
# Overview of Cellular Automata

May 11th, 2020

Justin Stevens

# Outline

# Definition [1]

A **cellular automata** is defined as a 4-tuple $(\mathbb{Z}^d, Q, N, \delta)$, where:

# Definition [1]

A **cellular automata** is defined as a 4-tuple $(\mathbb{Z}^d, Q, N, \delta)$, where:

- $\mathbb{Z}^d$ represents the Euclidean space we are in. For pathfinding on maps, this is typically $d = 2$.

# Definition [1]

A **cellular automata** is defined as a 4-tuple $(\mathbb{Z}^d, Q, N, \delta)$, where:

- $\mathbb{Z}^d$ represents the Euclidean space we are in. For pathfinding on maps, this is typically $d = 2$.
- $Q$ represents a set of finite discrete integer states, which the values of the cellular automata can take on.

# Definition [1]

A **cellular automata** is defined as a 4-tuple $(\mathbb{Z}^d, Q, N, \delta)$, where:

- $\mathbb{Z}^d$ represents the Euclidean space we are in. For pathfinding on maps, this is typically $d = 2$.
- $Q$ represents a set of finite discrete integer states, which the values of the cellular automata can take on.
- $N = (n_1, n_2, \cdots, n_{|N|})$ represents the neighbourhood function, which defines for a specific cell, which cells around it to consider in its computation, where $n_i \in \mathbb{Z}^d$.

# Definition [1]

A **cellular automata** is defined as a 4-tuple $(\mathbb{Z}^d, Q, N, \delta)$, where:

- $\mathbb{Z}^d$ represents the Euclidean space we are in. For pathfinding on maps, this is typically $d = 2$.
- $Q$ represents a set of finite discrete integer states, which the values of the cellular automata can take on.
- $N = (n_1, n_2, \cdots, n_{|N|})$ represents the neighbourhood function, which defines for a specific cell, which cells around it to consider in its computation, where $n_i \in \mathbb{Z}^d$.
- $\delta : Q^{|N|} \to Q$ is the transition function, which takes in an $N$-tuple and outputs a new state to transition to.

# Definition [1]

A **cellular automata** is defined as a 4-tuple $(\mathbb{Z}^d, Q, N, \delta)$, where:

- $\mathbb{Z}^d$ represents the Euclidean space we are in. For pathfinding on maps, this is typically $d = 2$.
- $Q$ represents a set of finite discrete integer states, which the values of the cellular automata can take on.
- $N = (n_1, n_2, \cdots, n_{|N|})$ represents the neighbourhood function, which defines for a specific cell, which cells around it to consider in its computation, where $n_i \in \mathbb{Z}^d$.
- $\delta : Q^{|N|} \to Q$ is the transition function, which takes in an $N$-tuple and outputs a new state to transition to.

A **configuration** of the cellular automata is a function $c : \mathbb{Z}^d \to Q$, which assigns a specific value to every state.

# Definition [1]

A **cellular automata** is defined as a 4-tuple $(\mathbb{Z}^d, Q, N, \delta)$, where:

- $\mathbb{Z}^d$ represents the Euclidean space we are in. For pathfinding on maps, this is typically $d = 2$.
- $Q$ represents a set of finite discrete integer states, which the values of the cellular automata can take on.
- $N = (n_1, n_2, \cdots, n_{|N|})$ represents the neighbourhood function, which defines for a specific cell, which cells around it to consider in its computation, where $n_i \in \mathbb{Z}^d$.
- $\delta : Q^{|N|} \to Q$ is the transition function, which takes in an $N$-tuple and outputs a new state to transition to.

A **configuration** of the cellular automata is a function $c : \mathbb{Z}^d \to Q$, which assigns a specific value to every state. A **computation** on the cellular automata is given by the global function $G$ specifying

$$\forall c \in Q^{\mathbb{Z}^d} \ \forall i \in \mathbb{Z}^d, G(c)(i) = \delta(c(i + n_1), c(i + n_2), \cdots, c(i + n_{|N|})).$$

# Outline

# Moore Neighbourhood

Known as 8-connected, used in Conway's game of life and several papers.

- [2] uses it in their update rules for growing the obstacles:

$$s_i^{t+1} = \begin{cases} 1 & s_i^t = 0 \wedge \exists x \in \eta_i^t | s_x^t = 1 \\ s_i^t & \text{otherwise} \end{cases}.$$

  Notice here 0 represents a space is free while 1 represents an obstacle, while $\eta$ represents the Moore neighbourhood. This is only applied once.

- [2] also uses it in their update rule for a path to the solution, but without taking into account diagonal cost. [1] does a better job:

$$q_{i,j}^{t+1} = \begin{cases} 1 \quad ; \; q_{i,j}^t = 1 \quad \left(\begin{matrix}\text{1 stands for obstacle} \\ \text{in cellular automaton}\end{matrix}\right) \\ \min \left\{ \begin{matrix} q_{i,j}^t \, , \, q_{i-1,j-1}^t + 14 \, , \, q_{i-1,j}^t + 10, \\ q_{i-1,j+1}^t + 14 \, , \, q_{i,j-1}^t + 10, \\ q_{i,j+1}^t + 10 \, , \, q_{i+1,j-1}^t + 14, \\ q_{i+1,j}^t + 10 \, , \, q_{i+1,j+1}^t + 14 \end{matrix} \right\} & \begin{matrix} \forall \, q'_{i+r,j+s} \neq 1 \; ; OW. \\ r=-1,0,1 \\ s=-1,0,1 \end{matrix} \end{cases}$$

# Extension of Cellular Automata

Instead of just simply classifying the states by discrete integer values, [3] takes this a step further and characterizes them as a 6-tuple

$$(S_{\text{state}}, S_{cf}, S_{pf}, t_{on}, P_{(i,j)}, \phi_s),$$

where $S_{\text{state}}$, $S_{cf}$, $S_{pf}$ are all binary values, which are on or off for if the state is activated, is a child, or is a parent, $t_{on}$ is the first time at which the state is activated (similar to a depth first search), $P_{(i,j)}$ gives the coordinates of the parents of the cell at $(i, j)$, and $\phi_s : \mathbb{Z}^2 \to \mathbb{Z}$ is the cumulative cost from the cell to the goal which is updated by rule 4.

# Extension of Cellular Automata

Instead of just simply classifying the states by discrete integer values, [3] takes this a step further and characterizes them as a 6-tuple

$$(S_{\text{state}}, S_{cf}, S_{pf}, t_{on}, P_{(i,j)}, \phi_s),$$

where $S_{\text{state}}$, $S_{cf}$, $S_{pf}$ are all binary values, which are on or off for if the state is activated, is a child, or is a parent, $t_{on}$ is the first time at which the state is activated (similar to a depth first search), $P_{(i,j)}$ gives the coordinates of the parents of the cell at $(i, j)$, and $\phi_s : \mathbb{Z}^2 \to \mathbb{Z}$ is the cumulative cost from the cell to the goal which is updated by rule 4.

They also use the Moore neighbourhood in their update rules, although its mathematical formulation looks more like they're just considering the diagonals. Three of the four update rules are shown in the next slide.

# Update Rules with Added Parameters [3]

**Rule: 2 -** A cell will only become a child of another cell if the other cell is already in an active state with a $t_{on} < t$.

$$S^t_{cf}(i,j) = \begin{cases} 1 & iff \quad \exists\, S^t_{state}(i+m, j+n) = 1 \\ & \quad \& \; t_{on}\ (i+m, j+n) < t \\ & \quad m, n \in \{1, -1\} \\ 0 & otherwise \end{cases} \tag{4}$$

**Rule: 3 -** A cell will only become a parent cell if it is active, if it is already a child and if its $t_{on} < t$.

$$S^t_{pf}(i,j) = \begin{cases} 1 & iff \quad \exists\, S^t_{state}(i,j) = 1 \\ & \quad \& \; S^t_{cf}(i,j) = 1 \\ & \quad \& \; t_{on}\ (i,j) < t \\ 0 & otherwise \end{cases} \tag{5}$$

**Rule: 4 -** A cell will only become a child of a neighbouring cell with the lowest accumulative cost function.

$$\phi_s(i,j) = \begin{cases} min(\phi_s(i+m, j+n) + \delta) & iff \\ & \quad \exists\, S^t_{state}(i+m, j+n) = 1 \\ & \quad \& \; S^t_{pf}(i+m, j+n) = 0 \\ & \quad m, n \in \{1, -1\} \\ 0 & otherwise \end{cases} \tag{6}$$

where $\delta$ is the cost of connecting the current cell $S(i,j)$ with a neighbouring cell. The costs are:

# Von Neumann neighbourhood

[4] uses a Von Neumann neighbourhood in its update rule by considering the possible locations of pedestrians. While it does not offer pseudocode for the update rules, it instead offers two model approaches: a “stop and go" approach, which stops when a transient obstacle is in its way, and go when the obstacle is cleared. The velocity of the robot is then a value between 0 and $v_{\max}$. This parameter could be an interesting way to balance exploration in a static environment. In the second approach, they use a “detour" method, where they have a precomputed path to the goal, but when approached by an obstacle, they take an alternate path. Some of the possible scenarios for the cellular automata are:

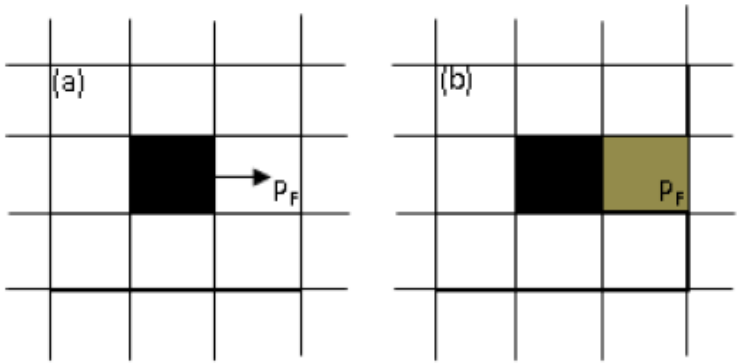



Figure 4. The possible configurations of "stop and go" method.

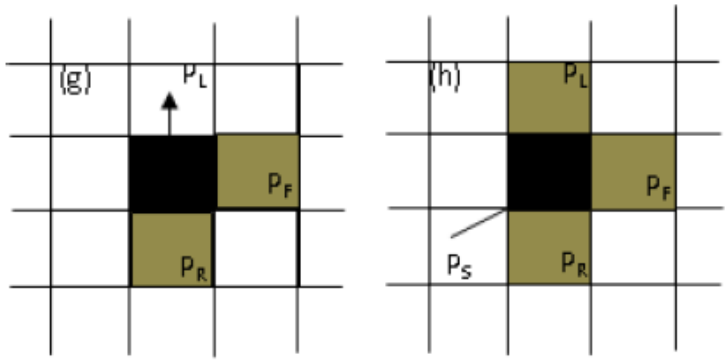



Figure 5. The possible configurations of detour method.

# Von Neumann neighbourhood

[5] also uses a Von Neumann neighbourhood. For each spot in the grid, they keep track of the “wave of excitation", which is simply just a $+$ if a spot has been visited. They also keep track of a parent vertex for how the wave reached a vertex. They also adopt the symbols $\#$ and $\cdot$, whose meaning are unclear. The transition rules are:

- $\#$, when $x^t$ is in states $\#$ or $+$.
- $+$, when $x^t = 0$.
- $\bullet$, when $x^t = \bullet$ and there in no neighbor $y^t = +$.
- $w_{xy}$, when $x^t = \bullet$ or $x^t > 0$ and there is at least one neighbor with $y^t = +$ and the weight of the edge between $x^t$ and $y^t$ is the minimum of other edges in the neighborhood.
- $x^t - 1$, when $x^t > 0$ and there is no other neighbor $y^t = +$ that has a weight in the edge between $x^t$ and $y^t$ in order that $w_{xy} < x^t$.

# Lee's Algorithm

[5] also discusses Lee's algorithm, which essentially fills a wave until the goal node is reached. They then converted this to a cellular automata with only 14 states which can be used in our project. Unfortunately, this can only handle the case of when the weights are uniform:

# Pseudocode in [5]

```
(01) cellular automaton Lee_routing;
(02)
(03) const dimension = 2;    --2-dimensional CA
(04)       distance = 1;     --neighbourhood distance
(05)
(06)       --relative coordinates of cells: center, north, east, south, west
(07)       C=[0,0];  N=[0,1]; E=[1,0]; S=[0,-1]; W=[-1,0];
(08)
(09) type  celltype = (free, obstacle, Spoint, Epoint,
(10)                   --phase 1: wave marks
(11)                   wave_up, wave_right, wave_down, wave_left,
(12)                   --phase 2: path directions
(13)                   path_up, path_right, path_down, path_left,
(14)                   clear, Ready);
(15)
(16) group wave = {wave_up, wave_right, wave_down, wave_left};
(17)       path = {path_up, path_right, path_down, path_left};
(18)       neighbours = {N, E, S, W};
(19)
(20) var   h : celladdress;   --temporary, address of a neighbor
(21)       z : celltype;      --temporary, tested state (wave/path mark)
(22)
(23) rule
(24)   case ^C of    --depending on contents * of cell with address C
(25)   free:   if one(h in neighbours & z in wave: ^h in {start, wave}) then
(26)           ^C := z; --set wave mark pointing to a neighbour in state start or wave
(27)   Epoint: if one(h in neighbours & z in path: ^h in {start, wave}) then
(28)           ^C := z; --set path mark pointing to a neighbour in state start or wave
(29)   wave:   if (^N = path_down) or (^E = path_left) or
(30)              (^S = path_up) or (^W = path_right) then
(31)           ^C := ^C + 4 --change wave mark into path mark
(32)           else
(33)           if one(h in neighbours: ^h in {path, clear}) then
(34)           ^C := clear; --clear unused path marks
(35)   clear:  ^C := free;
(36)   Spoint: if one(h in neighbours: ^h in path) then
(37)           ^C := ready; --termination: start->ready
(38)   end;  --of case
```

**Fig. 9** The shortest path algorithm described in the language CDL.

# Outline

# Robotics

The objectives of pathfinding in robotics is slightly different, since the robot is navigating in a Euclidean space without any discrete locations. In order to model this as a pathfinding problem in RTHS, the sequence of configurations of the robot is considered to be given by the set $Q = \{q_1, \cdots, q_n\}$ and the obstacles by $O = \{O_1, \cdots, O_n\}$. The goal is then to find a path $\tau : [0, 1] \to Q$, such that $\tau(0) = q_{\text{init}}$ and $\tau(1) = q_{\text{goal}}$ in the obstacle-free configuration space [3].

In [2] they use an extension of the obstacles given on a previous slide to ensure no collisions occur. In [3] they use a Minkowski sum to remodel the obstacles:

$$R \oplus O_i = \{x + y | x \in R, y \in O_i\}.$$

This then slightly enlarges the obstacle space. The image, which is typically large and taken by a camera, is then reduced to a grid:

# Transformation in [2]

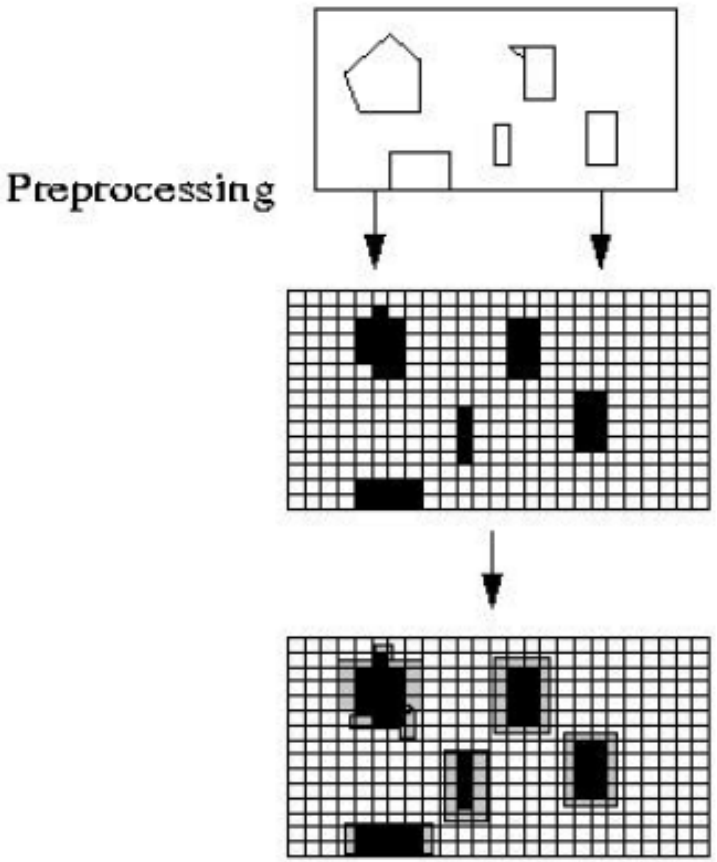



Image of the camera in resolution 640x480

Phase 1:
Transforming in cellular space
Resolution 76*60

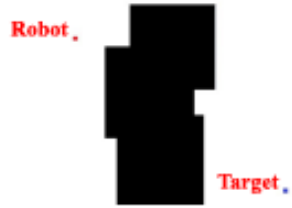
Phase 2:
Borders

# Transformation in [3]

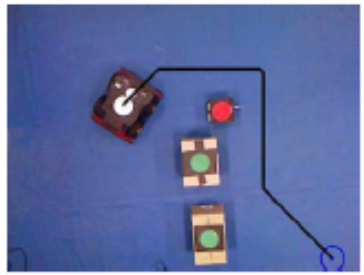
(a) Image acquisition

(b) Obstacle identification

(c) Configuration space

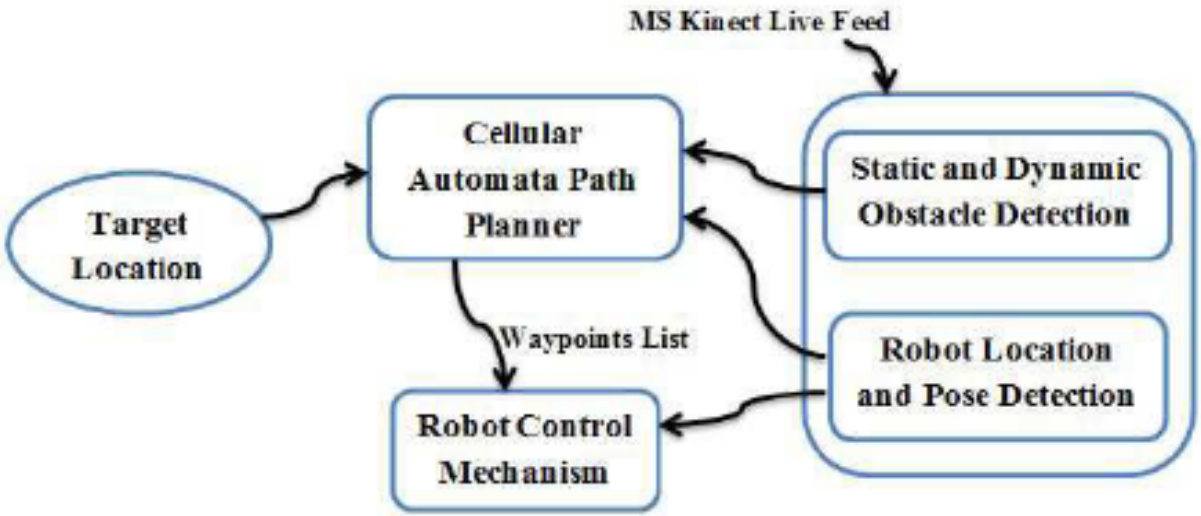



(d) Robot and target identified



(e) Planned path

They then used this to provide commands to the robot to move:



# Moving Obstacles

In [4] they take this analysis a step further as they allow for moving pedestrians, instead of stationary obstacles. In this model, they have pedestrians moving at certain velocities that the robot has to avoid. They simplify the situation to not consider the dynamics of the robot, and make it a purely geometric problem:

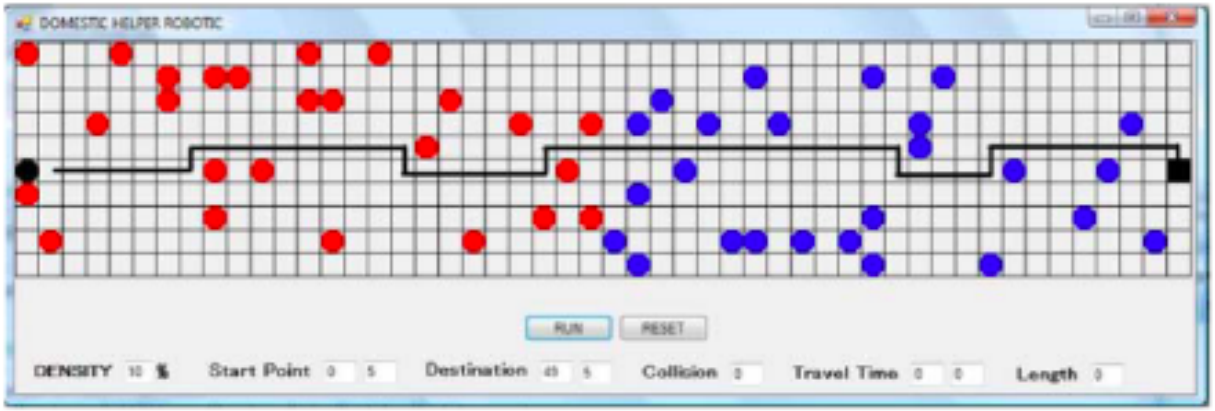



Figure 1: The path is precomputed using Dijkstra's algorithm

# Outline

# Big $O$ notation

The algorithm for [2] has runtime $O(x_{\max}^2 \cdot y_{\max}^2)$, when the transformed grid is $x_{\max} \times y_{\max}$. The algorithm for [1] has runtime $O(n^3 m^4)$, where there are $n$ agents on an $m \times m$ grid. Neither of these offer comparisons to how the proposed cellular automata approach does to the traditional approaches.

# Big $O$ notation

The algorithm for [2] has runtime $O(x_{\max}^2 \cdot y_{\max}^2)$, when the transformed grid is $x_{\max} \times y_{\max}$. The algorithm for [1] has runtime $O(n^3 m^4)$, where there are $n$ agents on an $m \times m$ grid. Neither of these offer comparisons to how the proposed cellular automata approach does to the traditional approaches.

When the proposed method in [3] was compared with other techniques such as Dijkstra's algorithm, D*, and Modified Pulse Coupled Neural Network (MPCNN), it consistently outperformed them giving more reliable results on several scenarios, with both static and dynamic environments.

# Big $O$ notation

The algorithm for [2] has runtime $O(x_{\max}^2 \cdot y_{\max}^2)$, when the transformed grid is $x_{\max} \times y_{\max}$. The algorithm for [1] has runtime $O(n^3 m^4)$, where there are $n$ agents on an $m \times m$ grid. Neither of these offer comparisons to how the proposed cellular automata approach does to the traditional approaches.

When the proposed method in [3] was compared with other techniques such as Dijkstra's algorithm, D*, and Modified Pulse Coupled Neural Network (MPCNN), it consistently outperformed them giving more reliable results on several scenarios, with both static and dynamic environments.

In [4], while the "stop and go" method always produced the shortest path, since it never deviated from the pre-computed route, the detour method was more time efficient, since it didn't stop and wait until a cell was available.

# Big $O$ notation

The algorithm for [2] has runtime $O(x_{\max}^2 \cdot y_{\max}^2)$, when the transformed grid is $x_{\max} \times y_{\max}$. The algorithm for [1] has runtime $O(n^3 m^4)$, where there are $n$ agents on an $m \times m$ grid. Neither of these offer comparisons to how the proposed cellular automata approach does to the traditional approaches.

When the proposed method in [3] was compared with other techniques such as Dijkstra's algorithm, D*, and Modified Pulse Coupled Neural Network (MPCNN), it consistently outperformed them giving more reliable results on several scenarios, with both static and dynamic environments.

In [4], while the "stop and go" method always produced the shortest path, since it never deviated from the pre-computed route, the detour method was more time efficient, since it didn't stop and wait until a cell was available.

In [5], for a 2D CA with $n$ cells with four neighbours, where the edges can be of weight $\{0, \cdots, v, \infty\}$, this can be solved in $O(vn)$ time, which is relatively fast since the computation is done in parallel.

# Comparison to A*

[3] also compares CA with A* to show it is more efficient. For their comparison, they use Euclidean distance as the heuristic. They noted both that the search time is independent of path length, and is increasingly more efficient with added obstacles.

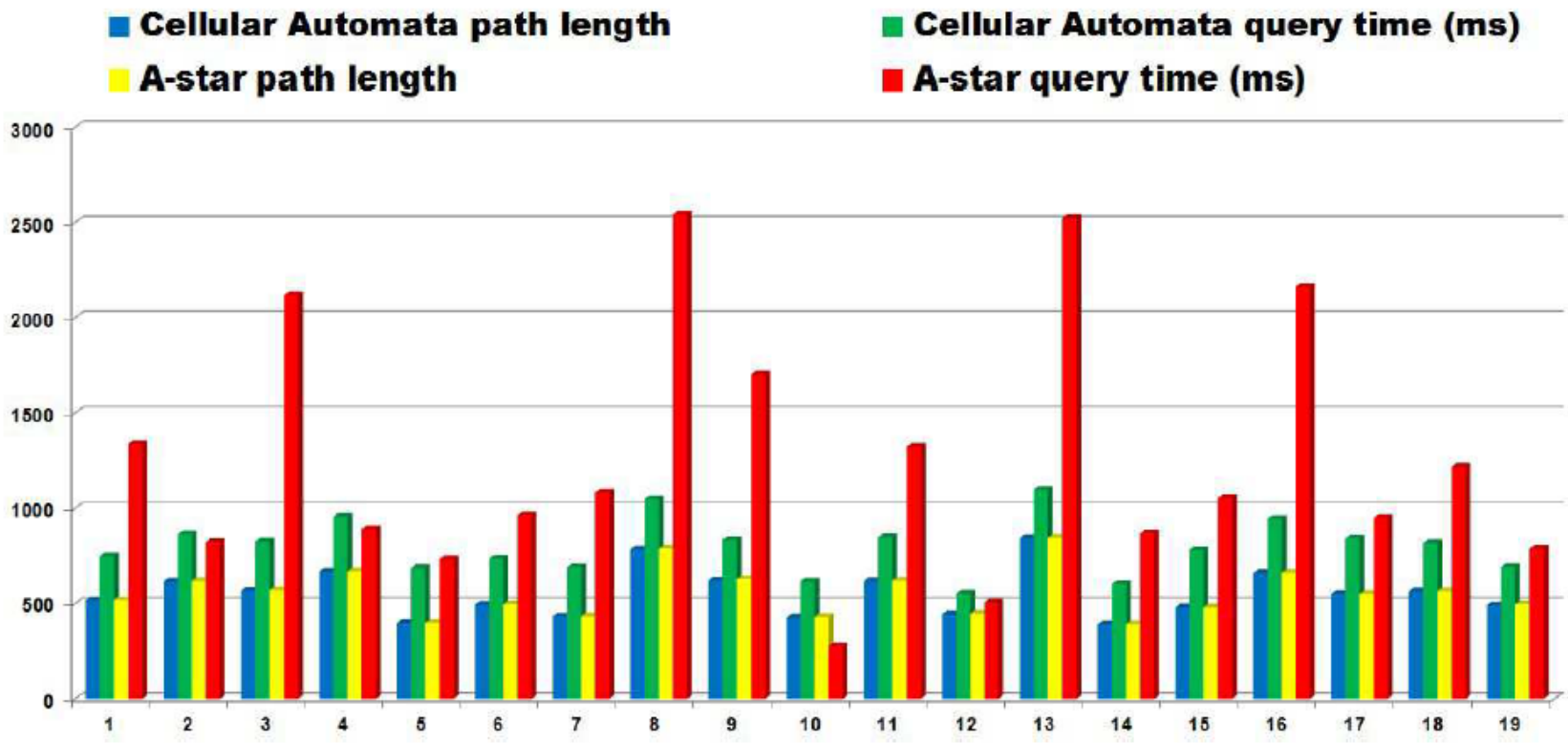



**Figure 5.** Path length and planning time-based comparison of CA with A*

# Outline

# Density Problem

In [6], they attempted to evolve cellular automata to solve the task of determining whether an initial configuration continued majority 1's or 0's. This is a difficult task, since while a normal program could use a counter variable to determine this, cellular automata are based only on local interactions. They used a larger radius ($r = 3$) than most of the previous programs we saw ($r = 1$), which is another parameter we can use in our work. With this radius, the size of all possible neighbourhoods is $2^{2r+1} = 128$, meaning there are $2^{128}$ possible rules with binary states.

# Density Problem

In [6], they attempted to evolve cellular automata to solve the task of determining whether an initial configuration continued majority 1's or 0's. This is a difficult task, since while a normal program could use a counter variable to determine this, cellular automata are based only on local interactions. They used a larger radius ($r = 3$) than most of the previous programs we saw ($r = 1$), which is another parameter we can use in our work. With this radius, the size of all possible neighbourhoods is $2^{2r+1} = 128$, meaning there are $2^{128}$ possible rules with binary states.

For their experiments, they considered a task with size $N = 149$, created 100 chromosomes at random with uniform density over 1's, tested them on 100 new IC's per generation with uniform density over $\rho \in [0, 1]$. The fitness was simply the fraction of IC's which output was exactly correct (i.e. all 0's or 1's at the end for $\rho_C < \frac{1}{2}$ or $\rho_C > \frac{1}{2}$, respectively). They then kept the top 20%, and used crossover with mutations to create 80 more chromosomes, with preference to those higher ranking "elite" chromosomes.

# Block Expanding Rule [6]

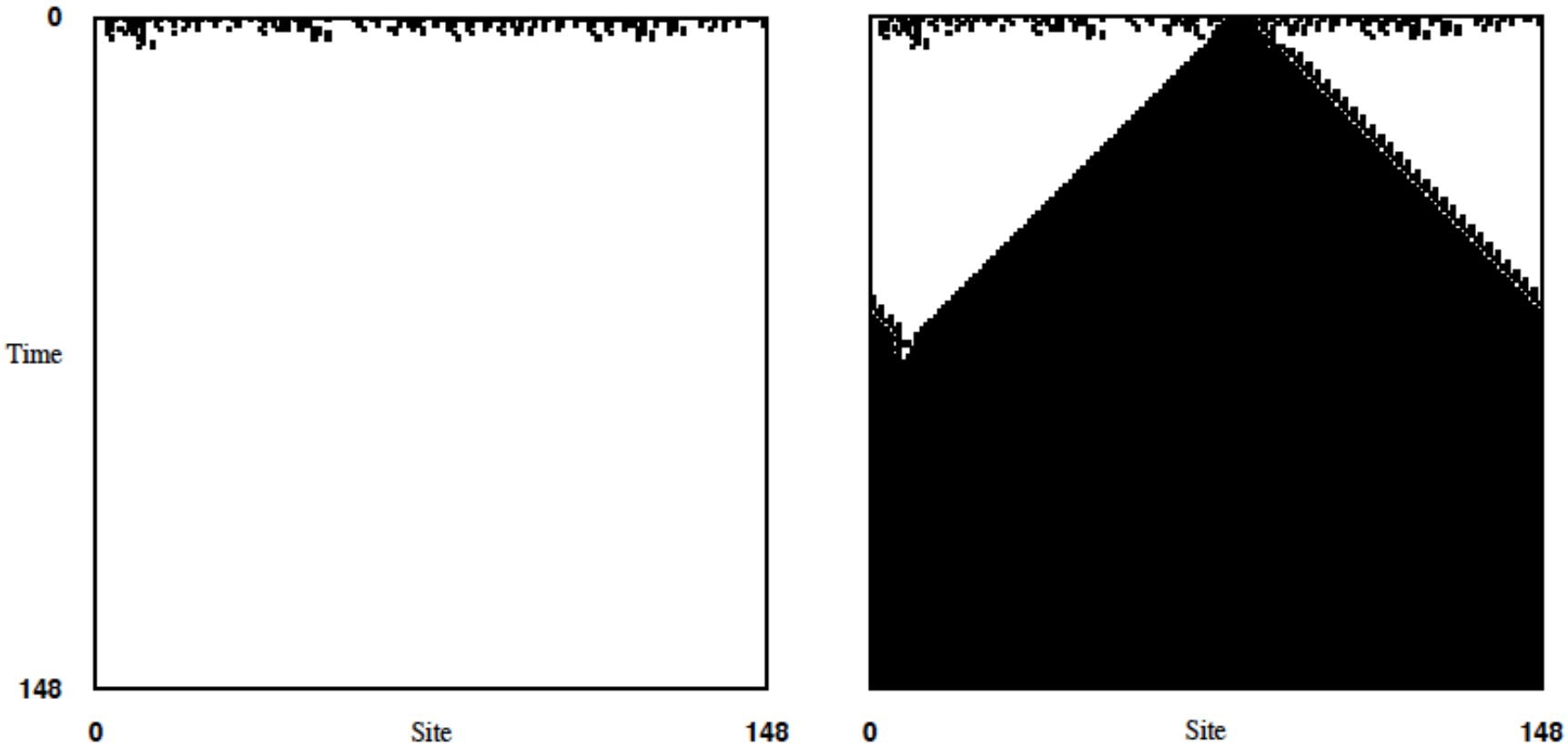



Figure 3: Space-time diagrams for a "block-expanding" rule, $\phi_{\text{exp}}$. In the left diagram, $\rho_0 < 1/2$; in the right diagram, $\rho_0 > 1/2$. Both ICs are correctly classified.

# Particle Based Rule [6]

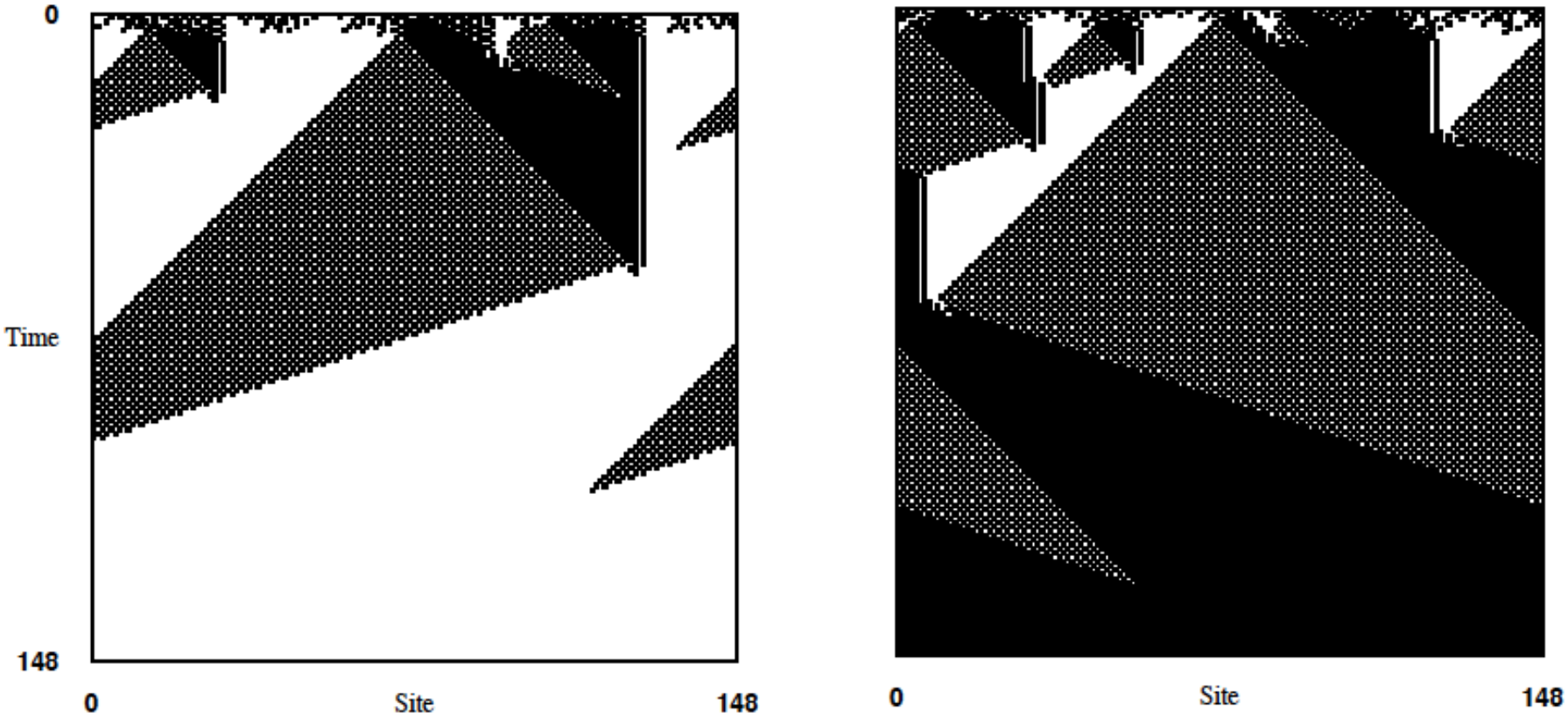



Figure 4: Space-time diagrams for $\phi_{\text{par}}$, a "particle-based" rule. In the left diagram, $\rho_0 < 1/2$; in the right diagram, $\rho_0 > 1/2$. Both ICs are correctly classified.

# Explainable Problem [6]

It can be determined that $\phi_{\text{exp}}$ used a rule that expanded blocks which looked promising, while $\phi_{\text{par}}$ propagated information about blocks using a checker board pattern to determine whether there were majority 1's or 0's, these are informal explanations. In general, it's difficult to explain the rule table produced by the CA, since the emergent behaviour is determined by bits, which are difficult to read. Instead, they can be analyzed in terms of their behaviour. For $\phi_{\text{par}}$ this can be characterized in the table below:

| Regular Domains | | |
|---|---|---|
| $\Lambda^0 = 0^*$ | $\Lambda^1 = 1^*$ | $\Lambda^2 = (01)^*$ |

| Particles (Velocities) | |
|---|---|
| $\alpha \sim \Lambda^0\Lambda^1$ (0) | $\beta \sim \Lambda^1 01 \Lambda^0$ (0) |
| $\gamma \sim \Lambda^0\Lambda^2$ (-1) | $\delta \sim \Lambda^2\Lambda^0$ (-3) |
| $\eta \sim \Lambda^1\Lambda^2$ (3) | $\mu \sim \Lambda^2\Lambda^1$ (1) |

| Interactions | |
|---|---|
| decay | $\alpha \to \gamma + \mu$ |
| react | $\beta + \gamma \to \eta,\ \mu + \beta \to \delta,\ \eta + \delta \to \beta$ |
| annihilate | $\eta + \mu \to \emptyset_1,\ \gamma + \delta \to \emptyset_0$ |

# Sources I


Y. Tavakoli, H. Seyyed Javadi, S. Adabi.
*A Cellular Automata Based Algorithm for Path Planning in Multi-Agent Systems with A Common Goal*.
IJCSNS, 2008.

C. Behring, M. Bracho, M. Castro and J. A. Moreno.
*An Algorithm for Robot Path Planning with Cellular Automata*.
Proceedings of the Fourth International Conference on Cellular Automata for Research and Industry, 2000.

U. Ahmed Syed and F. Kunwar.
*Cellular Automata Based Real-Time Path-Planning for Mobile Robotics*
Int J Adv Robot System, 2014.

K. Arai and S. Ray Sentinuwo

*Mobile Robot Motion Using Cellular Automaton Model to Avoid Transient Obstacles*

I.J.Modern Education and Computer Science, 2013.

M. I. Tsompanas, N. I. Dourvas, K. Ioannidis, G. Ch. Sirakoulis, R. Hoffmann, and A. Adamatzky

*Cellular Automata Applications in Shortest Path Problem*

Shortest path solvers. Springer, 2018

M. Mitchell, J P. Crutchfield, and R. Das

*Evolving Cellular Automata with Genetic Algorithms: A Review of Recent Work*

Proceedings of the First International Conference on Evolutionary Computation and Its Applications, 1996.

# Future Work


M. Mitchell, P T. Hraber, and J. P. Crutchfield

*Revisiting the Edge of Chaos: Evolving Cellular Automata to Perform Computation*

Santa Fe Institute

W. Gilpin

*Cellular automata as convolutional neural networks*

Stanford