Experimentelle Verifikation von oberen und unteren Advice-Schranken für das Coloring-Problem auf bipartiten Graphen

Semesterarbeit

Florian Lüthi*

ZHAW, 22. November 2013

*luethifl@students.zhaw.ch

Inhaltsverzeichnis

1. Einleitung

2. Advice Complexity

2.1. Online- vs. Offline-Probleme

Optimierungsprobleme lassen sich (unter vielerlei anderen Klassifikationsmöglichkeiten) in Online-Probleme und Offline-Probleme mit dazugehörigen Online- bzw. Offline-Algorithmen unterteilen. Der wesentliche Unterschied zwischen diesen beiden Klassen besteht darin, dass ein Offline-Algorithmus seine gesamte Eingabe zur Laufzeit kennt, während ein Online-Algorithmus dies nicht tut. Er bekommt einen Teil der Eingabe und berechnet daraus sofort einen Teil des Resultats. Ein solches Teilresultat kann im Nachhinein nicht mehr geändert werden.

Es liegt nun auf der Hand, dass ein Online-Algorithmus dazu tendiert, während der Berechnung eines Teilresultats Entscheidungen zu treffen, welche sich nach Berechnungen weiterer Teilresultate als suboptimal herausstellen – dies insbesondere darum, weil die während der Berechnung des Teilresultats i zur Verfügung stehenden Informationen über das gesamte Problem nur aus den Teileingaben der Berechnungen 1 bis i stammen können [1].

Eine weitere Folge des obengenannten Unterschieds manifestiert sich darin, dass ein Online-Algorithmus im Allgemeinen offensichtlich keine optimale Lösung findet. Auf der anderen Seite funktioniert ein qualitativer Vergleich zweier Algorithmen dadurch, dass deren asymptotische Laufzeitkomplexität verglichen wird. Da sich die Qualität der von einem Online-Algorithmus und dessen Offline-Pendant produzierten Lösungen grundlegend unterscheidet (während letzterer eine optimale Lösung anstrebt, ist ersterem diese konzeptbedingt verwehrt), muss eine andere Möglichkeit gefunden werden, diese beiden Algorithmen (und deren Lösungen) vergleichen zu können.

2.2. Competitive Ratio

Eine gängige Möglichkeit, dieses Problem zu lösen, ist die von Sleator und Tarjan eingeführte *Competitive Ratio* [2, 1], welche im Grunde genommen das Verhältnis zwischen den Kosten der optimalen Lösung und deren des zu untersuchenden Online-Algorithmus ausdrückt [1]. Da die Qualität der Lösung eines Online-Algorithmus wesentlich von der Beschaffenheit der Eingabe abhängt, wird dabei vom jeweils schlechtestmöglichen Fall ausgegangen [3].

Formal kann die Competitive Ratio folgendermassen definiert werden: Sei $\text{opt}(I)$ die optimale Lösung eines konkreten Online-Problems für eine jeweilige Eingabe I. Seien

ferner $\mathcal{S}$ alle möglichen Lösungen (mit $\text{opt}(I) \in \mathcal{S}$), $A(I) \in \mathcal{S}$ diejenige Lösung, welche der Online-Algorithmus zu diesem Problem liefern kann, und $C : \mathcal{S} \to \mathbb{R}$ die Kosten für die jeweilige Lösung.

Ein Online-Algorithmus wird nun als *c-kompetitiv* bezeichnet, wenn es Konstanten $c \geq 0$ und α gibt, sodass für sämtliche mögliche Eingaben I

$$C(A(I)) \leq c \cdot C(\text{opt}(I)) + \alpha$$

gilt [1].

2.3. Das SkiRental-Problem

Die Competitive Ratio lässt sich durch das wohlbekannte SkiRental-Problem illustrieren: Nehmen wir an, wir wollen zum ersten Mal ein Wochenende lang Skifahren gehen. Wir besitzen aber keine Skis. Es stellt sich nun die Frage, ob wir Skis mieten oder kaufen wollen – wobei erschwerend hinzukommt, dass wir momentan noch keine Ahnung haben, wieviele Wochenenden wir danach noch skifahrend verbringen möchten.

Angenommen, der Kauf eines Paars Ski kostet CHF 500, die Miete äquivalenter Skis für ein Wochenende hingegen CHF 50.[1] Es ist nun einfach auszurechnen, dass sich der Kauf der Skis vor dem ersten Wochenende nur lohnen würde, wenn wir an mindestens 10 Wochenenden Skifahren gehen würden.

Da wir das aber nicht wissen, sind wir quasi der Online-Algorithmus, der für jede Teileingabe (sprich für jedes geplante Skiwochenende) sofort die Frage nach Kauf oder Miete beantworten muss. Die einzige Eingabe für den Algorithmus ist jeweils die Anzahl an vergangenen Skiwochenenden (der Einfachheit halber inklusive des aktuellen). Wir bezeichnen diese Eingabe hier als i.

Die oben besprochene Eigenschaft der potenziellen Suboptimalität der Entscheidungen wird beim zehnten solchen Wochenende augenscheinlich, wenn bei den vorangegangen Wochenenden die Entscheidung jeweils auf Mieten gefallen sein sollte.

Bezeichne t nun dasjenige Wochenende, an welchem wir uns dafür entscheiden, die Skis zu kaufen. Wir schauen uns die folgenden Fälle an:

- Für $t = 1$ gilt, dass

2.4. Zelluläre Automaten

> Ein zellulärer Automat ist eine regelmäßige Annordnung von Zellen. Jede Zelle kann eine endliche Zahl von Werten / Zuständen annehmen und hat eine

[1] In der Realität wesentliche Aspekte wie Preisveränderungen über die Zeit, veränderte Qualitätsansprüche und Abnützung der Skis sollen bei dieser Betrachtung vernachlässigt werden.

> begrenzte Zahl von Nachbarzellen, die sie beeinflussen können. Das Muster des gesamten zellulären Automaten ändert sich in einzelnen Schritten, die durch eine Reihe von Übergangsregeln bestimmt werden, die für alle Zellen gelten.[?]

Also:

Definition 1 (Zellulärer Automat)**.** Ein zellulärer Automat ist durch folgende Eigenschaften festgelegt:

- einen Zellularraum R,
- eine endliche Nachbarschaft N, wobei $\forall r \in R\,(N_r \subset R)$,
- eine Zustandsmenge Q,
- eine Überführungsfunktion $\delta : Q^{|N|+1} \mapsto Q$.

Die Zustandsübergänge erfolgen für alle Zellen nach derselben Überführungsfunktion und gleichzeitig. Die Zellzustände können wie die Zeitschritte diskret sein. [?]

Bemerkung. Aus dieser Definition folgt unmittelbar, dass der neue Zustand einer Zelle nur vom momentanen Zustand dieser Zelle sowie den Zuständen der Nachbarzellen abhängig sein kann.

Für zweidimensional organisierte R sind zwei Arten von Nachbarschaft üblich:

Von-Neumann-Nachbarschaft Die Nachbarschaft besteht jeweils aus den vier geographisch nächsten Nachbarzellen: $N_{i,j} = \{R_{i,j-1}, R_{i,j+1}, R_{i-1,j}, R_{i+1,j}\}$

Moore-Nachbarschaft Die Nachbarschaft besteht jeweils aus allen Zellen der Von-Neumann-Nachbarschaft sowie zusätzlich der diagonalen Nachbarzellen:

$N_{i,j} = \{R_{i,j-1}, R_{i,j+1}, R_{i-1,j}, R_{i+1,j}, R_{i-1,j-1}, R_{i-1,j+1}, R_{i-1,j+1}, R_{i+1,j+1}\}$ [?]

Beispiel (Wolfram's eindimensionales Universum)**.** Stephen Wolfram[2] definiert in [?] und in etlichen Arbeiten aus der Mitte der 1980er-Jahre einen parametrierbaren zellulären Automaten, der nur aus einer einzigen Raumdimension besteht[?]. In seiner einfachsten Ausprägung ist die Nachbarschaft N_i definiert als $\{R_{i-1}, R_{i+1}\}$, und jede Zelle kann genau zwei Zustände annehmen (tot und lebendig bzw. 0 und 1). Damit ist

$$\delta : \{0,1\}^{|\{R_{i-1},R_{i+1}\}|+1} \mapsto \{0,1\} = \{0,1\}^3 \mapsto \{0,1\}\,,$$

ergo existieren 8 mögliche Zustandsänderungen. Ein Beispiel:

Man stellt nun fest, dass bei 8 möglichen Zustandsänderungen für die Anzahl der möglichen Konfigurationen gilt:

$$|K| = |Q|^{|\,\mathrm{dom}(\delta)|} = 2^8 = 256.$$

alte $R_{i-1}R_iR_{i+1}$	111	110	101	100	011	010	001	000
neues R_i	0	1	1	0	1	1	1	0

Tabelle 2.1.: Wolfram-Konfiguration 110

111	110	101	100	011	010	001	000	Zahl
0	0	0	0	0	0	0	0	$00000000_b = 0_d$
0	0	0	0	0	0	0	1	$00000001_b = 1_d$
0	0	0	0	0	0	1	0	$00000010_b = 2_d$
					⋮			
0	1	1	0	1	1	1	0	$01101110_b = 110_d$
					⋮			
1	1	1	1	1	1	1	1	$11111111_b = 255_d$

Tabelle 2.2.: Alle Wolfram-Konfigurationen

Jeder dieser Konfigurationen kann eine natürliche Zahl $\{0, 1, 2, \ldots, 255\}$ zugeordnet werden, indem die neuen R_i wie oben tabelliert und pro Zeile als binäre Zahl aufgefasst werden [**?**]:

Dadurch ist es möglich, sämtliche eindimensionalen Wolfram-Universen durch eine einzige Zahl zu identifizieren.

Als besonders spannend hat sich die in Tabelle 2.1 dargestellte Konfiguration 110 erwiesen, weil sie die Eigenschaft hat, ein Turing-vollständiges System zu sein [**?**, **?**]. Dadurch ist sie die Konfiguration einer universellen Turingmaschine, die mit nur 2 Zuständen und 5 Symbolen umgesetzt werden kann – somit hat die Wolfram-Konfiguration 110 als Turingmaschine einen Umfang von $2 \cdot 5 = 10$ und zählt damit zu den kleinsten bis dato bekannten Turingmaschinen [**?**].

Beispiel (Game of Life)**.** Das von Conway[3] 1970 entworfene Game of Life ist eine bis heute populäre Umsetzung der Automatentheorie und insbesondere der Idee der zellulären Automaten [**?**].

Der ursprüngliche Entwurf befindet sich in einem zweidimensionalen R unter Verwendung der Moore-Nachbarschaft. Die Zellen können zwei mögliche Zustände $\{q_{\text{lebend}}, q_{\text{tot}}\}$

[2]Stephen Wolfram (* 29. August 1959), britischer Physiker und Mathematiker, Schöpfer der Software Mathematica sowie der Suchmaschine Wolfram Alpha [**?**]

[3]John Horton Conway (* 26. Dezember 1937), englischer Mathematiker[**?**]

annehmen, und die Übergangsfunktion ist definiert [**?**] als:

$$\delta(r,N) = \begin{cases} q_{\text{lebend}} & (r = q_{\text{tot}} \wedge \varsigma(q_{\text{lebend}}, N) = 3) \\ q_{\text{tot}} & (r = q_{\text{lebend}} \wedge \varsigma(q_{\text{lebend}}, N) < 2) \\ q_{\text{lebend}} & (r = q_{\text{lebend}} \wedge 2 \leq \varsigma(q_{\text{lebend}}, N) \leq 3) \\ q_{\text{tot}} & (r = q_{\text{lebend}} \wedge \varsigma(q_{\text{lebend}}, N) > 3) \\ q_{\text{tot}} & (\text{sonst}) \end{cases}$$

unter Zuhilfenahme der Statuszählfunktion

$$\varsigma(q,N) = \sum_{i=1}^{8} \begin{cases} 1 & (N_i = q) \\ 0 & (N_i \neq q) \end{cases}$$

2.5. Vom zellulären Automaten zur Differentialgleichung

Die Berechnung und Darstellung physikalischer Begebenheiten (allgemein ausgedrückt durch folgende partielle Differentialgleichung, mit $u : u(\vec{x}, t)$)

$$k_n \frac{\partial^n u}{\partial t^n} + k_{n-1} \frac{\partial^{n-1} u}{\partial t^{n-1}} + \cdots + k_1 \frac{\partial u}{\partial t} + k_0 = \frac{\partial^n u}{\partial \vec{x}^n} + \frac{\partial^{n-1}}{\partial \vec{x}^{n-1}} + \cdots + \frac{\partial u}{\partial \vec{x}}$$

mit zellulären Automaten kann durchgeführt werden, indem folgendes getan wird:

- R entspricht einer sinnvollen (groben) Diskretisierung der örtlichen Variablen $\vec{x}$ in einer, zwei oder drei Dimensionen
- Jede Zelle in R ist ein Tupel (Q, D) mit Q als einer Menge von berechnungsfernen Zustandsinformationen und den Differentialen nach der Zeit
$$D = \left(u, \frac{\partial u}{t}, \frac{\partial^2 u}{\partial t^2}, \cdots, \frac{\partial^n u}{t^n} \right) \in \mathbb{R}^n$$
- Eine neue Generation entspricht jeweils der fortgelaufenen Zeit ∂t, welche sehr fein diskretisiert werden muss
- In der Übergangsfunktion δ steckt die eigentliche Differentialgleichung. In der Regel verändert sie nur die Elemente von D. Sollte die Differentialgleichung Terme mit verschiedenen Ordnungen enthalten, wird die Gleichung unter Zuhilfenahme entsprechender Hilfsgleichungen $\lambda_1, \lambda_2, \ldots, \lambda_n$ in ein äquivalentes System gewöhnlicher Differentialgleichungen umgeformt.

2.6. Numerische Lösung von Differentialgleichungen

Damit die numerische Lösung von Differentialgleichungen gelingt, sind vor allem zwei Fertigkeiten vonnöten: Die numerische Differenzierung sowie die numerische Integration. Wir wollen beide Gebiete kurz streifen, und zwar unter den für uns nützlichen Gesichtspunkten bezüglich der Wahl der Variablen sowie der vorliegenden impliziten Funktionen (beziehungsweise diskreten Werten als Zellinhalte des zellulären Automaten).

2.6.1. Differenzierung nach dem Ort

Wir starten mit den gegebenen Werten $u_{\vec{x}}$ (den Zuständen der Zellen aus R). Daraus erhalten wir die diskrete Funktion

$$u : R \mapsto \mathbb{R}, u(\vec{x}) = u_{\vec{x}}$$

welche durch komponentenweises Einsetzen des Differenzenquotienten $\frac{\Delta u}{\Delta \vec{x}}$ folgendermassen differenzierbar ist (unter Zuhilfenahme des Einheitsvektors $\vec{e}_i$ für jede Komponente von $\vec{x}$ sowie der Erkenntnis, dass durch die diskrete Ausgangsfunktion $\Delta\vec{x}$ komponentenweise 1 ist; die Interpretation, wo genau sich die Nachbarzelle $\vec{x} \pm n \cdot \vec{e}$ befindet, überlassen wir der für den zellulären Automaten gültigen Nachbarschaftsfunktion):

$$\left(\frac{\Delta}{\Delta\vec{x}}u\right)_{\vec{x}} = u_{\vec{x}} - \sum_{i=1}^{\dim(\vec{x})} u_{\vec{x}-\vec{e}_i} \tag{2.1}$$

Diese Differenzen können natürlich wiederum differenziert werden:

$$\left(\frac{\Delta}{\Delta\vec{x}}\left(\frac{\Delta}{\Delta\vec{x}}u\right)_{\vec{x}}\right)_{\vec{x}} = \left(\frac{\Delta}{\Delta\vec{x}}u\right)_{\vec{x}} - \sum_{i=1}^{\dim(\vec{x})}\left(\frac{\Delta}{\Delta\vec{x}}u\right)_{\vec{x}-\vec{e}_i}$$

Durch Einsetzen von 2.1 in die obige Gleichung ergibt sich dann für die Differenz zweiter Ordnung:

$$\begin{aligned}
\left(\frac{\Delta^2}{\Delta\vec{x}^2}u\right)_{\vec{x}} &= u_{\vec{x}} - \sum_{i=1}^{\dim(\vec{x})} u_{\vec{x}-\vec{e}_i} - \sum_{i=1}^{\dim(\vec{x})}\left(u_{\vec{x^*}} - \sum_{i=1}^{\dim(\vec{x})} u_{\vec{x^*}-\vec{e}_i}\right)_{\vec{x^*}:=\vec{x}-\vec{e}_i} \\
&= u_{\vec{x}} - \sum_{i=1}^{\dim(\vec{x})} u_{\vec{x}-\vec{e}_i} - \sum_{i=1}^{\dim(\vec{x})} u_{\vec{x}-\vec{e}_i} + \sum_{i=1}^{\dim(\vec{x})} u_{\vec{x}-2\vec{e}_i} \\
&= u_{\vec{x}} - 2\cdot\sum_{i=1}^{\dim(\vec{x})} u_{\vec{x}-\vec{e}_i} + \sum_{i=1}^{\dim(\vec{x})} u_{\vec{x}-2\vec{e}_i}
\end{aligned}$$

Bemerkung (Indizes)**.** Da der Definitionsbereich der besprochenen Funktion diskret ist, verringert sich die Kardinalität des Definitionsbereichs der Differenzen mit fortschreitender Ordnung jeweils um 1 pro Dimension. Geographisch gesprochen, liegt die Differenz zweier Zellen ja eigentlich auf der gemeinsamen Kante dieser zwei Zellen. Daher ist es in der Praxis sinnvoll, die Differenzen gerader Ordnungen (bei deren Berechnung wie eben gesehen 3 Zellen pro Dimension involviert sind) jeweils in die Zelle in der Mitte zu schieben. Im Falle der zweiten Ordnung ergibt sich damit:

$$\left(\frac{\Delta^2}{\Delta\vec{x}^2}u\right)_{\vec{x}} = \sum_{i=1}^{\dim(\vec{x})} u_{\vec{x}-\vec{e}_i} - 2\cdot u_{\vec{x}} + \sum_{i=1}^{\dim(\vec{x})} u_{\vec{x}+\vec{e}_i} \tag{2.2}$$

2.6.2. Integration nach der Zeit mit Einschrittverfahren

Forderung

Wir starten mit den gegebenen Werten $\left(\frac{\Delta}{\Delta\vec{x}}u\right)_{\vec{x}}$ (den vorgängig berechneten Differenzen zwischen den Zellen). Wir definieren die allgemeine Integrationsfunktion

$$\mathbf{int} : \mathbb{R}^2 \mapsto \mathbb{R}$$

für welche jeweils gelten muss:

$$\mathbf{int}\left(\left(\frac{\Delta}{\Delta\vec{x}}u\right)_{\vec{x},t}, h\right) \approx \int_t^{t+h} u_{\vec{x}}\mathrm{d}t$$

Da die $u_{\vec{x}}$ zu jedem Zeitpunkt t (also pro Zeitschritt h beziehungsweise $\mathrm{d}t$) jeweils von ihren Nachbarzellen abhängen, ist es vonnöten, sämtliche Zellen für einen einzigen Zeitschritt simultan durchzurechnen.

Die impliziten Runge-Kutta-Verfahren

Da uns hier nur Einschrittverfahren interessieren, schauen wir uns im Detail die Klasse der Runge-Kutta-Verfahren[4][5] an, welche auf dem expliziten Euler-Verfahren[6] basieren und durch Butcher-Tableaux[7] allgemein definiert werden können.

Sämtliche Runge-Kutta-Verfahren können folgendermassen dargestellt werden:

$$y_{n+1} = y_n + h \cdot \sum_{j=1}^{s} \gamma_j \cdot k_j$$

wobei h die gewählte Schrittweite, γ_j die charakteristischen Koeffizienten des gewählten Verfahrens sowie k_j die Auswertungen der zu integrierenden Funktion f an bestimmten Stützstellen repräsentieren. Für k_j gilt dann:

$$k_j = f\left(t_n + h \cdot \alpha_j, y^n + h \cdot \sum_{i=1}^{m} \beta_{j,i} \cdot k_i\right)$$

wobei α_j sowie $\beta_{j,i}$ wiederum charakteristische Koeffizienten sind [**?**, **?**], welche zusammen mit den γ_j in einem Butcher-Tableau folgendermassen angeordnet werden können:

$$\left[\begin{array}{c|c} a & B \\ \hline & c \end{array}\right] = \left[\begin{array}{c|ccc} \alpha_1 & \beta_{1,1} & \beta_{1,2} & \cdots & \beta_{1,m} \\ \alpha_2 & \beta_{2,1} & \beta_{2,2} & \cdots & \beta_{2,m} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \alpha_m & \beta_{m,1} & \beta_{m,2} & \cdots & \beta_{m,m} \\ \hline & \gamma_1 & \gamma_2 & \cdots & \gamma_m \end{array}\right]$$

[4] Carl David Tolmé Runge (1856–1927), deutscher Mathematiker [**?**]
[5] Martin Wilhelm Kutta (1867–1944), deutscher Mathematiker [**?**]
[6] Leonhard Euler (1707–1783), Schweizer Mathematiker und Physiker [**?**]
[7] John Charles Butcher (*1933), neuseeländischer Mathematiker [**?**]

Runge-Kutta-Verfahren sind im Allgemeinen implizit, weil zur Bestimmung der k_j sowie y^{n+1} ein (mindestens lineares) $(m+1) \times (m+1)$-Gleichungssystem gelöst werden muss; dies verursacht natürlich entsprechende Kalamitäten. Solchartige Verfahren sind aber sehr stabil.[**?**]

Die expliziten Runge-Kutta-Verfahren

Ist es nun möglich, Koeffizienten zu finden, deren Matrix B strikte nilpotente untere Dreiecksgestalt hat, so spricht man von einem expliziten Verfahren, weil sich das Gleichungssystem einfach durch Rückwärtseinsetzen lösen lässt (sogar wenn es nichtlinear ist):

$$\left[\begin{array}{c|c} a & B \\ \hline & c \end{array}\right] = \left[\begin{array}{c|cccc} \alpha_1 & 0 & & & \\ \alpha_2 & \beta_{2,1} & 0 & & \\ \vdots & \vdots & \vdots & \ddots & \\ \alpha_m & \beta_{m,1} & \beta_{m,2} & \cdots & 0 \\ \hline & \gamma_1 & \gamma_2 & \cdots & \gamma_m \end{array}\right]$$

Für ausgewählte explizite Verfahren lassen sich nun sowohl Butcher-Tableau als auch die geforderte Integrationsfunktion **int** folgendermassen aufstellen:

Euler

$$\begin{aligned} \left[\begin{array}{c|c} a & B \\ \hline & c \end{array}\right] &= \left[\begin{array}{c|c} 0 & 0 \\ \hline & 1 \end{array}\right] \\ \Rightarrow \mathbf{int}\left(\left(\frac{\Delta}{\Delta\vec{x}}u\right)_{\vec{x},t}, h\right) &= u_{\vec{x},t} + h \cdot \left(\frac{\Delta}{\Delta\vec{x}}u\right)_{\vec{x},t} \end{aligned}$$

Runge-Kutta 2. Ordnung

$$\begin{aligned} \left[\begin{array}{c|c} a & B \\ \hline & c \end{array}\right] &= \left[\begin{array}{c|cc} 0 & 0 & \\ \frac{1}{2} & \frac{1}{2} & 0 \\ \hline & 0 & 1 \end{array}\right] \\ \Rightarrow \mathbf{int}\left(\left(\frac{\Delta}{\Delta\vec{x}}u\right)_{\vec{x},t}, h\right) &= u_{\vec{x},t} + h \cdot k_2 \\ &= u_{\vec{x},t} + h \cdot \mathbf{int}\left(\left(\frac{\Delta}{\Delta\vec{x}}u\right)_{\vec{x},t+\frac{1}{2}h}\right) \end{aligned}$$

2.6.3. Die homogene Wellengleichung

Am Beispiel der homogenen Wellengleichung soll nun die Umsetzung einer Differentialgleichung in einem zellulären Automaten gezeigt werden. Die homogene Wellengleichung

ist eine partielle Differentialgleichung zweiter Ordnung und lautet [**?**]:

$$\frac{1}{c^2}\frac{\partial^2 u}{\partial t^2} - \sum_{i=1}^{n}\left(\frac{\partial^2 u}{\partial x_i^2}\right) = 0$$

oder ein bisschen umgeformt ($\frac{1}{c^2} = \frac{1}{k}$) und die Einzelkomponenten $x_1, x_2, \ldots, x_n$ als Vektor geschrieben:

$$\frac{\partial^2 u}{\partial \vec{x}^2} = k \cdot \frac{\partial^2 u}{\partial t^2}$$

Nach diesem Schema soll zweimal nach dem Ort differenziert und zweimal nach der Zeit integriert werden:

$$\begin{array}{rl}
u_i & u_i(t+\Delta t) = u_i(t) + \dfrac{\partial u_i(t+\Delta t)}{\partial t}\Delta t \\
\downarrow & \uparrow \\
\dfrac{\partial u_i}{\partial \vec{x}} = u_i - \sum u_{i-1} & \dfrac{\partial u_i(t+\Delta t)}{\partial t} = \dfrac{\partial u_i(t)}{\partial t} + \dfrac{\partial^2 u}{\partial t^2}\Delta t \\
\downarrow & \uparrow \\
\dfrac{\partial^2 u_i}{\partial \vec{x}^2} = -2u_i + \sum u_{i-1} + \sum u_{i+1} \underbrace{\longrightarrow}_{\frac{\partial^2 u}{\partial t^2} = k \cdot \frac{\partial^2 u}{\partial \vec{x}^2}} & \dfrac{\partial^2 u}{\partial t^2}
\end{array}$$

Als Nachbarschaft wird die Von-Neumann-Nachbarschaft gewählt, weil bei der Moore-Nachbarschaft die Eckzellen nicht linear unabhängig wären. Ergo ergibt sich folgende Übergangsfunktion δ, abhängig von der Zelle r sowie den Nachbarzellen n, e, s, w (Norden, Osten, Süden, Westen für ein zweidimensionales Zellfeld):

$$\begin{aligned}
\delta(r,n,e,s,w) &= \begin{bmatrix} \delta_{\frac{\Delta r}{\Delta t}}(r,n,e,s,w) \\ \delta_{\frac{\Delta^2 r}{\Delta t^2}}(r,n,e,s,w) \end{bmatrix} \\
&= \begin{bmatrix} r_{\frac{\Delta r}{\Delta t}} + \delta_{\frac{\Delta^2 r}{\Delta t^2}}(r,n,e,s,w) \cdot \Delta t \\ \frac{\Delta^2 r}{\Delta \vec{x}^2} \cdot k \end{bmatrix} \\
&= \begin{bmatrix} r_{\frac{\Delta r}{\Delta t}} + \delta_{\frac{\Delta^2 r}{\Delta t^2}}(r,n,e,s,w) \cdot \Delta t \\ (-2r + \frac{1}{2}(n+e+s+w)) \cdot k \end{bmatrix}
\end{aligned}$$

Und damit sind wir im Grunde bereit für die Implementation.

3. Bemerkungen zur Implementation

3.1. Wahl des Umfelds

Die Wahl des Umfelds fiel auf eine Kombination aus JavaScript und dem HTML5-`canvas`-Element für die grafische Ausgabe. Die hauptsächlichen Gründe dafür sind die einfache Umsetzung sowie die sehr gute Portabilität, vorallem auch auf portable Devices mit Touch-Bedienung. Dadurch kommt der primäre Use Case der Software (Bewegung einer Flüssigkeit mittels dem Pointing Input) besonders gut zur Geltung, weil der Input eben mit dem Finger vorgenommen werden kann.

Bedenken wurden vorgängig und während der ersten Implementierungsphase vor allem in Bezug auf die Perfomance geäussert, konnten aber im Laufe des Projekts zerstreut werden, auch weil gezeigt werden konnte, dass Optimierungen möglich sind (paralleles Berechnen, Umstellung von Vektorgrafik-Operationen auf Bitmap-Manipulationen).

Als Alternativen waren das Game-Framework XNA sowie die 3D-LED-Installation in der Haupthalle des Zürcher Hauptbahnhofs in Betracht gezogen worden.

3.2. Implementatorische Paradigmen

Da JavaScript keine typisierte Sprache ist, ist ein formaler Aufbau à la Java weder möglich noch sinnvoll. Wir lehnen uns in unserem Stil dem Duck-Typing-Paradigma von Ruby an, damit sind wir inhärent polymorph. Die in den in den folgenden Abschnitten dargestellten UML-Diagrammen vorkommenden abstrakten Klassen oder Interfaces existieren darum nicht im realen Code, sondern dienen mehr der Veranschaulichung und Strukturierung der Diagramme.

Natürlich gibt es keine Regel ohne Ausnahme: Die Klasse `ATusk` existiert tatsächlich und wird über Prototyping gesubklasst – einfach damit wir das mal gemacht haben [**?**, **?**, **?**].

3.3. Architektur

Eine Übersicht über die grobe Architektur sei hier als Package-Diagramm skizziert:

3.4. Testing

Unit Testing mit JavaScript ist nicht ganz einfach – aber möglich und natürlich sinnvoll. Wenn die Unit Tests in einem tatsächlichen Browser auszuführen sind, weil sie beispielsweise mit dem Document Object Model interagieren, führt das zwangsläufig zu Mühsamitäten, vor allem bezüglich Continuous Integration.

Da allerdings das UI nur einen kleinen Teil unseres Programms ausmacht, schien es ratsam, vor allem die anderen Bereiche zu testen. Daraus ergibt sich die Möglichkeit, auf eine Implementation von JavaScript in einem nicht-interaktiven Kontext zu setzen. Die Wahl fiel auf Node.js, eine auf WebKit aufbauende Server-Implementation von JavaScript [**?**], sowie das Test-Framework Mocha [**?**]. Da das produktive Programm eine reine Browser-Geschichte ist, stehen während des produktiven Betriebs weder Build-Tools noch Server zur Verfügung, alle JavaScript-Files werden vom verwendenden HTML-Dokument über profane `<script>`-Tags eingebunden. Um das sich daraus ergebende Dilemma zwischen Test- und Produktionsumgebung zu lösen, werden die Komponenten über Node-Module wo nötig zusammengestöpselt, aber nur dann, wenn Node tatsächlich zur Verfügung steht (der Ausführer also ein Test Runner ist):

```
if (typeof(module) != "undefined") {
  module.exports = CellularAutomata;
  var Cell = require('../../src/automata/Cell.js');
  var Swimmer = require('../../src/automata/Swimmer.js');
}
```

Ein Unit Test (unter Verwendung der Should-Assertion Library) sieht dann beispielsweise folgendermassen aus:

```
describe('CellularAutomata', function() {

  it('should step without Tusk', function() {
    var auto = new CellularAutomata();
    auto.initCells();
    var iterations = auto.iterations;
    auto.step();
    auto.iterations.should.equal(iterations+1);
  });

});
```

Die Summe der Tests kann dann auf der Kommandozeile (oder durch einen CI-Server) aufgerufen werden mit:

```
~/node node_modules/mocha/bin/mocha
```

3.5. Automata

Die Klasse `CellularAutomata` implementiert zusammen mit ihren Hilfsklassen den zellulären Automaten:

`CellularAutomata` definiert zuallererst einmal eine coole Hilfsmethode, welche über alle Zellen des Automaten iteriert und eine beliebige Funktion ausführt:

```
this.forEachCell = function(fn) {
  for (var x = 0; x < this.cols; x++) {
    for (var y = 0; y < this.rows; y++) {
      var cell = this.model[x][y];
      fn(cell);
    }
  }
};
```

Diese Methode wird dann beispielsweise während der Initialisierung der Zellen folgendermassen verwendet[1]:

```
this.wireNeighbourCells = function() {
  if (this.tusk != null) {
    this.forEachCell(
      (function(automata) {
        return function(cell) {
          cell.neighbours = automata.tusk.getNeighbours(cell, automata.
              model);
        };
      })(this)
    );
  }
}
```

Des weiteren führt `CellularAutomata` die Zelltransitionen durch:

```
this.step = function() {
  if (this.tusk != null) {
    this.fireEvents();
    this.integration.integrate(this);
    this.travelSwimmers();
  }
  this.iterations++;
}
```

Spannend hierbei zu sehen ist die Aufteilung der Verantwortlichkeiten. `CellularAutomata` kennt weder die Übergangsfunktion δ, die Nachbarschaftsfunktion[2] noch die Integrati-

[1] Ein Wort zu der auf den ersten Blick unnötig komplizierten Definition der von `this.forEachCell()` auszuführenden anonymen Funktion: In JavaScript wird der Scope von Variablen durch die Funktion und nicht den Block begrenzt, darum gilt als Best Practice zur expliziten Scope-Gebung im Allgemeinen das Wrappen in einer anonymen Funktion und unmittelbarer Ausführung derselben [**?**]. Da `this` schlussendlich auch nur eine Variable in einem Funktionsscope ist, wäre `this.tusk` keine sinnvolle Referenz, weil `this` nicht bedingungslos (wie in anderen funktionalen Programmiersprachen üblich) eine Instanz von `CellularAutomata` (den Definitionsscope von `wireNeighbourCells()`), sondern der Caller ist (falls das Drücken eines Buttons zu einem Aufruf von `wireNeighbourCells()` führte, wäre `this` das Browser-`document`).

[2] Wie im Listing weiter oben ersichtlich, werden sämtliche Nachbarzellen sämtlicher Zellen während der Initialisierung aufgefunden und verknüpft. Da sich gemäss der Definition des Zellulären Automaten die Nachbarschaftsbeziehungen zwischen Zellen nie ändern, ergo von Anfang an feststehen, ist dies eine zulässige Performance-Optimierung; ansonsten müsste auf das Feld der Zellen bei jedem Iterationsschritt mindestens $|R| \cdot |N|$ mal zugegriffen werden (wobei R der ganze Zellraum und N die Menge der Nachbarzellen einer Zelle ist).

onsfunktion **int**, sondern verwendet via Strategy-Pattern die ihr zugeteilten Implementationen (beispielsweise die Wellengleichung mit dem Euler-Verfahren).

Es ist sogar so, dass der eigentliche Zellinhalt (der ja nur von der Übergangsfunktion δ verwendet wird) unbekannt ist. Jeder Tusk besitzt entsprechend eine Factory-Methode, um einen solchen Zellinhalt herzustellen. Die Unterschiede in den Datenstrukturen sind frappant. Das Game Of Life begnügt sich mit:

```
this.createCellData = function() {
  return {status: 0};
}
```

wogegen die Wellengleichung einige Daten mehr benötigt:

```
this.createCellData = function() {
  return {
    u: 0,
    udx: 0,
    udxdx: 0,
    udtdt: 0,
    udt: 0,
    vx: 0,
    vy: 0
  };
};
```

3.6. Integration

Die Wahl der Integrationsmethode ist wiederum eine Anwendung des Strategy-Patterns, wie in der Übersicht skizziert:

`integrate()` iteriert dann typischerweise über alle Zellen des Automaten und führt die Integration unter ein- oder mehrmaligem Aufruf der Übergangsfunktion δ des Tusk durch. Hier ist das Beispiel für das explizite Euler-Verfahren dargestellt:

```
this.integrate = function(automata) {
  var dt = 1 / automata.tusk.slices;
  for (var t = 0; t < automata.tusk.slices; t++) {

    automata.forEachCell(
      function(cell) {
        var differentials = automata.tusk.calcDifferentials(cell, dt,
            function(cell2) {
          return cell2.currentData;
        });
        cell.nextData = automata.tusk.applyDifferentials(cell, dt,
            differentials, function(cell2) {
          return cell2.currentData;
        })
      }
    );

    automata.forEachCell(
```

```
      function(cell) {
        cell.currentData = cell.nextData;
      }
    );

  }
}
```

3.7. Tusks, Regen, Enten und andere obskure Objekte

Die Tusks sind die Hard Worker in der ganzen Konstruktion:

3.7.1. Zellzustände und Pools

Der Aufbau der Zellstände sind nur für den Tusk von Bedeutung; die Verwender des Tusk (beispielsweise `CellularAutomata` weiss nichts darüber. Der Tusk implementiert die δ-Funktion (über `calcDifferentials()` und `applyDifferentials()`) und dient als Fassade für alle weiteren Zellzustands-Operationen:

createCellData() dient als Factory-Methode, um neue Zellzustände herzustellen;

setCellValue(Cell, value) setzt den u-Wert einer Zelle,

getVelocity(Cell) liefert den Geschwindigkeitsvektor einer Zelle (falls sinnvoll).

Des weiteren kann ein Tusk beliebig viele Pools definieren, die im Normalfall einen einzigen Aspekt des Zellzustands visualisieren sollen. Diese Pools können mit einem Namen und einem Bild versehen werden und tauchen dann in der Liste der Sekundär-Pools der View auf. Die Methode `pool.getValue()` liefert dann den Wert dieses Aspektes, wie zum Beispiel das erste Differential $\frac{\Delta}{\Delta \vec{x}} u$ in `DifferentialEquation`:

```
this.pools = [
  new Pool("du/dx", "_assets/pics/udx.png", function(cell) {
    return cell.currentData.dudx;
  })
];
```

Für den Primär-Pool für u gilt dasselbe:

```
this.primaryPool = new Pool("u", "", function(cell) {
  return cell.currentData.u; }
);
```

3.7.2. Swimmers

Zu einer ernstzunehmenden Simulation von Flüssigkeiten gehören natürlich schwimmende Objekte – im Idealfall sind das Badeenten. Diese Badeenten bewegen sich entlang der Gefälle der Oberfläche, sprich der Ableitungsvektoren entlang der Dimensionen

$\frac{\Delta}{\Delta x_1}u, \frac{\Delta}{\Delta x_2}u, \ldots, \frac{\Delta}{\Delta x_n}u$. Aus diesem Grund gibt es im Tusk die Methode `getVelocity(Cell)`, zum Beispiel für die Wellengleichung:

```
this.getVelocity = function(cell) {
  var v = new Vector();
  v.x = cell.currentData.vx;
  v.y = cell.currentData.vy;
  return v;
}
```

Die Badeenten (oder auch Fussbälle) ändern dann ihre Positionen anhand dieser Vektoren:

```
this.forEachSwimmer(
  (function(automata) {
    return function(swimmer) {
      var cell = automata.model[Math.floor(swimmer.location.y)][Math.
          floor(swimmer.location.x)];
      var velocity = automata.tusk.getVelocity(cell);
      swimmer.move(velocity);
    }
  })(this);
}
```

Zu guter Letzt wird noch der Darstellungswinkel der Schwimmer über ein paar Takte Trigonometrie (für zweidimensionale Zellräume) berechnet:

$$\varphi = \begin{cases} \tan^{-1}\left(-\frac{x}{y}\right) & \text{für } x \geq 0 \wedge y < 0, \\ \tan^{-1}\left(-\frac{y}{x}\right) + \frac{\pi}{2} & \text{für } x > 0 \wedge y \geq 0, \\ \tan^{-1}\left(-\frac{x}{y}\right) + \pi & \text{für } x \leq 0 \wedge y > 0, \\ \tan^{-1}\left(-\frac{y}{x}\right) + \frac{3}{\pi}2 & \text{für } x < 0 \wedge y \leq 0 \end{cases}$$

Die dritte Art von Schwimmern sind Boote (aus Holz), die über einen Motor verfügen und darum eine Eigengeschwindigkeit haben, die zur aus dem Gefälle resultierenden Geschwindigkeit gerechnet wird:

```
function AutoSwimmer() {
  this.move = function(d) {
    this.velocity
      = this.autoVelocity.clone().scale(0.8).add(d.scale(0.2));
    this.location.add(this.autoVelocity);
  }
}
```

3.7.3. Events

Events bringen die Natur zurück in die Informatik durch periodische Manipulation der Zellzustände. Die Tusks definieren, welche Events für sie sinnvoll sind. Hier die Übersicht:

`CellularAutomata` lässt dann vor jeder Iteration jedes Event ausführen:

```
this.fireEvents = function() {
  // fire events, such as rain, vorteces etc.
  if (this.tusk.events) {
    for (var i = 0; i < this.tusk.events.length; i++) {
      if (this.tusk.events[i].enabled) {
        this.tusk.events[i].apply(this);
      }
    }
  }
}
```

Wenn man beispielsweise Regen als gleichverteiltes Verändern der Wassermenge an einzelnen Punkten betrachtet, könnte man folgendes programmieren:

```
function RainEvent(applyCell) {

  this.dropsPerIteration = 4;
  this.sign = 1;

  this.apply = function(automata) {
    var drops = this.dropsPerIteration == 0 ? 0
      : this.dropsPerIteration >= 1 ? this.dropsPerIteration
      : automata.iterations
        % (Math.floor(1 / this.dropsPerIteration)) == 0 ? 1 : 0;

    for (var i = 0; i < drops; i++) {
      var x = Math.floor(Math.random() * automata.cols);
      var y = Math.floor(Math.random() * automata.rows);
      var cell = automata.model[x][y];
      this.sign = this.sign * -1;
      applyCell(cell, this.sign);
    }
  }
}
```

Da die Tusks die Events definieren (in diesem Beispiel so:)

```
this.events = [
  new RainEvent(
    function(cell, value) {
      cell.currentData.udt = value - cell.currentData.u;
      cell.currentData.u = value;
    }
  ),
  /* ... */
];
```

wissen die Events nichts von den Tusks, also geben die Tusks eine Funktion mit, mittels derer das Event die Zellzustände sinnvoll manipulieren kann (`applyCell`).

Des weiteren ist jedes Event unter Umständen in Aspekten konfigurierbar – im Beispiel des Regens die Intensität, dargestellt als Anzahl Tropfen pro Iteration. Die View weiss allerdings nichts davon, also braucht es hier eine Konstruktion, mit der das Event sein Konfigurations-UI dynamisch zur Verfügung stellen kann. Dies wird von der Methode `createView()` übernommen:

```
this.createView = function(doc) {
  var table = doc.createElement("table");
  var tr1 = doc.createElement("tr");
  var tr1Label = doc.createElement("td");
  var tr1Val = doc.createElement("td");
  var drops = doc.createElement("input");
  drops.type = "text";
  drops.style.width = "50px"
  drops.value = this.dropsPerIteration;
  drops.onchange = (function(evt, val) {
    return function() {
      evt.dropsPerIteration = val.value;
    }
  })(this, drops);

  table.appendChild(tr1);
  tr1.appendChild(tr1Label);
  tr1.appendChild(tr1Val);
  tr1Val.appendChild(drops);
  tr1Label.innerHTML = "Drops/Iteration";

  return table;
}
```

Sobald die View dann weiss, welcher Tusk aktiv ist (beispielsweise durch Auswahl durch den Benutzer), durchläuft die View alle Events des Tusks, ruft die entsprechende `createView()`-Methode auf und baut das zurückgegebene Element ins `document` ein.

3.8. View

Um auch das UI testbar zu halten, wird es hauptsächlich in eine `View`- und eine `Doc`-Klasse aufgeteilt, wovon die zweitere das HTML-`document` abstrahiert und für das Testing entsprechend zu mocken ist:

Die für die Interaktivität nötige Logik ist in den verschiedenen Controller-Klassen, hier als Anschauung ein Ausschnitt aus `TransportController`:

```
function TransportController(doc, view) {

  this.doc = doc;
  this.view = view;

  this.running = false;

  /* ... */

  this.step = function() {
    this.view.automata.step();
    this.view.paintAll();
    this.doc.iterationLabel.innerHTML = this.view.automata.iterations;
    if (this.running) {
      window.setTimeout((function(controller) {
```

```
        return function() {
          controller.step();
        };
      })(this), 0);
    }
  }

  /* ... */

}
```

3.8.1. Pool Canvas Painting

Die grafische Ausgabe des zellulären Automaten findet in einem `canvas`-Element statt, in das mittels JavaScript-Code gezeichnet wird. Es gibt verschiedene Arten, dies zu tun, ergo existieren verschiedene Implementationen:

Die relevanten Methoden sind:

`begin()` Beginnt den Zeichnungszyklus.

`paintCell(cell)` Zeichnet eine einzelne Zelle (`cell`) neu.

`paintSwimmer(swimmer, cell)` Zeichnet eine Ente (`duck`) in eine einzelne Zelle (`cell`).

`end()` Beendet den Zeichnungszyklus.

Die Methoden `begin()` und `end()` sind für Implementationen gedacht, die sich nur sinnvoll verhalten wenn sie den gesamten `canvas` neu bezeichnen können und deswegen nach einem atomaren Aufruf von `paintCell()` einen inkonsistenten Zustand aufweisen.

In den folgenden Abschnitten sind die verfügbaren Implementationen (ausser `NoopCanvasPainter`, welche trivial ist und verwendet werden kann, um die grafische Ausgabe komplett abzuschalten) grob erklärt.

`VectorCanvasPainter`

Diese Klasse bezeichnet den `canvas` durch Vektorgrafik-Operationen:

```
this.paintCell = function(cell) {
  var x = cell.x * this.scaling.x;
  var y = cell.y * this.scaling.y;

  var baseColor = this.baseColor;
  var color = ViewUtils.getFormattedColor(this.pool.getValue(cell),
      baseColor.r, baseColor.g, baseColor.b);

  this.context.fillStyle = color;
  this.context.fillRect(x, y, this.scaling.x, this.scaling.y);
};
```

`PixelCanvasPainter`

Diese Klasse bezeichnet den `canvas` durch direkte Pixel-Manipulationen. Grundsätzlich funktionert das folgendermassen [?]:

```
var myImageData = context.createImageData(cssWidth, cssHeight);
// ... do manipulation here ...
context.putImageData(myImageData, 0, 0);
```

Es muss allerdings gesagt werden, dass die Methode `putImageData()` in allen verfügbaren Implementationen (Gecko, WebKit usw.) einigermassen langsam ist. Dies wird vor allem damit erklärt, dass das zu manipulierende Pixel-Array (`myImageData.data` in unserem Beispiel) ein Array von Integern ist, das von `putImageData()` jeweils noch durchlaufen und auf Gültigkeit (Werte $\in [0:255]$) geprüft werden muss.

Andrew J. Baker zeigt nun in [?] einen Weg, durch den die Pixelmanipulationen drastisch beschleunigt werden können (solange die verwendete Implementation den Typ `Uint8ClampedArray` unterstützt, wie vom HTML5-Standard [?] eigentlich vorgesehen).

Dazu ist es erst einmal nötig, in `begin()` die Pixel-Daten vorzubereiten:

```
this.begin = function() {
  this.imageData = this.context.createImageData(WIDTH, HEIGHT);
  this.buf = new ArrayBuffer(this.imageData.data.length);
  this.buf8 = new Uint8ClampedArray(this.buf);
  this.data = new Uint32Array(this.buf);
};
```

Dadurch ist es möglich, die in `this.data` enthaltenen Daten als ganze Pixel zu bearbeiten (und nicht den Alpha-, Rot-, Grün- und Blau-Kanal einzeln):

```
this.paintCell = function(cell) {
  var x = cell.x * this.scaling.x;
  var y = cell.y * this.scaling.y;

  var baseColor = this.baseColor;
  var color = ViewUtils.getColor(this.pool.getValue(cell), baseColor.r,
      baseColor.g, baseColor.b);

  for (var ix = x; ix < x + this.scaling.x; ix++) {
    for (var iy = y; iy < y + this.scaling.y; iy++) {
      var p = (iy * this.view.CANVAS_WIDTH + ix);

      this.data[p] =
        (255   << 24) | // alpha
        (color.b << 16) |
        (color.g <<  8) |
        color.r;
    }
  }
};
```

Abschliessend werden dann die manipulierten Pixel wieder zurückgeschrieben:

```
this.end = function() {
  this.imageData.data.set(this.buf8);
```

```
  this.context.putImageData(this.imageData, 0, 0);
};
```

Ein Problem bleibt aber noch: By design exponiert `Uint32Array` die exakten Daten so wie sie im Memory liegen – das heisst, die obige Implementation funktioniert nur für Geräte, die mit Little Endian kutschieren. Bei der Ausführung auf Big Endian-Geräten würden die Farben ein bisschen lustig aussehen, weil die Werte in vertauschter Reihenfolge gesetzt werden müssten:

```
this.data[p] =
  (color.r   << 24) |
  (color.g << 16) |
  (color.b <<  8) |
  255; // alpha
```

Aus Mangel an Testgeräten (selbst iPhones sind Little Endian) wird aber davon abgesehen, dies auch noch zu implementieren (obschon das Testen auf die Endianness gar nicht mal so eine Sache wäre). [**?**]

3.8.2. Strategie, Strategie, Strategie...

Wir haben uns nun einiges an Theorie und noch mehr an Code zu Gemüte geführt – und meistens haben wir gesehen, dass mehrere Wege zum Ziel führen, beziehungsweise es mehrere richtige Lösungsansätze gibt. Deshalb ist das Strategy-Pattern eine coole Sache. Damit es aber schlussendlich nicht an der Darstellung scheitert, wird noch ein Satz Hilfsmethoden implementiert, welche der View helfen sollen, mit den verschiedenen Strategien umgehen zu können. Als Beispiel sei hier die Methode gelistet, die alle Implementationen in eine ComboBox einfüllen kann und entsprechend reagiert, wenn der Benutzer eine auswählt:

```
bindStrategiesToCombobox: function(id, combobox, strategies, text,
    onchange) {
  ViewUtils.clearStrategies(id);
  for (var strategy in strategies) {
    var opt = document.createElement("option");
    opt.text = text(strategies[strategy]);
    opt.value = strategy;
    combobox.options.add(opt);
    ViewUtils.addStrategy(id, opt);
  }
  combobox.onchange = function() {
    var strategy = strategies[combobox.value];
    onchange(strategy);
  };
}
```

Entsprechend existieren solche Methoden für alle möglichen Darstellungsarten von Strategien.

4. Schlussfolgerung

4.1. Erweiterungsmöglichkeiten

Wir haben lange über Multithreading nachgedacht, um die heutzutage üblichen Multi Core-Geräte sinnvoll auszulasten. Das ist grundsätzlich mittels WebWorker-Spawning möglich [**?**]; man würde dann den Zellraum geografisch oder modular auf die WebWorkers aufteilen und so den einzelnen Zeitschritt parallel rechnen. Das würde darum funktionieren, weil die Resultate nur von den Resultaten des vorhergehenden Zeitschritts abhängig sind. Die Hauptproblematik sehen wir jedoch darin, dass die WebWorkers (v/o Threads) über kein gemeinsames Memory verfügen. Es müsste also jedes Mal der gesamte Zellraum hin- und herkopiert werden, was in uns grosse Bedenken bezüglich der Overhead-Performance weckt. Ausserdem würde das einigermassen tief in die momentane Architektur eingreifen, weil der Memory-Transfer über JSON-Serialisierung funktioniert, welches (aus nachvollziehbaren Gründen) keine zirkulären Objektreferenzen zulässt – aber genau das haben wir durch die Vorverlinkung der Nachbarzellen natürlich en masse. Chrome hat zwar ein eigenes Verfahren entwickelt über Ownership-Transfer [**?**], aber das ist halt nicht kompatibel zum Rest der Welt. Ausserdem wird dabei – wie der Name sagt – das Objekt verschoben. Da in einem zellulären Automaten aber jede Zelle der Nachbar von irgendeiner anderen Zelle ist und sie ergo in der δ-Funktion des Nachbarn benötigt wird, muss sie geklont werden (weil sie in verschiedenen Threads gleichzeitig verwendet werden möchte), und wir sind so schlau als je zuvor. Vielleicht ist die Zeit einfach noch nicht reif für diese Erweiterung.

Eine sinnvolle Erweiterung wäre aber natürlich die Generalisierung der Runge-Kutta-Verfahren. Die Idee dabei wäre, ein beliebiges Butcher-Tableau konfigurieren zu können, das dann zur Anwendung kommt. Ausserdem wäre die momentane Beschränkung auf explizite Verfahren aufzuheben – mit der entsprechenden Implementierung eines iterativen Verfahrens zur Lösung von nichtlinearen Gleichungssystemen.

Des weiteren wäre es möglich, den Zellraum weiter zu generaisieren, so dass beispielsweise beliebig viele Dimensionen verwendet werden könnten. Das Problem ist dann natürlich die Darstellung; drei Raumdimensionen sind schon mühsam, vier Raumdimensionen eine echte Herausforderung.

4.2. Caveats

Der primäre Fokus dieses Projekts war nicht, ein perfektes UI zu schreiben – es wird also Konstellationen geben, in denen es sich nicht so verhält wie erwartet. Ausserdem

funktioniert das Pixel-Painting nur auf Little-Endian-Geräten, wie oben besprochen.

4.3. Fazit

Auf der technischen Ebene haben wir gezeigt, dass mit relativ wenig Code eine Struktur erzeugt werden kann, die sich zumindest irgendwie so verhält wie eine Flüssigkeit. Das Problem ist wie immer bei numerischen Verfahren schlussendlich die Genauigkeit – und da sich die Flüssigkeit immer weiterbewegt, führt das zwangsweise zur Akkumulierung der Ungenauigkeit und schlussendlichem Abdriften der Lösung ins Absurde. Als Seiteneffekt dessen wurde wieder einmal gezeigt, dass das explizite Euler-Verfahren in der Praxis nutzlos ist – Runge-Kutta-Verfahren höherer Ordnung aber einigermassen gut funktionieren, so mühsam sie auch zu rechnen sein mögen.

Abgesehen davon war es ein äusserst lehrreiches Projekt. Wir haben gezeigt, dass es grundsätzlich möglich ist, Differentialgleichungen mit einem zellulären Automaten darzustellen – Konrad Zuse hatte in dieser Beziehung also recht. Diese Art von Automaten sind in der Tat mächtig genug, diesen Aspekt der Natur abzubilden. Ob allerdings der Umkehrschluss, dass die Natur selber ein zellulärer Raum sei, zutrifft, ist eine Frage mit weit grösserer philosophischer Tragweite, als dass sie in einem derartigen Projekt auch nur ansatzweise beantwortet werden könnte. Rein intuitiv ist sie allerdings nicht von vornherein zu verneinen.

Literaturverzeichnis

[1] Hans-Joachim Böckenhauer, Dennis Komm, Rastislav Královič, Richard Královič, and Tobias Mömke. On the advice complexity of online problems. In Yingfei Dong, Ding-Zhu Du, and Oscar Ibarra, editors, *Algorithms and Computation*, volume 5878 of *Lecture Notes in Computer Science*, pages 331–340. Springer Berlin Heidelberg, 2009.

[2] Daniel D. Sleator and Robert E. Tarjan. Amortized efficiency of list update and paging rules. *Commun. ACM*, 28(2):202–208, February 1985.

[3] Luca Trevisan. Cs261: Optimization, lecture 17. http://theory.stanford.edu/~trevisan/cs261/lecture17.pdf, 2011.

A. Projektdurchführung

Das Projekt wurde iterativ angesetzt. Es wurden vier Iterationen mit jeweils drei Wochen Laufzeit durchgeführt. Im Folgenden ist die Zuordnung zu den User Stories mit den enstprechenden Story Point Estimates sowie das Burndown-Chart aufgeführt.

A.1. Iteration #1 (20. März–4. April)

- Benutzer möchte die Differentialgleichung der Diffusion in einem zellulären Automaten dargestellt haben. (3p)
- Benutzer möchte interaktiv die Anfangswerte der Zellen verändern können. (1p)

A.2. Iteration #2 (5. April–30. April)

- Benutzer möchte die Differentialgleichung der Welle dargestellt haben. (4p)
- Benutzer möchte alle Ableitungen nach dem Ort dargestellt haben. (2p)
- Benutzer möchte die Schrittweite der Integration einstellen können. (1p)
- Benutzer möchte Objekte mit spezifischer Geometrie ins Medium werfen. (4p)

A.3. Iteration #3 (1. Mai–30. Mai)

- Benutzer möchte Badeenten (ohne Eigengeschwindigkeit, durch 1 Zelle identifiziert) in den Pool setzen können. (3p)
- Benutzer möchte genauere Kontrolle über den Ablauf haben. (1p)
- Schönere Darstellung. (1p)
- Darstellung und Berechnung soll beschleunigt werden. (5p)
- Benutzer möchte verschiedene Parameter des Mediums beeinflussen können. (2p)
- Benutzer möchte zwischen verschiedenen Differentialgleichungen umschalten können. (3p)

A.4. Iteration #4 (1. Juni–15. Juni)

- Benutzer möchte Integrationsalgorithmus einstellen können. (2p)
- Benutzer möchte eine sinnvolle Dokumentation der mathematischen Zusammenhänge. (5p)
- Benutzer möchte eine sinnvolle Dokumentation der Implementation. (2p)