Modelling and Simulation Project 2
Final Report

Parallel Discrete Time Simulation of a Field Battle Using Cellular Automata

Ankit Srivastava, Patrick Flick, Rahul Nihalani

Background

Creating real war like situations for developing military strategies can be expensive and exhausting in the real world. Manual simulations were used for developing military strategies before computers were around. It is theorized that chess was developed for modeling war and developing strategies for the same [1].

More recently, a war game, *kriegsspiel*, was developed for the purpose of training officers in the nineteenth century German and Prussian armies [2]. The advent of computers saw a rise in the development of computer based military simulations. SIMNET was one of the earliest of these computer simulations, developed by the United States military for training purposes [3].

Introduction

In this project, we simulate a battle between two armies. Each army consists of a number of soldiers of various kinds. Different soldiers will differ from each other in parameters like skills, aggressiveness and the distance at which they can kill another soldier.

We simulate the dynamics of the battlefield using cellular automata. The two armies start at some point in the battle field. In each time step, both the armies move towards each other and try to kill the enemy soldiers. Each army has a predefined strategy as to how to move towards the enemy and kill their soldiers. We can have various criteria according to which we declare the winner of the battle. For example: reaching a particular point in the opponent's area, killing a minimum number of opponent's soldiers, killing a particular enemy soldier, just to name a few.

Conceptual Model

Objectives

We simulate a battle between two armies using a cellular automata grid where armies move and try to kill the soldiers of the enemy army according to a predefined strategy. We can use this simulation to compare various strategies and determine conditions under which different strategies work well.

Content

Each army consists of a number of soldiers of various kinds. In this project, we simulate three kinds of soldiers, described below:

- **Swordsmen**: These are fighters with sword as their fighting weapon. In our model, this translates to the fact that they have the ability to kill an enemy soldier that is in one of the neighboring cells.
- **Leaders**: Just like swordsmen, their fighting weapon is a sword as well. However, their skill and aggressiveness (described later) is generally higher than the swordsmen (and archers). Moreover, they are responsible for leading their army towards the enemy.
- **Archers**: Archers have bows and arrow as their weapon, so they have a larger kill radius compared to swordsmen. In other words, they can kill an enemy from a larger distance compared to swordsmen. To compensate for archers having a larger kill radius than swordsmen, we set their average skill level lower than that of the swordsmen.

In addition, each soldier has a few characteristics attached to it (each parameter has a value in the range [0,255]):

- **Skill**: The skill level in killing the opponent's soldier and defending against its attack.
- **Aggressiveness**: Aggressiveness represents to what extent does a soldier seek the company of it's own soldiers. In other words, how much a soldier cares about its own happiness. A more aggressive soldier would give a higher priority to accomplishing the army's goal than caring about it's own happiness. A less aggressive soldier would first try to become happy by seeking company of its own army's soldiers and then try to focus on the goals of its army.

We model the two dimensional battlefield as an array of cells. We use the floor field model [4][5] to simulate the movement of soldiers in the battle field. Each time step in our simulation is divided into two phases: The **movement phase** and the **killing phase**. We first describe the movement phase, after which we will describe the killing phase.

Each cell (x, y) can contain at most one soldier at any given time. Every soldier has following attributes associated to it:

- Army
- Type: Leader, archer or a swordsman
- Skill
- Happiness (described later)
- Aggressiveness

In addition to the soldiers, each cell has fields associated with it. A field represents how attractive that cell is for soldiers to visit. The higher the field of a cell, more attractive it is for soldiers. We have two kinds of fields for each cell (value of each kind of field for a cell lies in the range [0,1]):

- **Static field** $\tau_s(x, y)$: Static field is used to represent the battlefield. This can be used to model battlefields of different shapes and compositions. For example, an irregular battlefield can be modelled by circumscribing the smallest possible rectangle around it and setting the static fields for cells outside the battlefield to be zero. Rivers and marshy areas within the battlefield can be modelled by setting their static field to be lower compared to the static field of normal surfaces.
- **Dynamic field** $\tau_d(x, y)$: Dynamic field at any cell can be thought of as footprints of soldiers that have recently visited the cell. Each soldier, on leaving a cell, leaves a dynamic field in that cell. This field then decays as the time passes. The leaders in the army will leave a higher field than the other soldiers, and this helps them lead their army towards enemy. Moreover, we maintain separate dynamic fields for each army, which helps guide soldiers better.

We define **Matrix of Preference (MoP)** for each soldier which represents its preference to move to a neighboring cell (or stay at the current position) at any given time. Specifically, MoP is a 3x3 matrix, with the center element representing the current position and all other elements representing the neighboring positions. The value of each element represents the weight with which the current soldier wants to move to that position in the next time step.

Each soldier has three MoPs associated with it at each time step:

- **Global MoP:** This matrix is built based on the goal of the army. A higher weight is given to the direction which tries to accomplish army's goals to maximum extent.
- **Local MoP**: This matrix is built based on the goal of increasing soldier's **happiness**. A soldier's happiness is a function of how many soldier's of its own army are present in its neighborhood. The more own soldiers are present, the more happy a soldier is. We scan an extended neighborhood of the soldier (a $k \times k$ matrix around the soldier). A higher weight is given to the direction in which more soldiers of the same army lie.
- **Last move MoP:** To prevent soldiers from changing their movement direction very frequently, we give some weight to the direction which the soldier took in the last time step. Specifically, the last move MoP is 1 for the direction in which the soldier last moved, and 0 for all other cells.

The net MoP is determined based on how aggressive and how happy a soldier is. Specifically, if a soldier is very aggressive, then it will give a higher weight to the global MoP. If on the other hand it is less aggressive, then there are two cases. If the soldier is happy, then again it would give a higher weight to the global MoP. Otherwise it would give a higher weight to the local MoP. Therefore:

$$M = p.\, M_m \; + \; (1-p)(\, a \cdot M_g + (1-a) \cdot [h \cdot M_g + (1-h) \cdot M_l])$$

Where:

- M is the net MoP
- M_g is the global MoP
- a is the aggressiveness of the soldier.
- h is the happiness of the soldier.
- p is the weight given to the last movement MoP.
- M_l is the local MoP.
- M_m is the last movement MoP

Using the calculated MoP, the static floor field, and the dynamic floor field for calculating a 3×3 ***transitional probability matrix*** (p) for the soldier. The transitional probability matrix assigns to each vacant neighboring and the current cell, the probability with which the current soldier wants to move to that cell in the next time step. In every time step, the soldier chooses a target cell based on transitional probabilities. Note that the soldier can choose to stay in the same cell with a non zero probability. It is calculated as follows:

$$p_{ij} = NM_{ij} exp(\beta \Delta_s(i,j)) exp(\beta \Delta_d(i,j))(1 - n_{ij}) d_{ij} \;\; ,$$

where

- (i,j) is the index in the 3×3 matrix.
- $\Delta_s(i,j) = \tau_s(i,j) - \tau_s(0,0)$ and $\Delta_d(i,j) = \tau_d(i,j) - \tau_d(0,0)$.
- N is a normalization factor, such that $\Sigma p_{ij} = 1$,
- n_{ij} is occupation number of the cell represented by (i,j),
- *Need to determine what values to assign to these. They may not be relevant for our purposes.*
- β is inverse temperature
- d_{ij} is a correction factor used to prevent soldiers from following their own field.

Since multiple soldiers may want to move to the same cell, we need to do conflict resolution for each cell. This is done as follows:

- Let a target cell have *n* candidate soldiers who want to acquire it in the next time step. Further, let $p^{(1)}$, $p^{(2)}$ … $p^{(n)}$ be their transitional probabilities for the current cell. Then the probability with which candidate *i* acquires the current cell is given by:

$$p_r^{(i)} = \frac{p^{(i)}}{p^{(1)}+p^{(2)}+ \ldots + p^{(n)}}$$

- Based on the calculated relative probabilities for different soldiers, a soldier is picked to move to this cell. All the other soldiers, who claimed this cell, will retain their position.

This completes the movement phase. We now describe the killing phase.

In every time step, each soldier picks one enemy soldier that is present in its killing range (if at all). The target soldier is selected with a uniform probability. As discussed earlier, each soldier has a skill level of killing an opponent and defending against it. Killing is performed according to the following rule:

- Let there be *n* enemy soldiers that wish to kill a particular soldier (we will refer to this soldier as home soldier). Let s_{self} be the skill of the home soldier and let S be the sum of the skills of all the enemy soldiers that attacked this soldier in this time step. Then the probability with which the home soldier survives in the next time step is given by:

$$p_{survival} = \frac{s_{self}}{s_{self}+S}$$

The movement and killing phases are repeated till one of the armies is successful in winning the battle based on set criteria.

We now describe the **winning criteria** for a battle. We model two different winning criteria in our battle simulation:

- Capturing a flag: Each army has a target point in the battle field where it's target flag is hoisted. Usually the target for one army is near the starting point of the enemy army. The army that captures its target flag first wins the battle.
- Destroying the enemy army: The army that is able to kill all the soldiers in the enemy army wins the battle.

The winning criteria is a parameter in our

Input and Output

Following entities would be the input to the simulation program:

- The battlefield terrain: For standard shapes (eg. rectangle, circle), the field can just be described as a simple equation. For arbitrary shapes battle field, as discussed, the shape can be circumscribed in smallest possible rectangle which can then be passed as input with the static field for each cell marked appropriately.
- Initial configurations of the two armies in the field. This will include the positioning of the soldiers along with characteristics like type, skill and aggressiveness.
- Criteria for winning the war. We model the following criteria for winning the battle:
    - Capturing the enemy flag or raising own flag in an enemy territory.
    - Killing all the soldiers of the enemy army.

The output of the simulation is two fold. We capture the snapshot of the battlefield in each time step (or every few time steps). These snapshots can be concatenated to create a video of the battle simulation. In addition, we output some statistics about the battle. For example, time taken for the battle to end, number of soldiers killed, just to name a few.

Assumptions

We make the following assumptions in our simulation:

- The strategy of an army does not change during the battle.
- New soldiers are not added during the battle, unless explicitly done so by the user of the simulation.

Simplifications

We have the following simplifications in our model:

- The moving speed of all soldiers is identical: one cell per time step.
- If a soldier is not able to move to its first choice of the neighborhood cells, he stays in the current cell.
- Archers have infinite number of arrows.
- Archers don't miss their target.
- Attacks on soldiers don't accumulate across iterations.
- The aggressiveness of a soldier does not change during the battle.
- Both the armies have same quality of weapons.

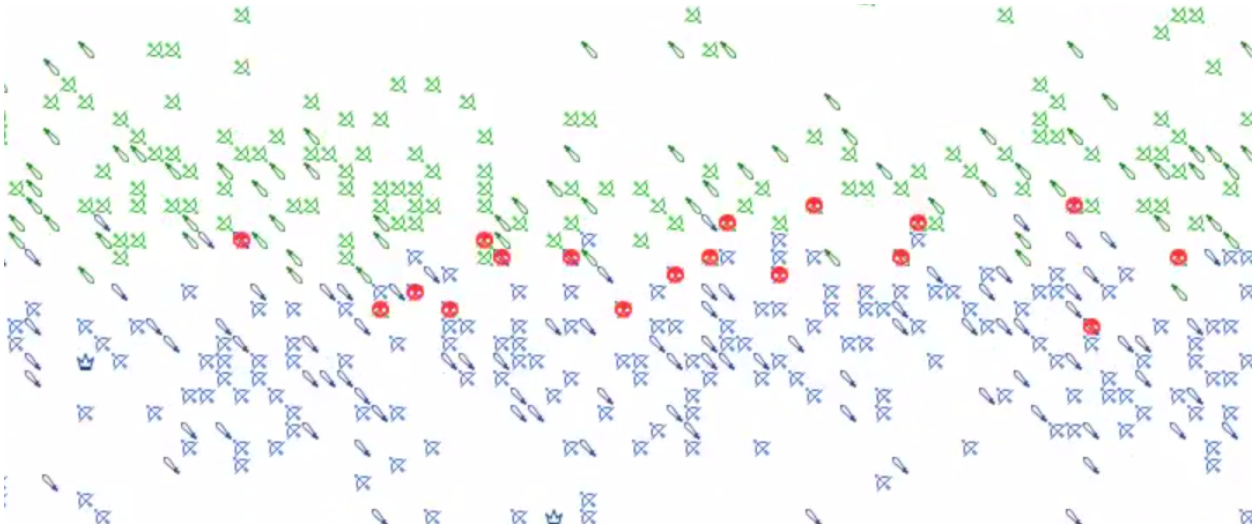
Methodology

We used C++03 for the development of core part of simulation model for the above described conceptual model. We modeled the battlefield as a cellular automata that is distributed on a logical mesh topology. We have a class for soldiers, which stores all the corresponding attributes. A class for the floor field model encapsulates the actual grid on which soldiers move as well as the static and dynamic fields. There are two public functions, one for movement and another for kill, which would be called in each time step. At the end of each time step, we would check if the target has been achieved by one of the armies and terminate if that is the case.

We used Python (version 2.7) for the development of the front-end, for setting parameters for the battle, configuring the battlefield, etc. We used SWIG [6] for generating an interface to the C++ back-end code, which was used by the Python front-end for passing the parameters and calling appropriate functions on the C++ side.

Visualization

We used two Python libraries for rendering videos of the battle: Gizeh [7] and MoviePy [8]. Using these libraries we plotted a pixel (faster) or an icon (slower) for every soldier in both the armies in every time step, after each move and kill step described above. For example, we simulated a battle with the same set of soldiers but different win criteria described above and recorded the resulting battle. The icon version of resulting battle is shown in the videos which we have uploaded at the following links: capture the flag & kill the enemy. A screenshot from the second link is shown in the Figure below.

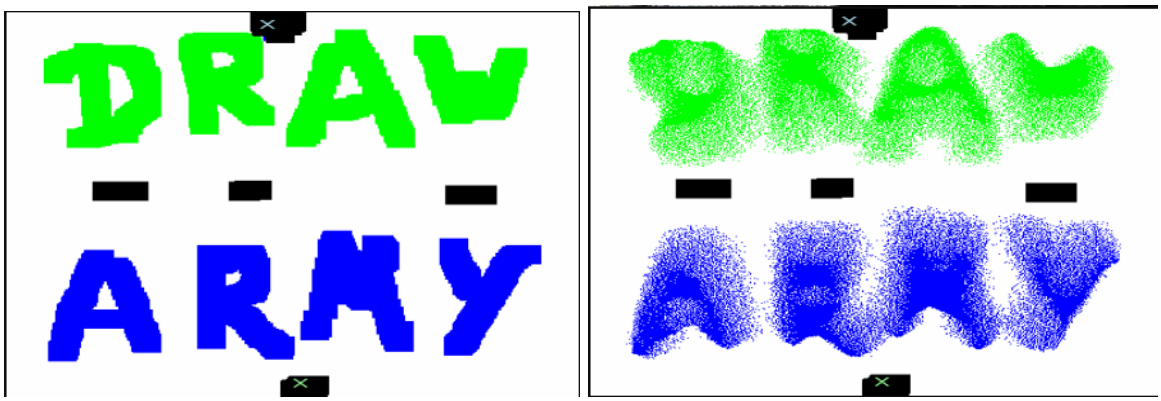


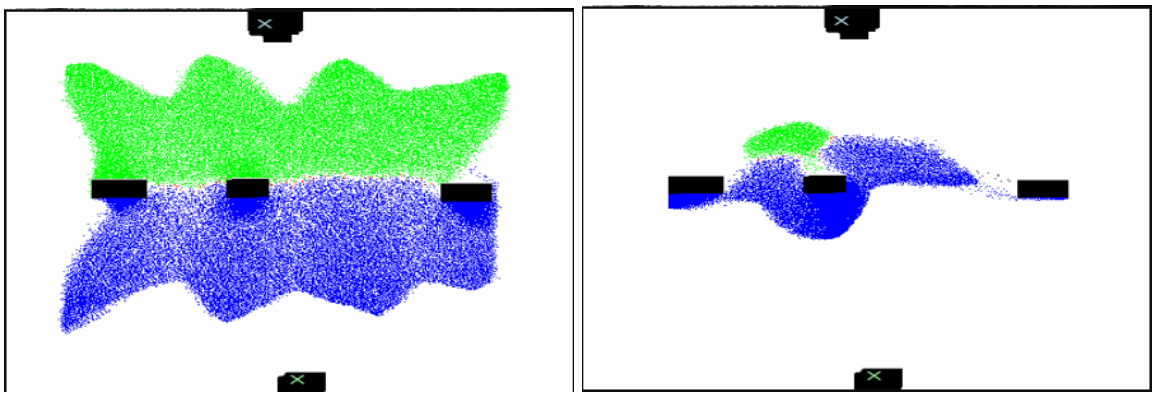
As can be seen, we represent the two armies with different colors: blue and green. We also added different icons for different types of soldiers: bow and arrow for Archers, swords for Swordsmen, and crown for Leaders. We mark death of a soldier using a Red Skull. We also check if one of the army has won the battle and display a message to that effect as shown in the Figure below.

FIN!

The objective of both the armies was to kill all the soldiers and the blue army won in this case. Therefore, a message indicating the end of the battle is displayed as soon as the last soldier of the green army dies.

Using Pygame [9], we created an interactive version of our simulation which allowed us to interactively draw the two armies (green and blue), along with bounded areas (black), and then simulate the battle on the go. Here, each soldier/each cell is represented by a single pixel. This allows interactive visualizations of large battles. It is also possible to pause the battle, modify the armies or the battlefield (e.g., provide reinforcements) and then continue the battle. Additionally, target zones for both armies can be set separately. Below are some screenshots from this interactive visualization:





An example video can be viewed here:
https://www.youtube.com/watch?v=QzDJUp8iV0Q

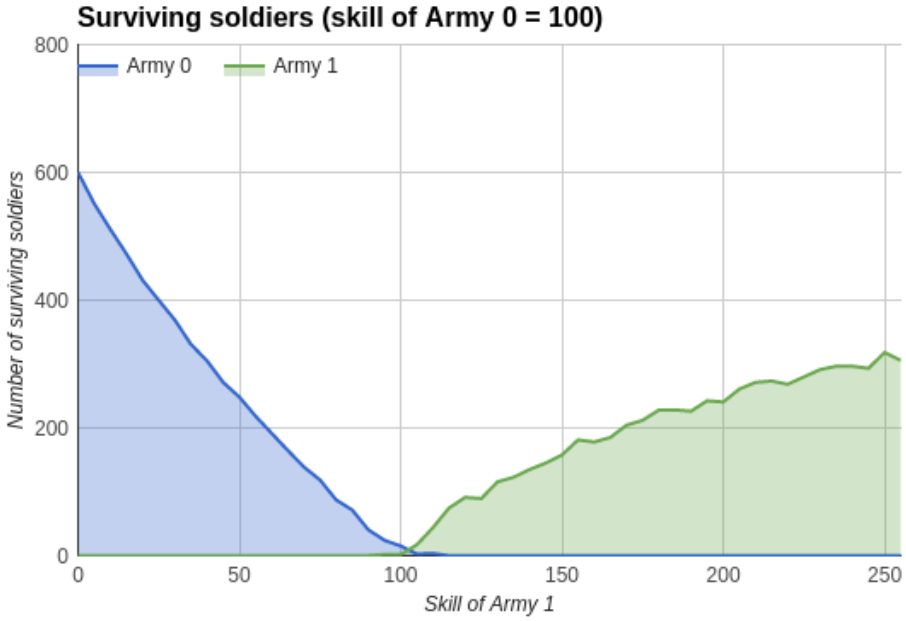
Parallelization

In order to be able to simulate large battles quickly, we parallelize our implementation using MPI. This allows large scale, distributed memory parallel simulation of large battlefields. We equally distribute the 2-dimensional battlefield onto a 2-dimensional cartesian processor mesh. Each processor holds only its local part of the battlefield in local memory. For an efficient cellular automata simulation, each processor also contains an additional boundary around its local share of elements. These boundaries contain the current state of the neighboring processors. The boundaries are updated after every iteration by communicating with the four neighboring processors.
Different field require different sized boundaries. The static field requires only a boundary of size 1, which never has to be updated after initial synchronization. The dynamic field also requires a boundary size of 1, and needs updating in every iteration. The field of soldiers needs a larger boundary, due to the soldiers field of view (extended neighborhood size) and the kill radius of archers. We use the maximum of these two extends as the boundary size of the soldier field. Other temporary fields (for claiming movements and accumulating kill probabilities) may also require larger boundaries.
Using this parallelization, we achieved speedups of > 3.1 times for running on 4 processors. Due to unmet build-dependencies on Jinx, we did not run our code on larger distributed memory machines.

Results

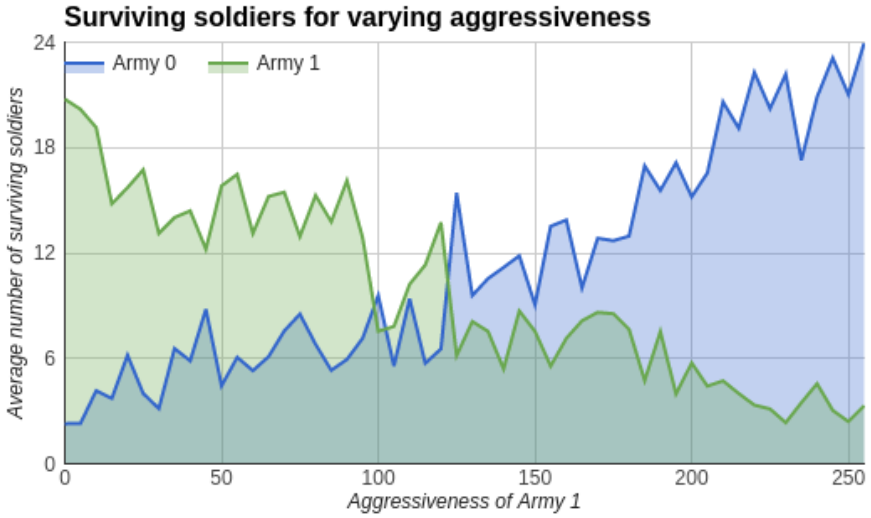
We ran our simulation program for various settings of different parameters. Specifically, we tried to vary individual parameters while keeping others fixed, and see the effect it has on the run. We observed the following effects:

- **Last move weight**: We varied the weight given to the last move direction for calculating the current move direction. We observed that as we increased the weight, the movement of soldiers became more streamlined and less random. The frequency of direction change also reduced, which is the expected behavior.
- **Skill level**: We ran the simulation with different average skill levels for the two armies. We observed that even a minute increase of skill level of one army over the other almost always results in that army winning the war. The figure below shows the average number of surviving soldiers for varying skill levels of the army. For this plot, we fix the skill of Army 0 and vary the skill of Army 1 from 0 to the maximum of 255. The battle is fought on a symmetric field, with each army starting with 600 soldiers. This example shows how the skill drastically influences battle outcome.

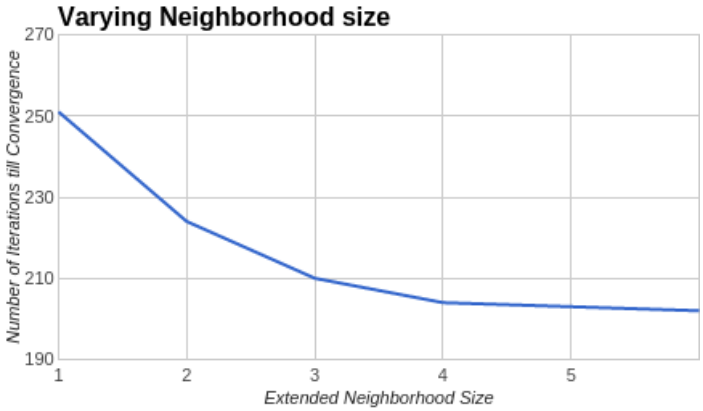


- **Aggressiveness**: We further experimented with the aggressiveness of soldiers. We found that lower aggressiveness (i.e., more defense oriented strategy) is of advantage. The Figure below shows the average number of surviving soldiers for an equal and symmetric battle. The aggressiveness of Army 0 is fixed to 100, while we vary the aggressiveness of Army 1 from 0 to the maximum value of 255. The figure clearly shows that a high aggressiveness is of disadvantage, whereas

a very low aggressiveness has a clear winning advantage. We averaged the results over 50 separate runs using different random seeds. The observed high variation in the plot is due to the small total numbers of surviving soldiers.



- **Extended neighborhood size (ENS)**: ENS represents the distance to which the soldiers can view their surroundings. An increase in the ENS leads to increase in the soldiers' capabilities to form groups and attain happiness. Their movement becomes more target oriented and less random. The figure below shows the influence of the extended neighborhood size on number of iterations that are needed for convergence. The plot was generated for a battle of size 200x200 and 600 soldiers per army.



Notice that the movement of the two armies is guided by the target flag set for the two armies. This means that if the target is set such that the paths of the two armies don't collide then the armies would never meet if the neighborhood size is very small.

- **Dynamic field**: Even though the purpose of dynamic field is for the swordsmen and archers to be able to follow their leaders, its effect was milder compared to

the effect of ENS. In other words, change in ENS had a larger effect in the movement of soldiers compared to the change in dynamic field.

- **Kill radius**: We observed that a larger kill radius for archers helped the armies destroy their opponents faster when compared to a smaller kill radius.

Conclusion

In this project, we build a large scale battlefield simulation based on floor field models. We specialized the floor field model to different types of soldiers and their respective targets in a battle. We further modeled different properties of soldiers, such as aggressiveness, happiness/motivation, skill, and urge to follow leaders (due to the dynamic fields). We build different visualization engines, allowing the generation of video and gif animated battles, as well as, an interactive visualization that allows the user to play and try out different strategies. Moreover, we experimented with the various different parameters and observed the effect on the outcome of the battle. These included different strategies, aggressiveness and skill distribution, the number of each type of soldiers et cetera. For increasing performance, we parallelized our implementation for distributed parallel machines using MPI. Our parallel implementation scales well and reaches good speedups over the basic sequential implementation.

References

1. G. Kende and G. Seres, “Use of chess in military education,” New Challenges in the Field of Military Sciences, 2006.
2. S. J. Banks, “Lifting off of the digital plateau with military decision support systems,” tech. rep., DTIC Document, 2013.
3. T. Lenoir and H. Lowood, “Theaters of war: the military-entertainment complex,” 2005.
4. Burstedde, C., Klauck, K., Schadschneider, A., & Zittartz, J. (2001). Simulation of pedestrian dynamics using a two-dimensional cellular automaton. Physica A: Statistical Mechanics and its Applications, 295(3), 507-525.
5. Kirchner, A., & Schadschneider, A. (2002). Simulation of evacuation processes using a bionics-inspired cellular automaton model for pedestrian dynamics. Physica A: Statistical Mechanics and its Applications, 312(1), 260-276.
6. Simplified Wrapper and Interface Generator, SWIG http://www.swig.org/
7. Python Gizeh Library https://github.com/Zulko/gizeh
8. Python MoviePy Library https://github.com/Zulko/moviepy
9. Python PyGame Library http://www.pygame.org/