt 1 ← Teil B Punkt 1
    Punkt 2 ← Teil B Punkt 2
  end if
end while
return Bester Punkt
```

Im Pseudocode fehlt die Überprüfung der Punkte auf Gültigkeit der Übersichtlichkeit halber. Ein Bereich, aus dem Stichproben entnommen werden können, ist durch zwei Punkte definiert, dem Punkt links oben (Punkt 1) und dem Punkte rechts unten (Punkt 2). Anders als Pseudocode muss die Karte nicht notwendigerweise mit dem Pixel (1|1) beginnen[3]. Stichproben werden zufällig aus dem angegebenen Bereich entnommen.

Laufzeitkomplexität Es ist sinnvoll, die Größe der Stichprobe an die Größe der Karte zu koppeln, da bei größeren Karten auch größere Stichpro-

[3]Analog zum Greedy-Algorithmus mit vollständiger Suche

ben benötigt werden. Die Größe der Stichprobe ergibt sich aus $s = q \cdot r$, wobei r die Größe des Kartenausschnitts (Anzahl der Pixel) und q das Verhältnis der zu entnehmenden Pixel bezogen auf den Kartenausschnitt ist. Im ersten Schritt wird die Karte in zwei gleich große Teile geteilt und es werden zweimal (für jeden Kartenausschnitt) $0,5 \cdot p \cdot q$ Pixel ausgewählt (p ist die Größe der Karte). Im nächsten Durchlauf werden zweimal $0,25 \cdot p \cdot q$ Pixel ausgewählt. Pro Automat werden also $\sum_{i=0}^{\infty} 2^{-i} \cdot p \cdot q = 2 \cdot p \cdot q$ Pixel untersucht. Somit untersucht der Algorithmus insgesamt $\theta(a \cdot 2 \cdot p \cdot q)$ Pixel. In der Implementation des Algorithmus wurde $q = 0,1$ gesetzt.

Hinweis: In der Regel ist bereits der Greedy-Algorithmus schnell genug. Beide Greedy-Algorithmen waren bei unseren Tests (vorgegebene Testfälle) so schnell, dass sich kein wirklicher Unterschied feststellen lässt.

Programm-Dokumentation Der Greedy-Algorithmus mit Stichproben-Suche befindet sich ebenfalls in der Klasse `algorithmen.Greedy`. Falls der Parameter `_stichproben` auf `true` gesetzt ist, wird der Stichproben-Algorithmus statt der vollständigen Suche verwendet. Die Funktion `FindeBestePositionStichprobenSuche` ermittelt die neue Position, an der ein Automat platziert werden soll. Die Abbruchbedingung der `while`-Schleife ist die Überprüfung, ob der betrachtete Bereich nur noch aus wenigen Pixeln besteht. Jede entnommene Stichprobe wird automatisch daraufhin untersucht, ob der Pixel besser als alle zuvor untersuchten Pixel ist. Der insgesamt beste entnommene Pixel (Stichprobe) wird als Ergebnis der Funktion zurückgegeben. Dadurch, dass der Algorithmus gute Bereiche näher untersucht, werden aus diesen guten Bereichen mehr Stichproben entnommen, als aus anderen Bereichen. Es gibt eine Klasse `GreedyStichprobe`, die keinen Algorithmus-Quelltext enthält, sondern nur die `Greedy`-Klasse mit den entsprechenden Parametern aufruft.

6.5.3 Optimierung durch Verschiebung einzelner Automaten

Grundprinzip Dieser Algorithmus führt eine Abschlussoptimierung einer bestehenden Lösung durch. Es muss also bereits eine Lösung vorliegen. Es wird dann jeder Automat einzeln untersucht und überpüft, ob sich in der unmittelbaren Nachbarschaft des Automaten eine bessere Position befindet. Eine Position ist genau dann besser, wenn sich durch die Änderung die Güte der Lösung verbessert.

Pseudocode Der Algorithmus geht bei der Berechnung nach folgendem Prinzip vor.

```
L ← Bestehende Lösung()
G ← Güte(L)
for i = 1 to 3 do
  A ← Mischen(Automaten)
  for all a ∈ A do
    for all n ∈ Nachbarschaft(a) do
      if Güte(n) > G then
        L ← n
        G ← Güte(n)
      end if
    end for
  end for
end for
return L
```

Bevor der Algorithmus beginnen kann, benötigt er eine bestehende zulässige Lösung, die zum Beispiel vom Greedy-Algorithmus oder einer Metaheuristik (nächstes Kapitel) erstellt wurde. Als Vergleichswert wird die Güte dieser Lösung berechnet.

Nun folgen drei Durchläufe. In jedem Durchlauf werden alle Automaten in zufälliger Reihenfolge sequentiell bearbeitet. Für jeden Automaten wird die Nachbarschaft generiert, die alle Lösungen enthält, bei denen der ausgewählte Automat um bis zu eine Radienbreite in jede Richtung verschoben sein kann. Falls in der Nachbarschaft eine bessere Lösung existiert, wird diese ausgewählt und übernommen. Natürlich werden nur zulässige Lösungen akzeptiert, bei denen sich die Automaten nicht überschneiden[4].

Laufzeitkomplexität Die Laufzeit dieses Algorithmus hängt linear von der Anzahl der Automaten und der Größe der Nachbarschaft ab. Die Überprüfung, ob ein Automat an einer zulässigen Position gesetzt wurde, geht in linearer Zeit (in Abhängigkeit von a), da in jedem Schritt nur ein einziger Automat verschoben wird. Wenn der Radius der Automaten r beträgt und a Automaten vorliegen, hat der Algorithmus eine Laufzeit von $\mathcal{O}(a^2 \cdot 4r^2)$.

Programm-Dokumentation Der Algorithmus befindet sich in der Klasse `algorithmen.Einzelverschiebung` und implementiert das Interface `informaticup.Algorithmus`. Der eigentliche Algorithmus befindet sich in der Methode `Berechne`. Um in einem Durchlauf einen zufälligen Automaten auszuwählen, wird ein Array `automat` mit den Indizes der Automaten erstellt.

[4] Auf die Überprüfung auf Zulässigkeit wurde, wie auch bei allen anderen Pseudocodes, der Übersichtlichkeit halber im Pseudocode verzichtet.

Dieses Array wird von der Funktion `MischeArray` zufällig durcheinandergemischt, indem jedes Element mit einem zufälligen Element vertauscht wird. Die genaue Bedeutung der einzelnen Variablen und Funktionen kann dem gut dokumentierten Quelltext entnommen werden.

6.6 Metaheuristiken

Im Gegensatz zu Heuristiken sind Metaheuristiken nicht auf ein spezifisches Problem beschränkt. Sie beschreiben allgemeine Vorgehensweisen zur Lösung von Problemen einer bestimmter Art. Wir haben zwei verschiedene Metaheuristiken - Simulierte Abkühlung und Tabu-Suche - implementiert, die beide in etwa nach folgendem Prinzip vorgehen.

1. Ermittle eine Initiallösung mit einem beliebigem Eröffnungsverfahren.
2. Durchsuche die (lokale) Nachbarschaft der aktuellen Lösung.
3. Wähle die beste gefundene Lösung aus der Nachbarschaft aus.

Ein Eröffnungverfahren ist ein erster Algorithmus, der eine zulässige Anfangslösung findet. Je besser diese Anfangslösung ist, desto einfacher kann die Metaheuristik später gute Lösungen finden. Als Eröffnungsverfahren können beispielweise alle Automaten zufällig verteilt werden, sofern diese zufällige Verteilung zulässig ist. Besser ist jedoch, die Initiallösung mit dem Greedy-Algorithmus zu generieren.

Die Nachbarschaft einer zulässigen Lösung ist die Menge von Lösungen, die aus der aktuellen Lösung durch elementare Operationen erzeugt werden können. Eine elementare Operation könnte zum Beispiel das Verschieben eines Automaten um wenige Einheiten sein.

6.6.1 Anwendbarkeit von Metaheuristiken

Metaheuristiken können bei unserem Problem angewandt werden, weil es sich um ein kombinatorisches Optimierungsproblem handelt, d.h. die Menge der (gültigen) Lösungen ist diskret und aufzählbar. Durch elementare Operationen, wie z.B. das Verschieben eines Automaten, kann jede Lösung, also auch eine optimale Lösung, erreicht werden.

Laufzeitprobleme können sich dann ergeben, wenn die lokale Nachbarschaft einer Lösung zu groß ist, um alle Nachbarn zu überprüfen. Deshalb betrachten die implementierten Metaheuristiken immer nur eine zufällig ausgewählte Teilmenge der Nachbarschaft.

Die Güte der durch die Metaheurisitik gefundenen Lösung kann nicht abgeschätzt werden. Die gefundene Lösung kann beliebig schlecht sein. Wählt

man die Parameter (z.B. Temperatur bei Simulierter Abkühlung) der Algorithmen günstig, so erhält man in der Regel sehr gute Ergebnisse.

6.6.2 Simulierte Abkühlung (simulated annealing)

Grundprinzip Simulierte Abkühlung ist die Nachbildung des physikalisches Prozesses der Abkühlung eines Metalles. Anfangs hat das Metall eine sehr hohe Temperatur und die Atome im Metall befinden sich in einem sehr hohen energetischen Zustand. Durch langsame Abkühlung können sich die Atome so neu anordnen, dass ein energiearmer Zustand erreicht wird. Kühlt man das Metall zu schnell ab, kann es *brüchig* werden und ist von schlechter Qualität. Das Metall befindet sich dann nur in einem lokal minimalen Energiezustand.

Genauso wie das Metall in einen engergieärmeren Zustand überführt wird, soll nun die Initiallösung in einen energieärmeren Zustand überführt werden. Ein Zustand ist genau dann besonders energiearm, wenn er eine hohe Güte aufweist.

Pseudocode Der Algorithmus geht bei der Berechnung nach folgendem Prinzip vor.

$L \leftarrow$ Eröffnungsverfahren()
$E \leftarrow$ Energie(L)
$T \leftarrow$ Initialtemperatur()

while $T > 0$ **do**
  for $i = 1$ **to** Durchläufe **do**
    $L_{alt} \leftarrow L$
    for $j = 1$ **to** Operationen **do**
      $L \leftarrow$ ElementareOperation(L)
    end for

    $E_{neu} \leftarrow$ Energie(L)
    $\Delta E \leftarrow E_{neu} - E$

    if $\Delta E > 0$ **then**
      $z \leftarrow$ Zufallszahl(0, 1)
      if $z \geq e^{\frac{-\Delta E}{T}}$ **then**
        $L \leftarrow L_{alt}$
      end if
    end if

end for

$T \leftarrow T - 1$
end while
return L

Zuerst wird mit einem Eröffnungsverfahren (z.B. Greedy-Algorithmus) eine gültig Initiallösung erzeugt. Eine Bewertungsfunktion weist dieser Lösung dann ein Energieniveau zu. Je besser die Lösung ist, desto niedriger ist die Energie. Der Benutzer hat die Möglichkeit, die Initialtemperatur beliebig festzulegen. Bei höheren Temperaturen dauert die Berechnung länger, jedoch wird auch das Ergebnis besser.

In jedem Durchlauf der While-Schleife wird die Temperatur nun um eine Einheit erniedrigt. Erst wenn die Temperatur auf Null ist, terminiert der Algorithmus: Das System ist *gefroren*. Für jede Temperatur wird nun eine bestimmte Anzahl an *Durchläufen* durchgeführt. Jeder Durchlauf entspricht dem Generieren eines Nachbarn. Dazu wird eine bestimmte Anzahl an elementaren Operationen ausgeführt. Je höher diese Anzahl an elementaren Operationen ist, desto weiter kann der Nachbar von der Ausgangslösung entfernt sein. Jedoch steigt dann auch die Wahrscheinlichkeit, dass unzulässige Lösungen generiert werden, sodass effektiv weniger Durchläufe stattfinden. Der Einfachheit halber wird im Pseudocode davon ausgegangen, dass nur zulässige Lösungen generiert werden können.

Es wird nun das Energieniveau der neuen Lösung und die Abweichung gegenüber der vorherigen Lösung berechnet. Falls dieser Delta-Wert größer als Null ist, hat sich die Lösung verbessert und sie wird akzeptiert. Wurde die Lösung jedoch schlechter, so wird sie nur mit einer gewissen Wahrscheinlichkeit angenommen. Die Metropolis-Regel besagt, dass die schlechtere Lösung nur mit einer Wahrscheinlichkeit von $e^{\frac{-\Delta E}{T}}$ (Akzeptierungsfunktion) akzeptiert wird. Würde man nur bessere Lösungen akzeptieren, würde der Algorithmus ziemlich sicher in einem lokalen Energieminimum steckenbleiben.

Die Akzeptierungsfunktion bildet auf einen Bereich zwischen 0 und 1 ab. Um mit der Wahrscheinlichkeit der Akzeptierungsfunktion zu akzeptieren, wird eine Zufallszahl zwischen 0 und 1 gebildet. Ist die Zufallszahl kleiner als der Wert der Akzeptiertungsfunktion wird akzeptiert. Ansonsten werden alle elementaren Operationen im aktuellen Durchlauf rückgängig gemacht. Nach einigen Durchläufen wird die Temperatur erniedrigt.

Verbesserungen und Erweiterungen Der Algorithmus wurde um einige Ideen erweitert, um ihn leistungsfähiger zu machen.

- **Variable Größe der elementaren Operationen.** Es ist möglich,

die Größe der elementaren Operationen, also der Verschiebungen der Automaten, von der Temperatur abhängig zu machen. So werden dann die Automaten bei einer hohen Temperatur stärker verschoben als bei einer niedrigeren Temperatur, was der Anschauung genügt, dass sich die Atome im Metall bei hoher Temperatur schneller bewegen.

- **Mehrfache Anwendung des Algorithmus.** Verschiedene Probleminstanzen erfordern ggf. verfschiedene Parameter. Deshalb wird der Algorithmus nach der ersten Ausführung bei niedrigerer Initialtemperatur nochmals mehrere Male aufgerufen. Parameter wie die Anzahl der Durchläufe, die Anzahl der elementaren Operationen oder die Berechnung der elementaren Operationen in Abhängigkeit von der Temperatur werden dabei variiert.
- **Speichern der global besten Lösung.** Unter allen Aufrufen des Algorithmus und allen temporär generierten Lösungsvorschlägen (bei jeder Temperatur) wird die beste gefundene Lösung gespeichert. Es ist jedoch sehr wahrscheinlich, dass die gefundene Lösung nach Anwendung des Algorithmus ohnehin die *global* beste gefundene Lösung ist.

Generieren der Nachbarn Jeder Durchlauf entspricht dem Generieren eines Nachbarn. In jedem Durchlauf wird eine bestimmte Anzahl an elementaren Operationen ausgeführt. Eine elementare Operation ist das Verschieben eines einzigen Automaten um einen zufälligen Wert. Hängt die Größe der elementaren Operationen nicht von der Temperatur ab, beträgt die maximale Verschiebung in x-Richtung[5] $max_x = \frac{6 \cdot \text{Welt Breite}}{100}$. Soll die Verschiebung dagegen in Abhängigkeit von der Temperatur erfolgen, so beträgt die maximale Verschiebung in x-Richtung $max_x = \frac{10 \cdot T \cdot \text{Welt Breite}}{100 \cdot T_{initial}}$. Die tatsächliche Verschiebung in x-Richtung ist ein zufälliger Wert zwischen $-max_x$ und max_x.

Laufzeitkomplexität Das Laufzeitverhalten des Algorithmus hängt linear von der Initialtempertur, der Anzahl der Durchläufe und der Anzahl der Operationen ab. Die Überprüfung, ob ein Lösungsvorschlag zulässig ist, geht in $\mathcal{O}(a^2)$, da jeder Automat mit jedem anderen Automaten verglichen[6] werden muss. Die Güte einer Lösung kann in $\mathcal{O}(a)$ ermittelt werden, da dazu nur jede Automatenposition in der Gewichtskarte nachgeschlagen werden muss. Somit ergibt sich insgesamt ein Laufzeitverhalten von $\mathcal{O}(T \cdot \text{Durchläufe} \cdot \text{Operationen} \cdot a^2)$.

[5] y-Richtung analog
[6] Vergleich des Abstandes

Programm-Dokumentation Der Abkühlungs-Algorithmus befindet sich in der Klasse `Abkuehlung` und implementiert das Interface `informaticup.-Algorithmus`. Wenn die Methode `Berechne` aufgerufen wird, wird das Berechnungsverfahren gestartet. Der eigentliche Algorithmus wird dann mehrere Male aufgerufen, wobei Parameter wie die Anzahl der Durchläufe oder die Anzahl der Veränderungen pro Durchlauf verändert werden. Wenn die Methode `Optimiere` aufgerufen wird, startet die Simulierte Abkühlung. Die Methode `ElementareOperationen` wird verwendet, um einen neuen Nachbarn (ein Durchlauf) zu erzeugen. `BerechneZustandsGuete` berechnet die Güte (nicht Energie!) des akutellen Zustandes und wurde so auch in anderen Algorithmen übernommen. Bei der Methode `AkzepttiereAenderung` muss beachtet werden, dass sich die Energiedifferenz zweier Zustände aus der invertierten Differenz der Güte $-$(neue Güte$-$alte Güte) ergibt, weil das Energieniveau umso niedriger ist, je höher die Güte ist. Die genaue Bedeutung und Verwendung der einzelnen Variablen kann dem gut kommentierten Quelltext entnommen werden.

6.6.3 Tabu-Suche

Grundprinzip Tabu-Suche ist eine Metaheuristik, die in jedem Schritt die bestmögliche zulässige Lösung in der Nachbarschaft sucht und akzeptiert. Ziel ist es, eine Lösung mit einer möglichst hohen Güte zu finden. Um nicht in lokalen Maxima stecken zu bleiben und um nicht *im Kreis* zu laufen, werden in einer Tabu-Liste die letzten bereits besuchten Lösungen gespeichert. Diese Lösung sind von nun an *tabu* und dürfen nicht mehr besucht werden. Es wird immer die beste gefundene Lösung in der Nachbarschaft akzeptiert, unabhaängig davon, ob die aktuelle Lösung dadurch besser oder schlechter wird. Würde man keine Tabu-Liste anlegen, wäre es beispielsweise denkbar, dass die aktuelle Lösung X einen besten Nachbarn Y hat, der jedoch schlechter als X ist. Dennoch würde der Algorithmus nun Y wählen. Im nächsten Schritt könnte der beste Nachbar erneut X sein. Würde man nun X erlauben, hätte man einen endlosen Kreis X, Y, X, Y, ...

Pseudocode Der Algorithmus geht bei der Berechnung nach folgendem Prinzip vor.

$L \leftarrow$ Eröffnungsverfahren()
$G \leftarrow$ Güte(L)
$T \leftarrow ()$
$Beste \leftarrow L$

for $i = 1$ **to** Iterationen **do**

repeat
$L_{neu} \leftarrow$ Nächste beste Lösung()
until $L_{neu} \notin T$

$Beste \leftarrow \max(Beste, L_{neu})$
Füge L_{neu} zu T hinzu
Entferne alte Elemente aus T
end for
return Beste

Der erste Schritt besteht im Generieren einer gültigen Initiallösung durch ein Eröffnungsverfahren. Die Güte der Lösung wird ebenfalls ermittelt und gespeichert. Außerdem wird eine leere Tabu-Liste angelegt und die Variable für beste bisher gefundene Lösung auf die Initiallösung gesetzt.

Es wird nun eine bestimmte Anzahl an Iterationen ausgeführt, die vom Benutzer beliebig festgelegt werden kann. In jeder Iteration wird zunächst der beste zulässige Nachbar gesucht. Ein Nachbar ist genau dann zulässig wenn er gemäß Kapitel *Güte einer Lösung* zulässig ist und nicht auf der Tabu-Liste auftaucht. Dieser beste zulässige Nachbar wird dann auf die Tabu-Liste gesetzt und ggf. in die Variable für die beste bisher gefundene Lösung gespeichert. Es ist sinnvoll, die Größe der Tabu-Liste zu beschränken, um ein (konstant) besseres Laufzeitverhalten zu erhalten. Wird die Tabu-Liste zu groß, werden die ältesten Elemente in der Liste entfernt.

Implementiert man den Algorithmus genau so, wie eben beschrieben, läuft der Algorithmus relativ langsam, weil die Menge der Nachbarn sehr groß ist. Nachbarn sind alle diejenigen Lösungen, bei denen ein einziger Automat horizontal, vertikal oder horizontal und vertikal um maximal den Durchmesser der Automaten verschoben ist. Die Berechnung dauert besonders lange, weil viele Iterationen durchgeführt werden sollen. Deshalb wurde der Algorithmus leicht abgeändert. In jeder Iteration wird ein Automat zufällig bestimmt. Dann werden nur diejenigen Nachbarn generiert, die Verschiebungen des ausgewählten Automaten darstellen. Ein Eintrag in der Tabu-Liste enthält außerdem nur die Information, welcher Automat verschoben wurde und in welche Richtung der Automat verschoben wurde. Eine Nachbar ist genau dann tabu, wenn die inverse Operation in der Tabu-Liste auftaucht. So darf man beispielsweise einen Automaten X nicht zuerst um $(5|1)$ verschieben und im später um $(-5|-1)$ verschieben, solange sich diese Operation auf der Tabu-Liste befindet. Die Länge der Tabu-Liste wird auf ein Zehntel der Anzahl der Iterationen gesetzt.

Laufzeitkomplexität Das Laufzeitverhalten des Algorithmus hängt linear von der Anzahl der Iterationen, der Länge der Tabu-Liste und der Größe der Nachbarschaft ab. Wenn ein Automat den Radius r aufweist, besteht die Nachbarschaft aus maximal $4r \cdot 4r = 16 \cdot r^2$ Elementen. Um einen Nachbarn auf Gültigkeit zu überprüfen, sind dieses Mal nur a Tests notwendig, da nur ein einziger Automat verschoben wurde. Wenn die Größe der Tabu-Liste $\frac{1}{10} \cdot$ Iterationen beträgt, ergibt sich ein Laufzeitverhalten von $\mathcal{O}(\frac{1}{10}\text{Iterationen}^2 \cdot 16r^2 \cdot a)$.

Programm-Dokumentation Der Tabu-Algorithmus befindet sich in der Klasse `algorithmen.tabu` und implementiert, wie auch der Abkühlungs-Algorithmus, das Interface `informaticup.Algorithmus`. Die Methode `Optimiere` enthält den eigentlichen Algorithmus. Die Funktion `GeneriereNachbarschaftsLösung` wertet den aktuellen Zustand aus und generiert für einen zufällig ausgewählten Automaten die beste zulässige Lösung in der Nachbarschaft, die nicht tabu ist. Der Rückgabewert der Funktion ist eine Instanz der Klasse `Aenderungsvorschlag`, die nur die Änderung gegenüber dem aktuellen Zustand enthält. Es wird also nur gespeichert, welcher Automat um wie viele Pixel verschoben wurde. Zusammen mit dem aktuellen Zustand kann daraus der neue Zustand berechnet werden. Außerdem wird die Güte der des neuen Zustandes mit der Änderung gespeichert. Ist dieser Wert während der Berchnung einmal -1, so ist der Änderungsvorschlag entweder nicht zulässig oder tabu. Die Tabu-Liste ist eine verkettete Liste, die mit einem `ListIterator` in linearer Zeit durchsucht werden kann. Das Entfernen des letzten Elements und das Einfügen eines neuen Elements am Anfang ist in konstanter Zeit möglich.

6.7 Fest positionierte Automaten

Es besteht die Möglichkeit, dass der Benutzer Automaten selbst platzieren kann. Diese Automaten werden bei der Anwendung von Algorithmen (ausgenommen Backtracking) nicht gelöscht oder verschoben. Dazu gibt es in der Klasse `Automat` ein Feld `_automatGesperrt`. Wenn diese Variable auf `true` steht, wird der Automat von den Algorithmen nicht beachtet. Das heißt, vor jeder Operation auf den Automaten wird geprüft, ob der Automat gesperrt ist oder nicht. Falls er gesperrt ist, werden keine Änderungen vorgenommen. Natürlich werden gesperrte Automaten aber zur Gültigkeitsprüfung herangezogen. Es kann also nicht vorkommen, dass sich irgendein Automat mit einem gesperrten Automat überschneidet, es sei denn, der Benutzer platziert zwei gesperrte Automaten so, dass sie sich überschneiden.

7 Übersicht über die Programmoberfläche

Dieses Kapitel beschreibt, wie die GUI in Java technisch umgesetzt wurde und ist bewusst kurz gehalten, da sie für die meisten *Leser* wohl eher von uninteressant ist. Beim Entwurf der Oberfläche versuchten wir, die folgenden Grundsätze so gut wie möglich zu beachten.

- **Zweisprachigkeit:** Das Programm enthält eine englischsprachige sowie eine deutschsprachige Übersetzung der meisten GUI-Texte. Dies wurde über Ressourcen-Dateien (`informaticup.resources.*`) gelöst.
- **Falsche Bedienung vermeiden:** So weit es möglich ist, werden falsche Eingaben durch den Benutzer erst gar nicht erlaubt. So kann der Benutzer beispielsweise die Berechnung erst gar nicht starten, wenn er einen ungültigen Wert für die Temperatur eingibt. Außerdem kann er Automaten nicht so platzieren, dass eine ungültiger Zustand entsteht.
- **Trennung von GUI und Algorithmus:** Durch die Aufteilung in Packages und verschiedene Klassen ist uns das relativ gut gelungen. Lediglich die Klasse `informaticup.InformaticupView` enthält die Steuerung des Programmablaufes. Dort werden zum Beispiel Parameter für die Algorithmen ausgewertet und die einzelnen Berechnungsschritte (z.B. Algorithmen) angestösen. Dadurch konnten wir uns jedoch Arbeit sparen, weil oft auf Parameter-Werte aus der GUI zurückgegriffen werden muss.

7.1 Hauptfenster

Das Hauptfenster `informaticup.InformaticupView` ist die zentrale Steuerkomponente des Programms. Von dort aus können über das Menü neue Berechnungen gestartet werden und die Lösung kann in eine Textdatei exportiert werden.

Nach der Berechnung wird der Zustand der Welt mit allen Automaten und Stadtteilen auf ein `JPanel` gezeichnet. Dabei handelt es sich jedoch nicht um reguläres Panel, sondern um eine Erweiterung zum `informaticup.drawing-Panel`. Dieses modifizierte Panel enthält Methoden zum Zeichnen von Stadtteilen, Automaten und der Gewichtskarte. Ebenso reagiert das Panel auf Mauseingaben. Mit der Maus können neue Automaten erzeugt und bestehende Automaten verschoben, gelöscht und gesperrt werden. Bei diesen Aktionen ist es wichtig, die Mauskoordinaten in Pixelkoordinaten der Pixelkarte umzurechnen. Das Panel ermöglicht außerdem das Scrollen. Wenn das Fenster zu klein für die Karte ist, wenn der Slider zur Ausgabeskalierung nach rechts

geschoben wurde, kann an jede beliebige Position der Karte scrollen. Eine weitere wichtige Eigenschaft des Panels ist das automatische Neu-Zeichnen des Inhaltes, zum Beispiel wenn die Größe des Fensters verändert wurde.

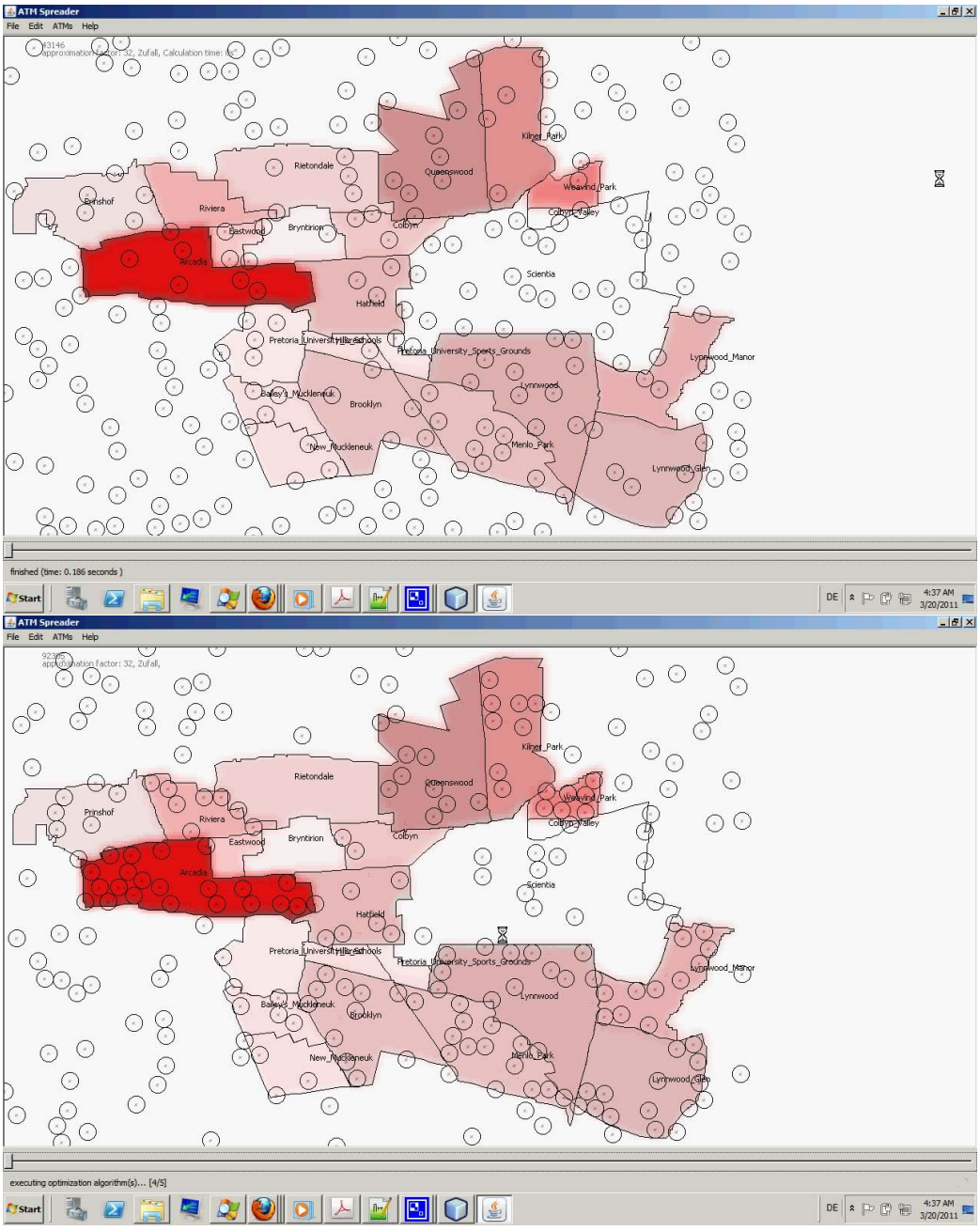
Über die Menüleiste des Hauptfensters kann der Benutzer außerdem verschiedene Modi der Lösungsbearbeitung aktivieren und deaktivieren (Automaten erstellen, löschen und sperren).

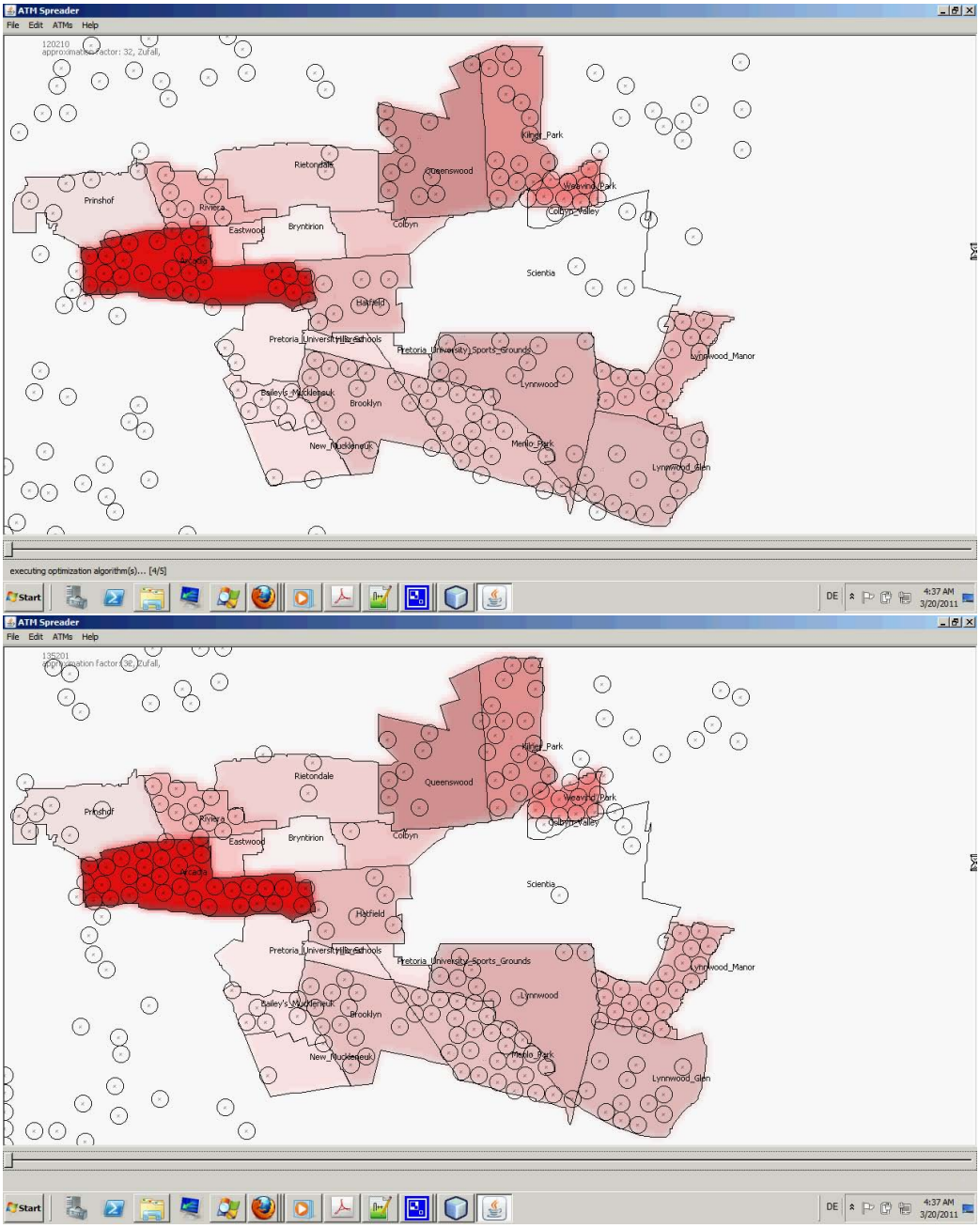
7.2 Nebenfenster

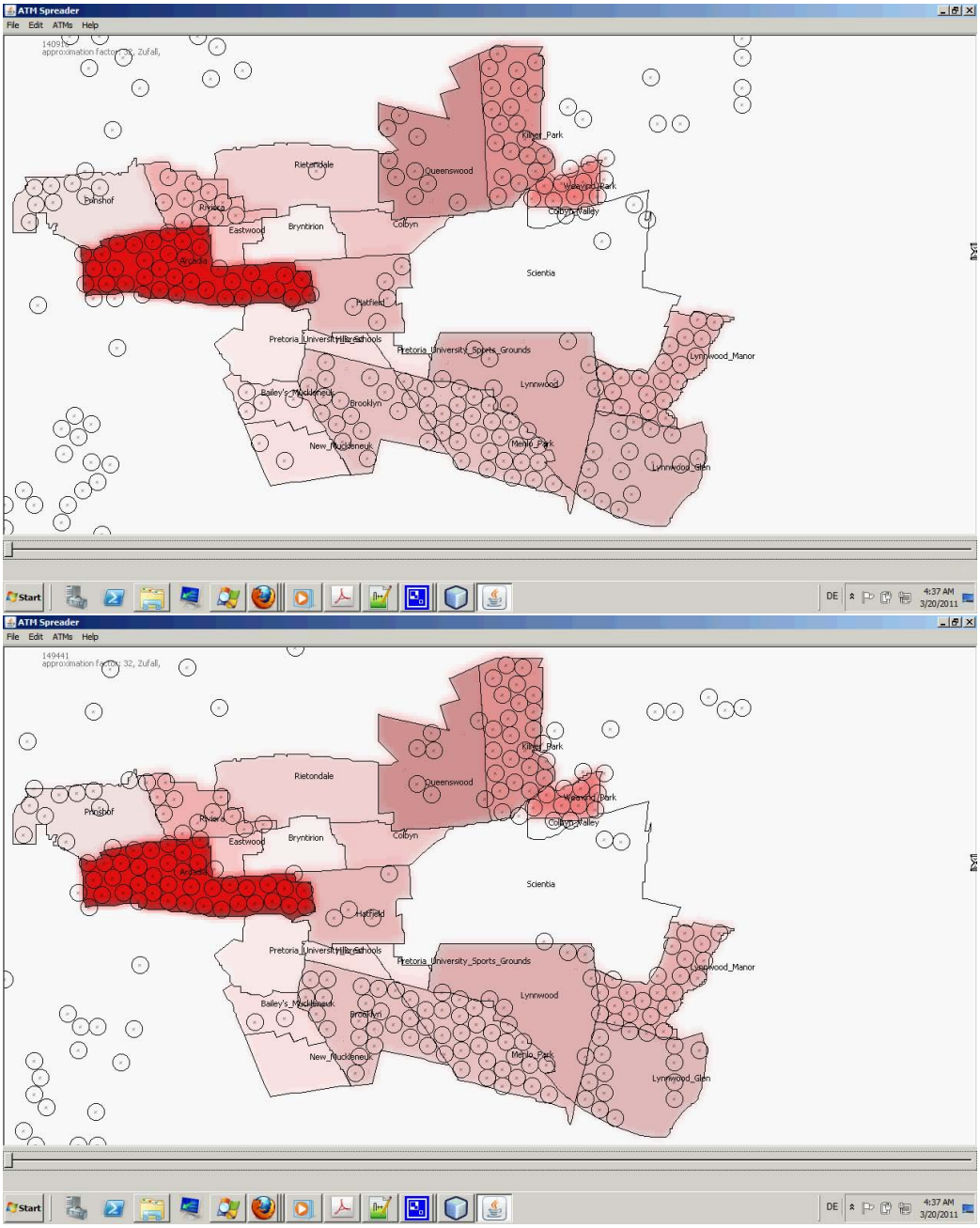
Neben dem Hauptfenster gibt es weitere Fenster, wie die About-Box mit den Namen der Autoren und Dialog mit den Parameter-Einstellungen für die Berechnung (Berechnungsdialog). Der Berechnungsdialog `informaticup.-createCalcFrame` enthält Textfelder, Optionsbuttons und Checkboxen für die Auswahl des Algorithmus und der Festlegung der Berechnungsparameter. Eine Tabelle `JTable` enthält die Gewichtungen der einzelnen Attribute, die der Benutzer frei wählen kann.

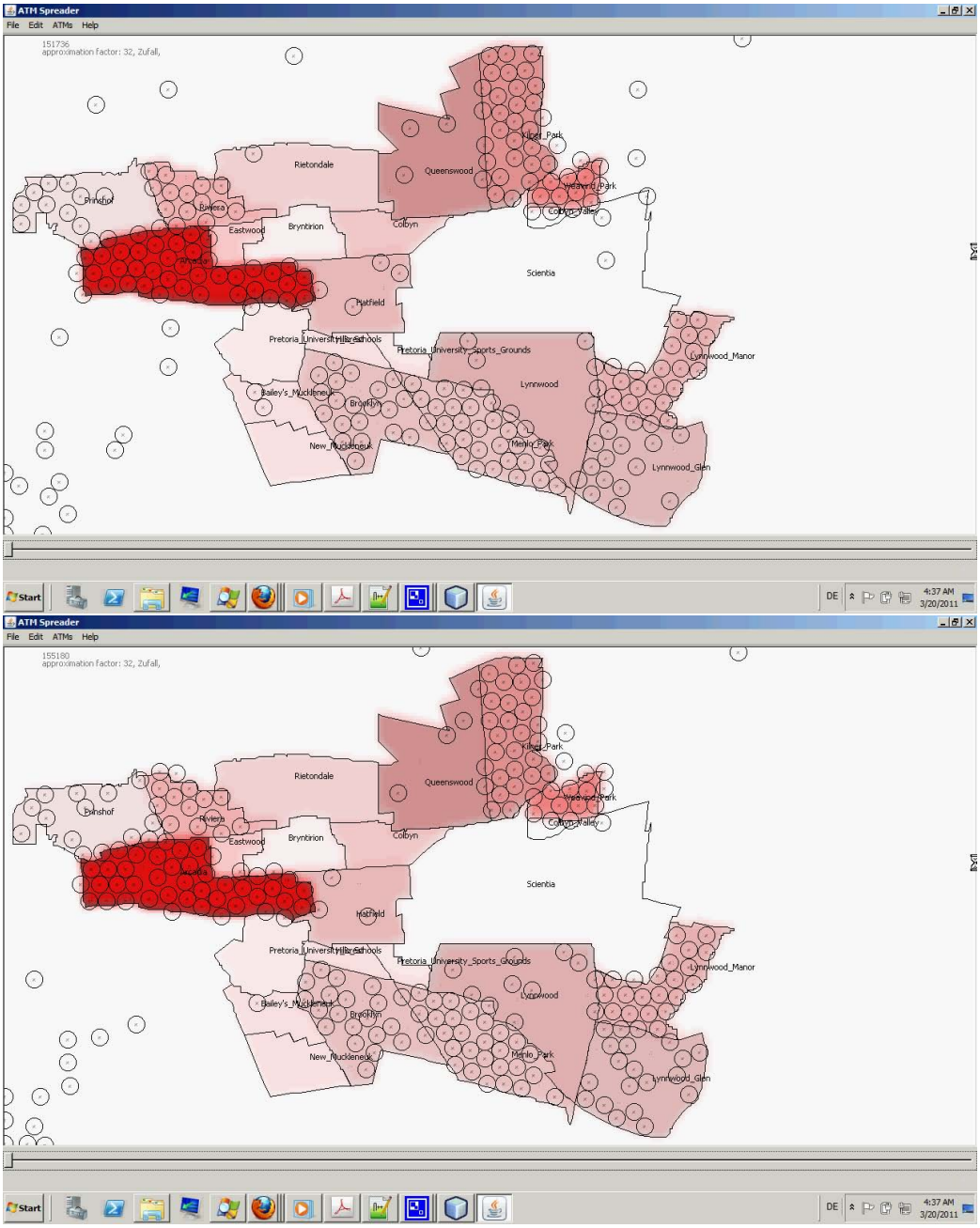
8 Beispielausgaben

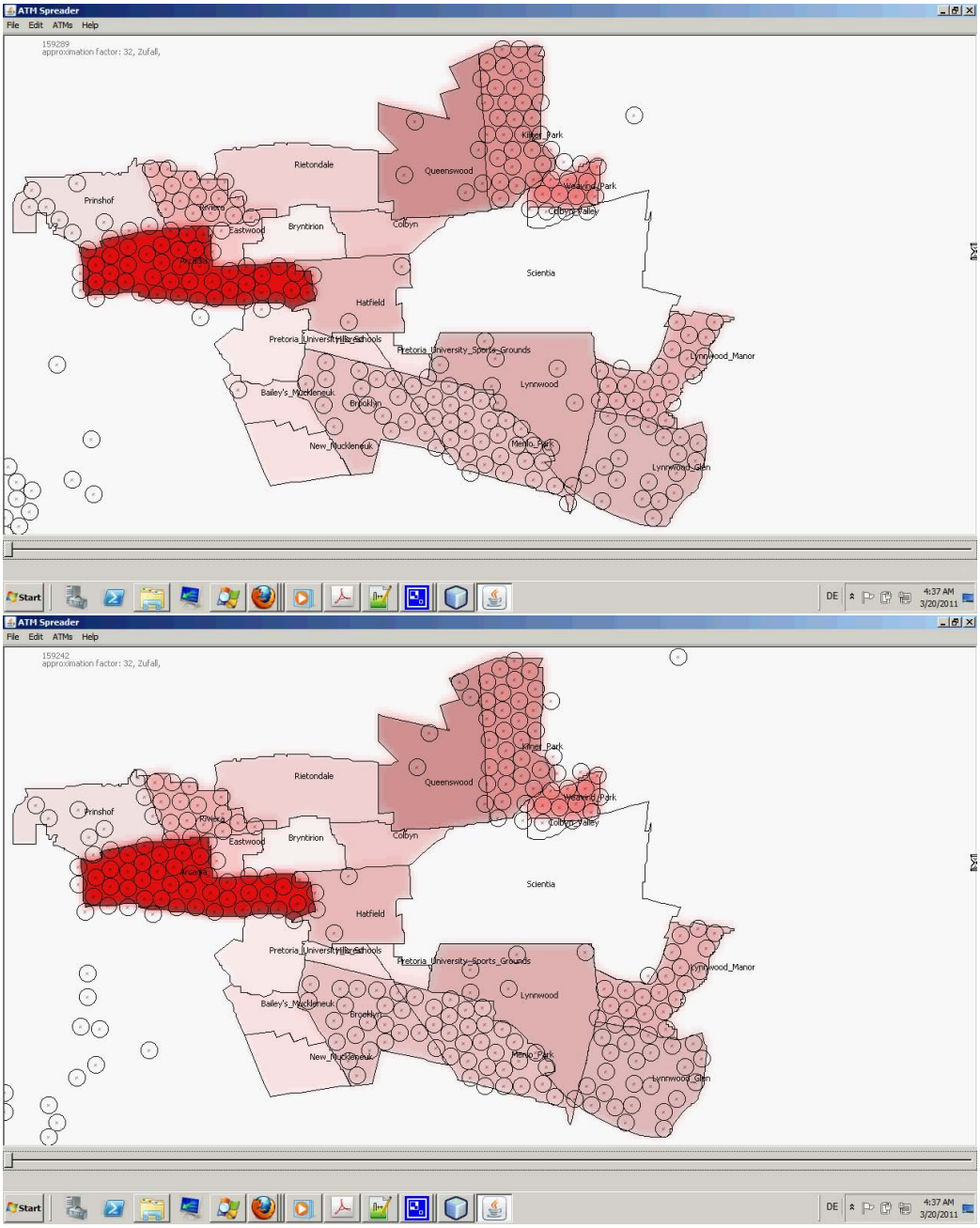
8.1 Tabu-Suche auf zufälliger Verteilung mit sehr vielen Automaten (`hatfield.txt`, 240 Automaten, 50000 Iterationen, Veränderungen durch den Algorithmus)

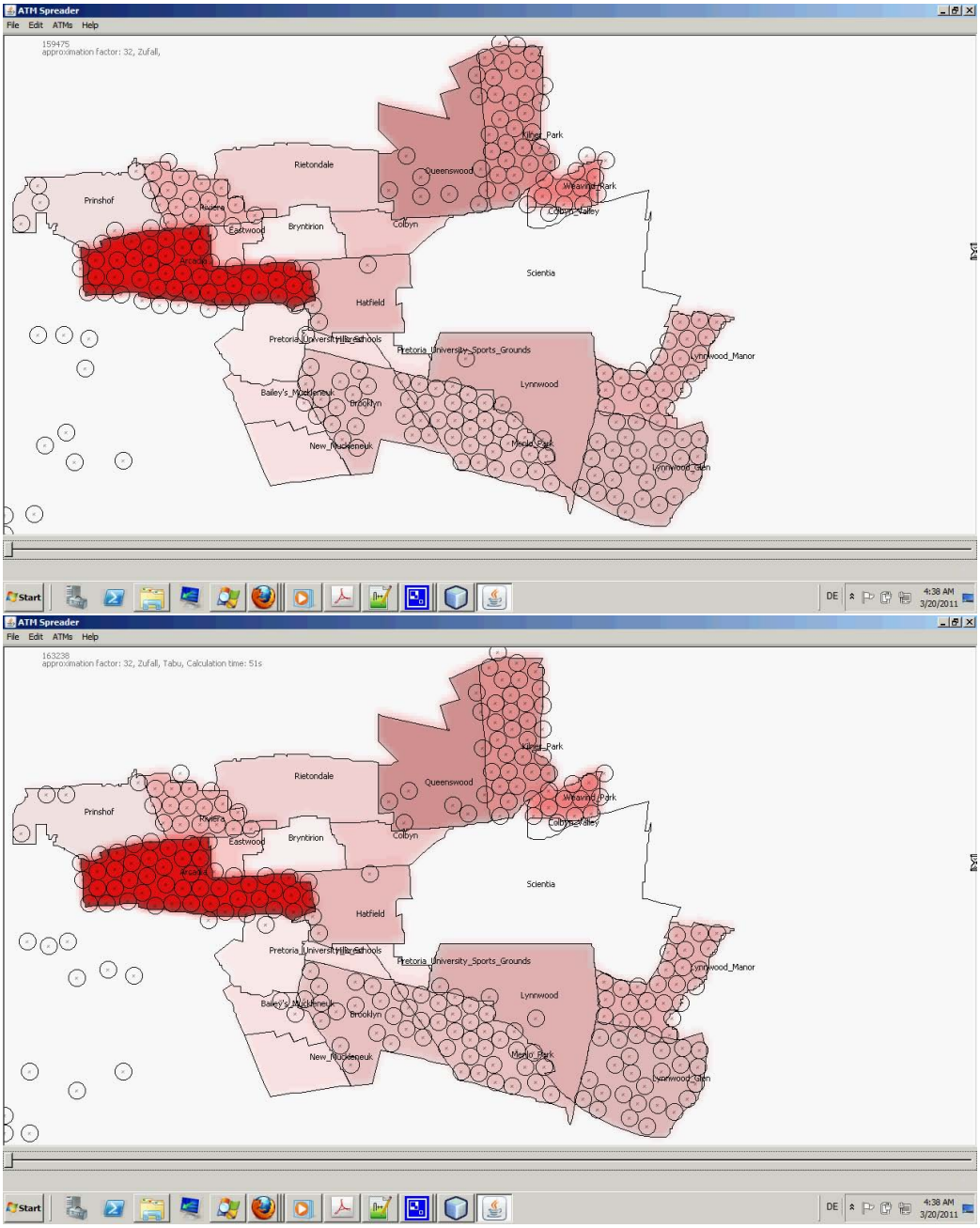



ATM Spreader
File Edit ATMs Help
executing optimization algorithm(s)... [4/5]
Rietondale
Queenswood
Prinshof
Riviera
Eastwood
Bryntirion
Colbyn
Scientia
Hatfield
Brooklyn
Lynnwood
Lynnwood_Manor
Lynnwood_Glen
Menlo_Park
ATM Spreader
File Edit ATMs Help

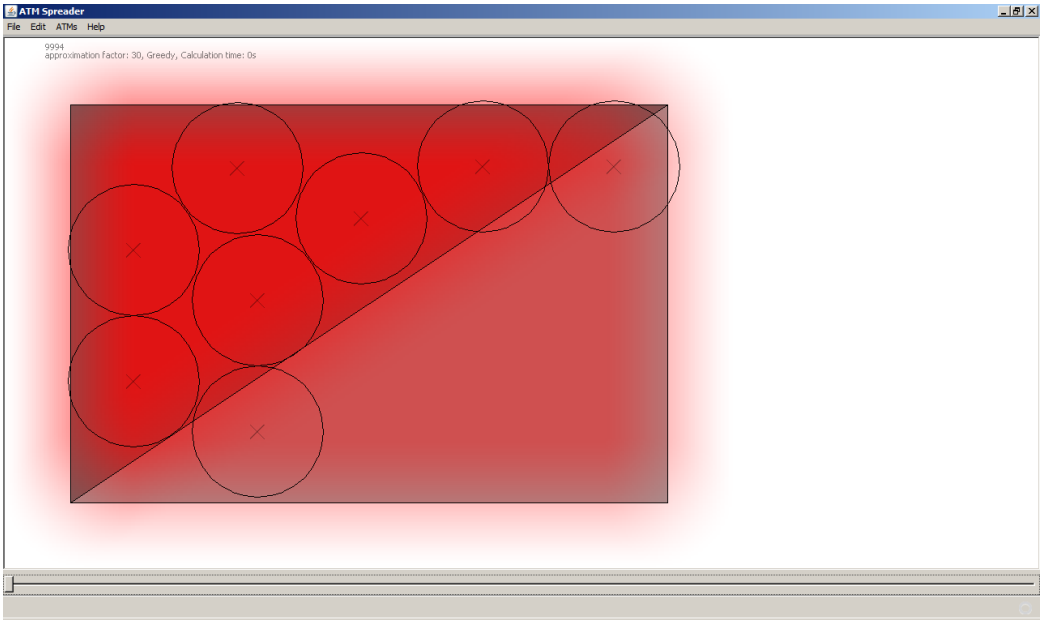

ATM Spreader
File Edit ATMs Help
Rietondale
Queenswood
Kilner_Park
Pinshof
Riviera
Eastwood
Bryntirion
Colbyn
Wierda_Park
Colbyn_Valley
Arcadia
Scientia
Hatfield
Pretoria_University_Sports_Grounds
Lynnwood_Manor
Lynnwood
Brooklyn
Menlo_Park
New_Muckleneuk
Lynnwood_Glen
DE
4:37 AM
3/20/2011


ATM Spreader
File Edit ATMs Help
151736
approximation factor: 32, 2ufall,
Rietondale
Queenswood
Pinehof
Riviera
Eastwood
Bryntirion
Colbyn
Scientia
Hatfield
Pretoria_University_Sports_Grounds
Lynnwood_Manor
Lynnwood
Brooklyn
Menlo_Park
Lynnwood_Glen
ATM Spreader
File Edit ATMs Help
155180
approximation factor: 32, 2ufall,
Rietondale
Queenswood
Pinehof
Riviera
Eastwood
Bryntirion
Colbyn
Scientia
Hatfield
Pretoria_University_Sports_Grounds
Lynnwood_Manor
Lynnwood
Brooklyn
Menlo_Park
Lynnwood_Glen

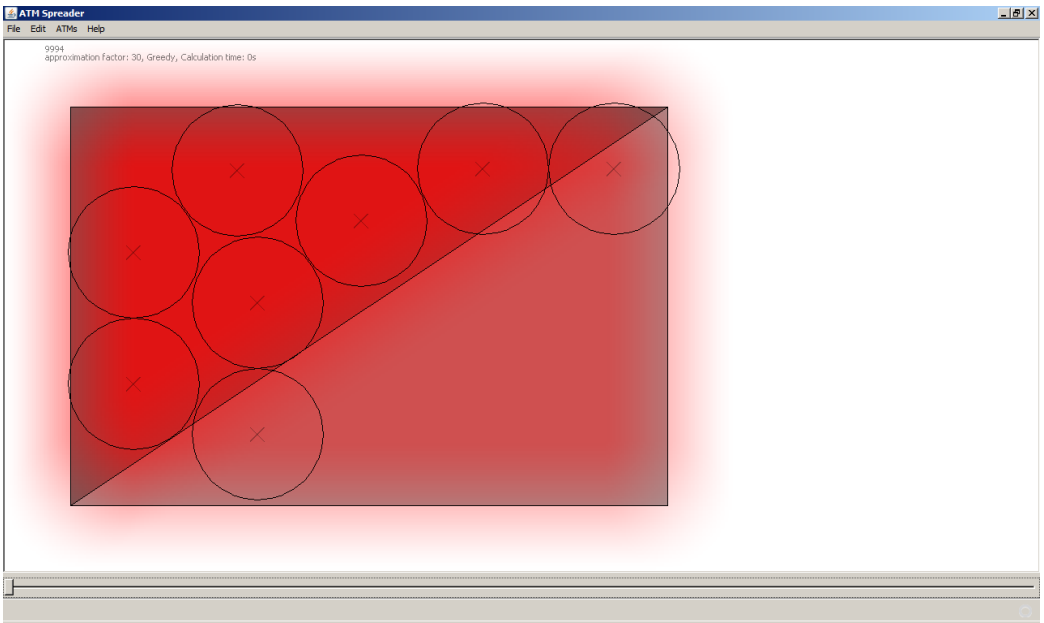

ATM Spreader
File Edit ATMs Help
159289
approximation factor: 32, 2ufall,
Rietondale
Queenswood
Prinshof
Eastwood
Bryntirion
Colbyn
Hatfield
Scientia
Pretoria_University_Sports_Grounds
Lynnwood
Lynnwood_Manor
Lynnwood_Glen
Menlo_Park
Brooklyn
ATM Spreader
File Edit ATMs Help
159242
approximation factor: 32, 2ufall,
Rietondale
Queenswood
Prinshof
Eastwood
Bryntirion
Colbyn
Hatfield
Scientia
Pretoria_University_Sports_Grounds
Lynnwood
Lynnwood_Manor
Lynnwood_Glen
Menlo_Park
Brooklyn


ATM Spreader
File Edit ATMs Help
159475
approximation factor: 32, Zufall,
Rietondale
Queenswood
Prinshof
Riviera
Eastwood
Bryntirion
Colbyn
Arcadia
Hatfield
Scientia
Pretoria_University_Sports_Grounds
Lynnwood_Manor
Lynnwood
Brooklyn
Menlo_Park
ATM Spreader
File Edit ATMs Help
163238
approximation factor: 32, Zufall, Tabu, Calculation time: 51s
Rietondale
Queenswood
Prinshof
Riviera
Eastwood
Bryntirion
Colbyn
Arcadia
Hatfield
Scientia
Pretoria_University_Sports_Grounds
Lynnwood_Manor
Lynnwood
Brooklyn
Menlo_Park

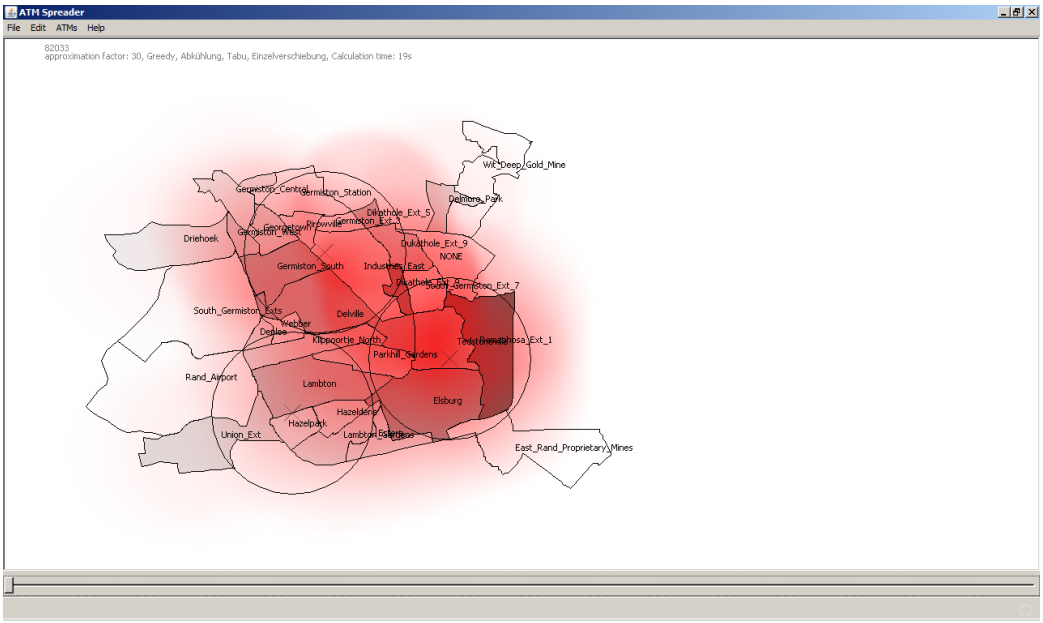
8.2 Greedy-Algorithmus (`example.txt`, 8 Automaten)



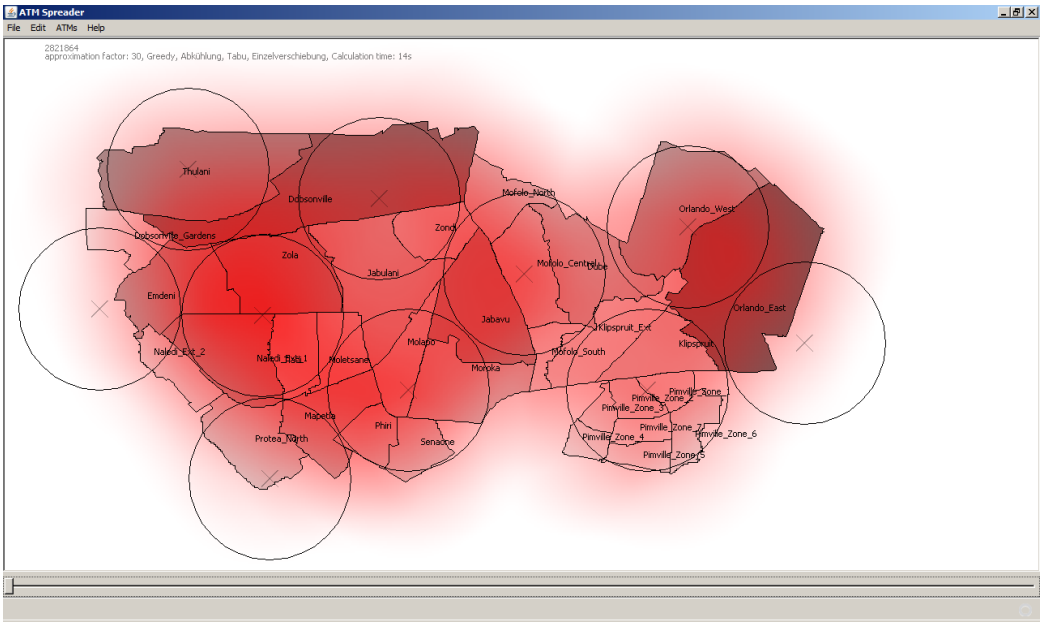
8.3 Greedy-Algorithmus und Simulierte Abkühlung (`example.txt`, 8 Automaten, T=1000)



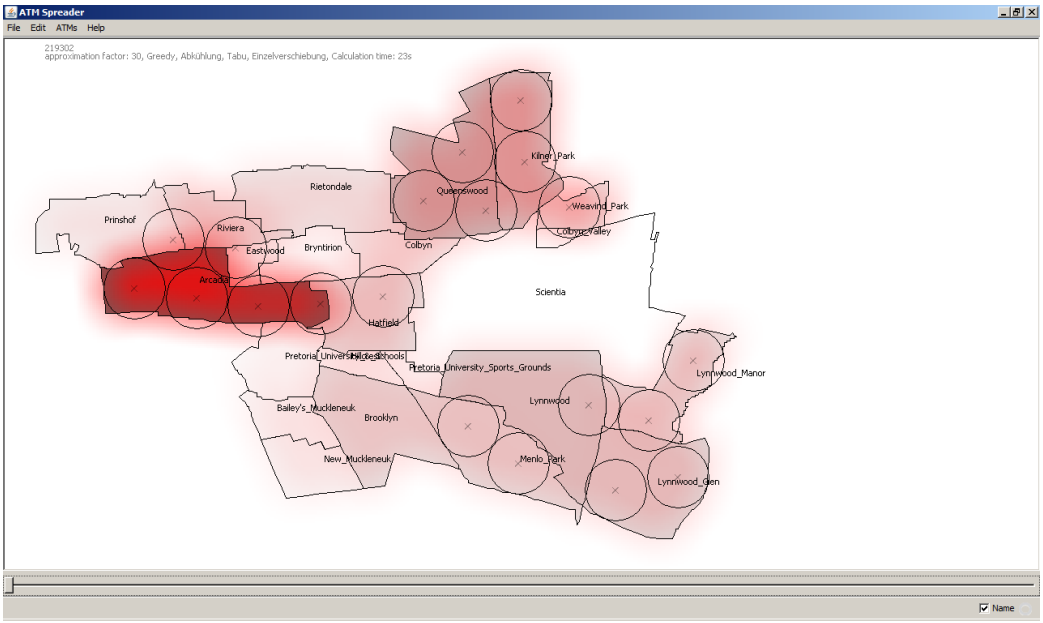
8.4 Greedy-Algorithmus und alle Optimierungsverfahren (`germiston.txt`, T=1000, 2500 Iterationen)



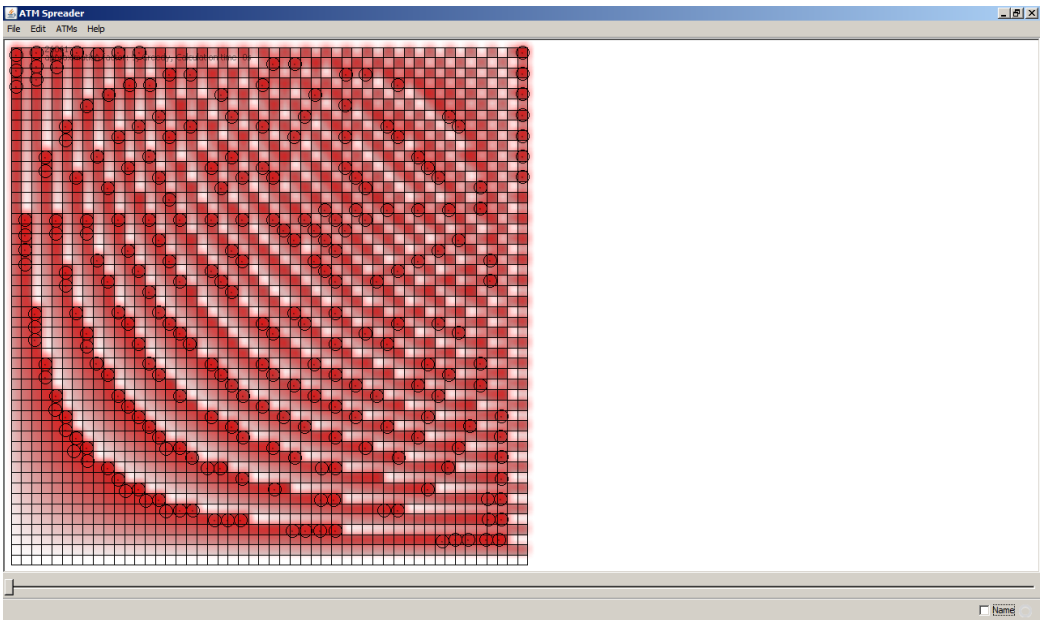
8.5 Greedy-Algorithmus und alle Optimierungsverfahren (`soweto.txt`, T=1000, 2500 Iterationen)



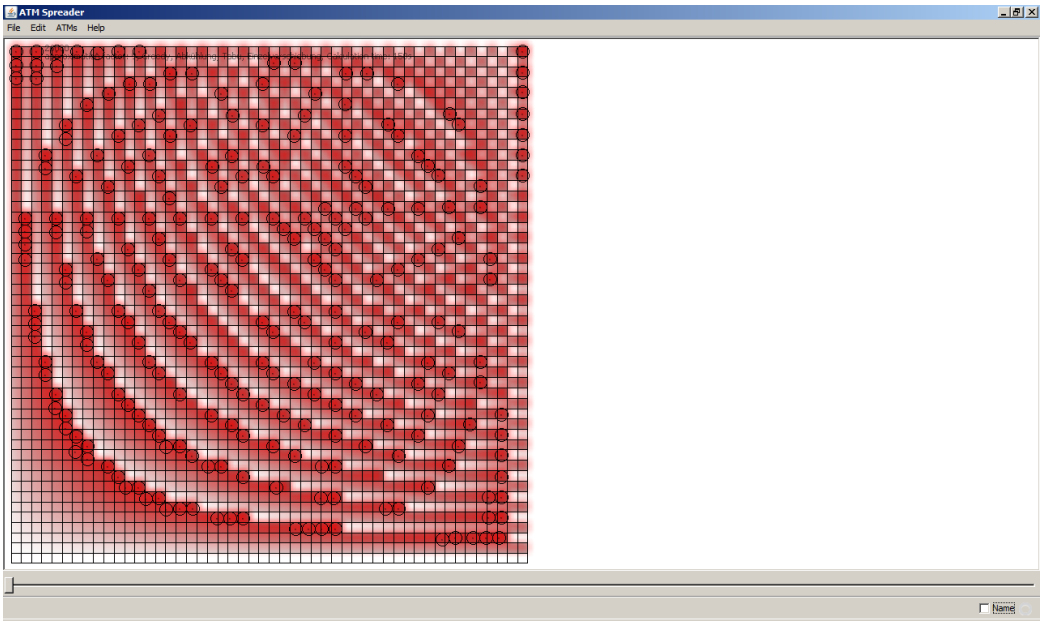
8.6 Greedy-Algorithmus und alle Optimierungsverfahren (`hatfield.txt`, T=2500, 5000 Iterationen)



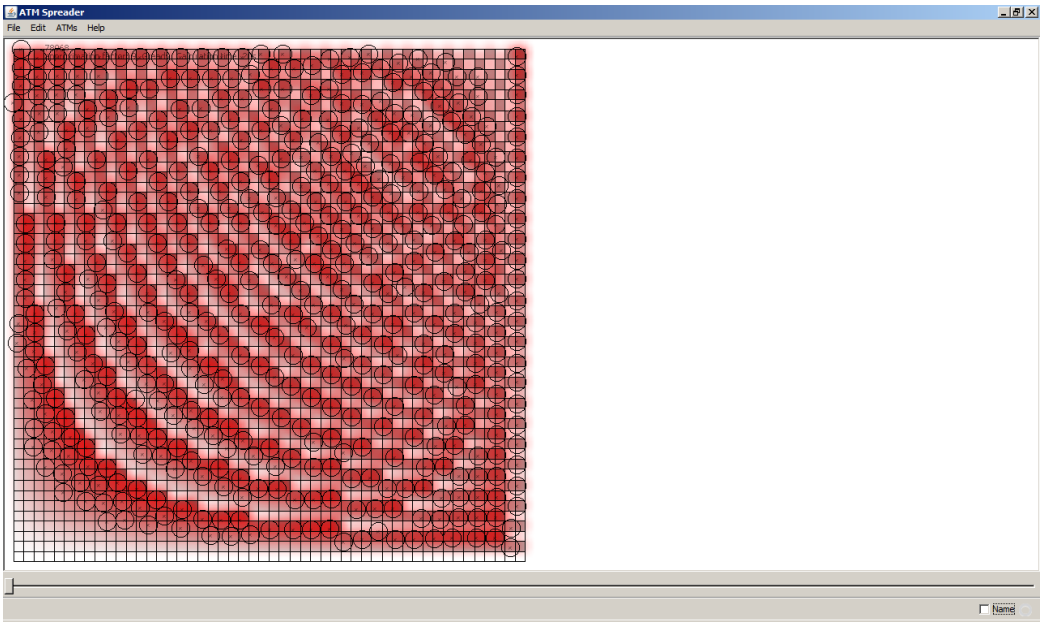
8.7 Greedy-Algorithmus (`crazy.txt`)



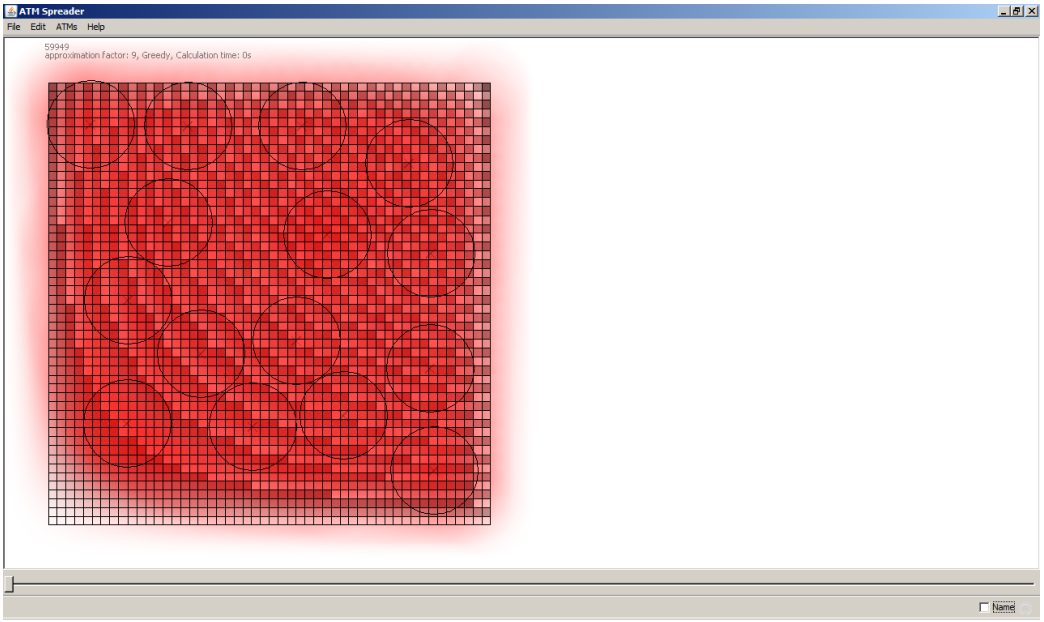
8.8 Greedy-Algorithmus und alle Optimierungsverfahren (`crazy.txt`, T=1000, 2500 Iterationen)



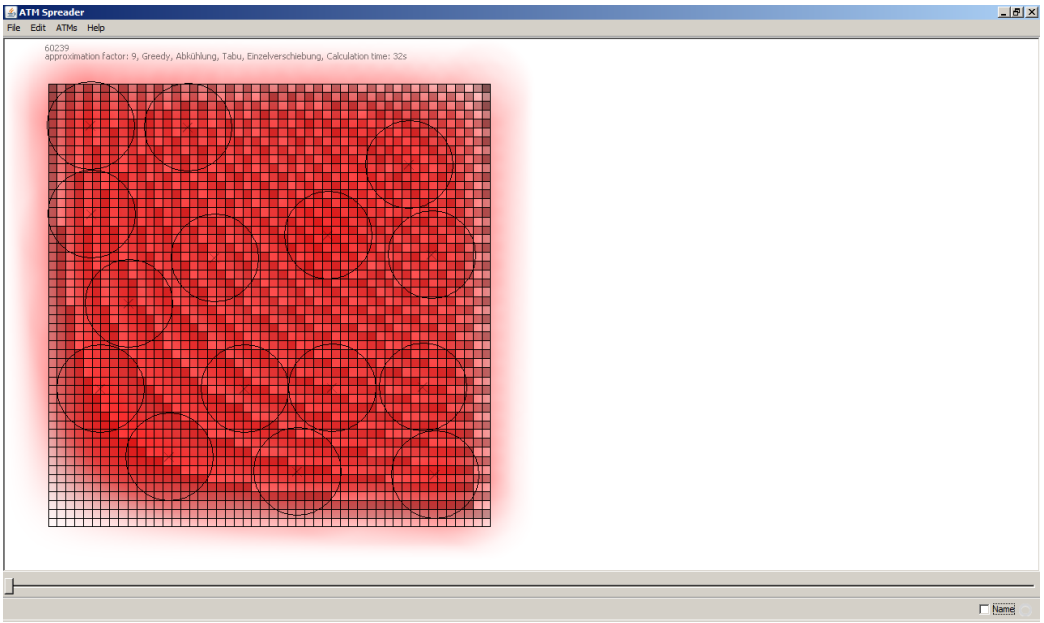
8.9 Greedy-Algorithmus (`crazy500.txt`, 500 Automaten)



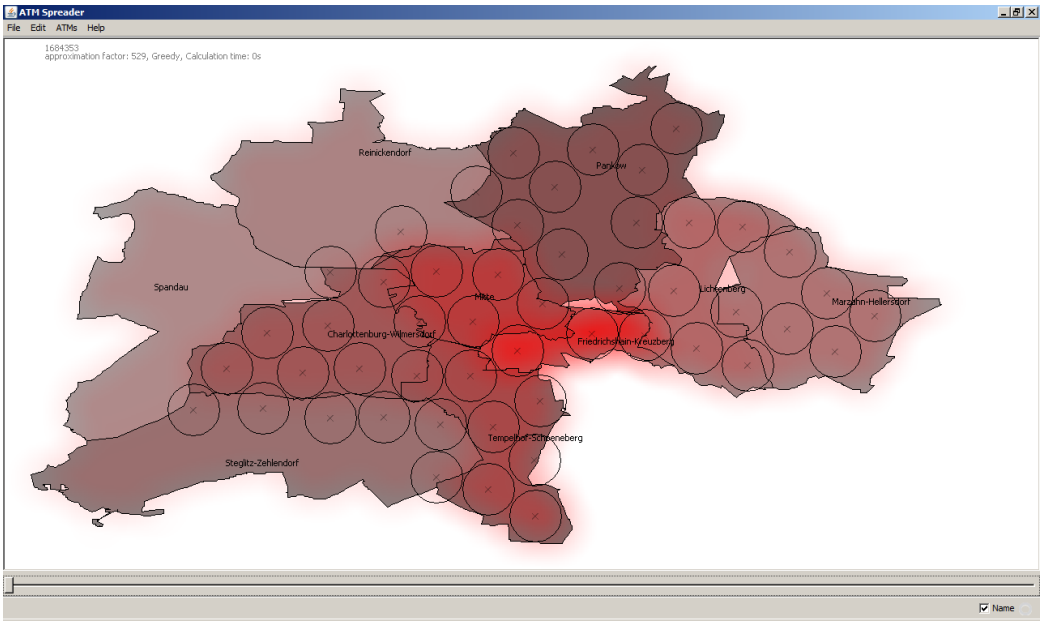
8.10 Greedy-Algorithmus (`crazy15.txt`, 15 Automaten)



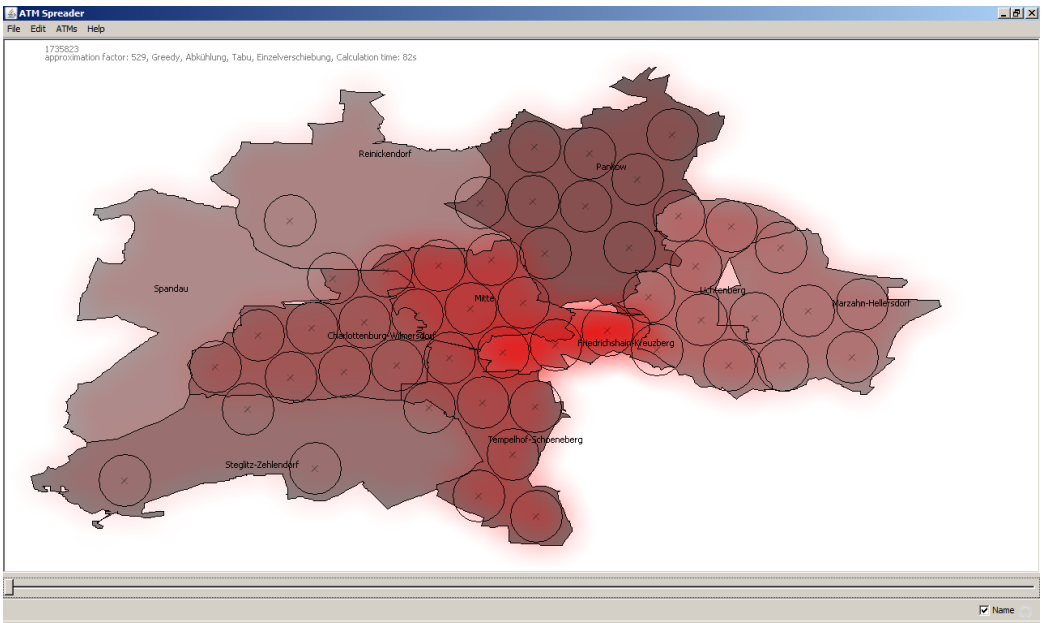
8.11 Greedy-Algorithmus und alle Optimierungsverfahren (`crazy15.txt`, 15 Automaten, T=1000, 2500 Iterationen)



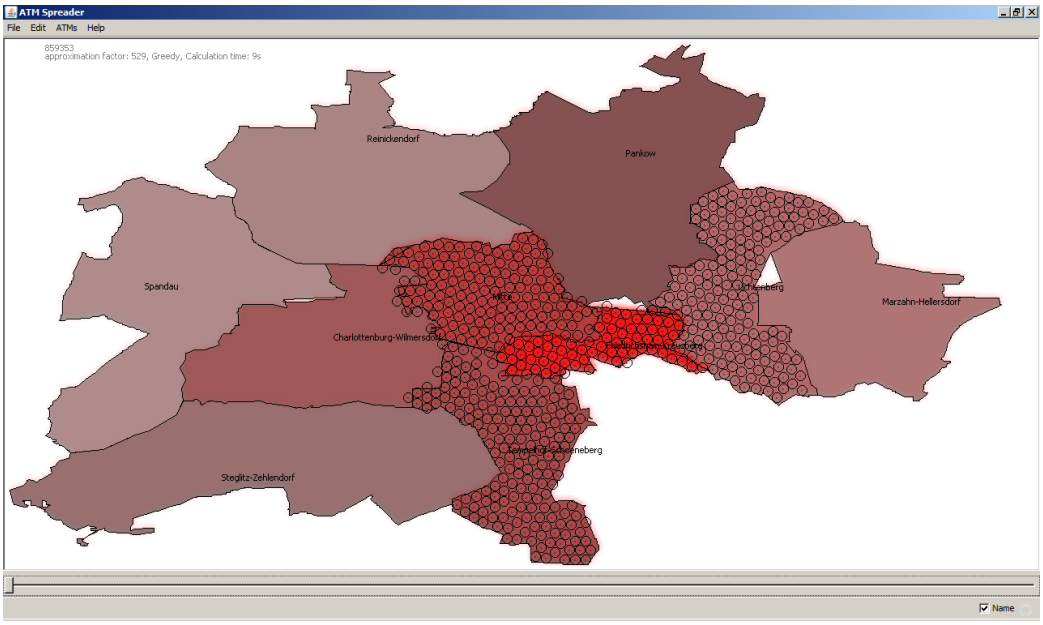
8.12 Greedy-Algorithmus (`berlin_50.txt`, 50 Automaten)



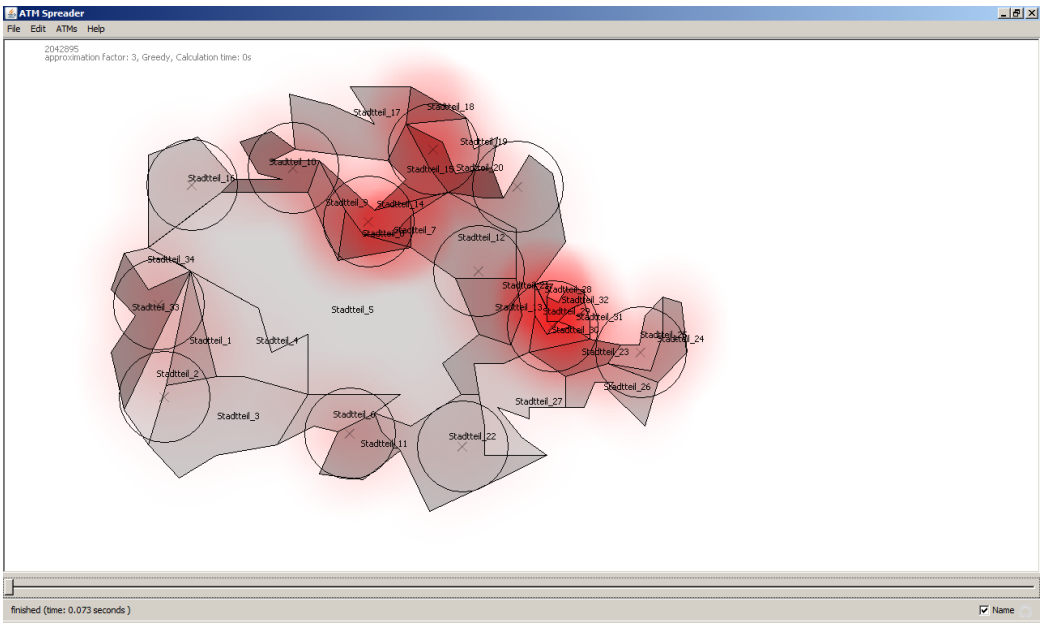
8.13 Greedy-Algorithmus und alle Optimierungsverfahren (`berlin_50.txt`, 50 Automaten, T=5000, 7500 Iterationen)



8.14 Greedy-Algorithmus (`berlin_580.txt`, 580 Automaten)



8.15 Greedy-Algorithmus (`eigene.txt`)



8.16 Greedy-Algorithmus und alle Optimierungsverfahren (`eigene.txt`, T=1000, 2500 Iterationen)