Evac Sim: Fall 2020 CSS600

Justin Downes and Chris Smith

December 2020

Abstract

Simulating large scale evacuation is cost-prohibitive in terms of realism and required man-hours. Agent-based simulation supports laying out a floor plan, modeling behaviors and at least capturing some flavor of how a real evacuation might play out. This research uses NetLogo to vary floor plans and capture the average escape times for the agents.

1 Introduction

2 Background

2.1 Previous Work

The literature abounds with previous efforts in this area. [1] [3] [2] [5] [7]

This paper is key as it is extremely similar and a NetLogo implementation. We should know this paper and incorporate into our paper [6]

3 Methodology

Here we give a brief introduction into our approach for tackling the problem. Priorities and trade-offs of our work. layout an intro to the environment, the experiments and our hopeful conclusions

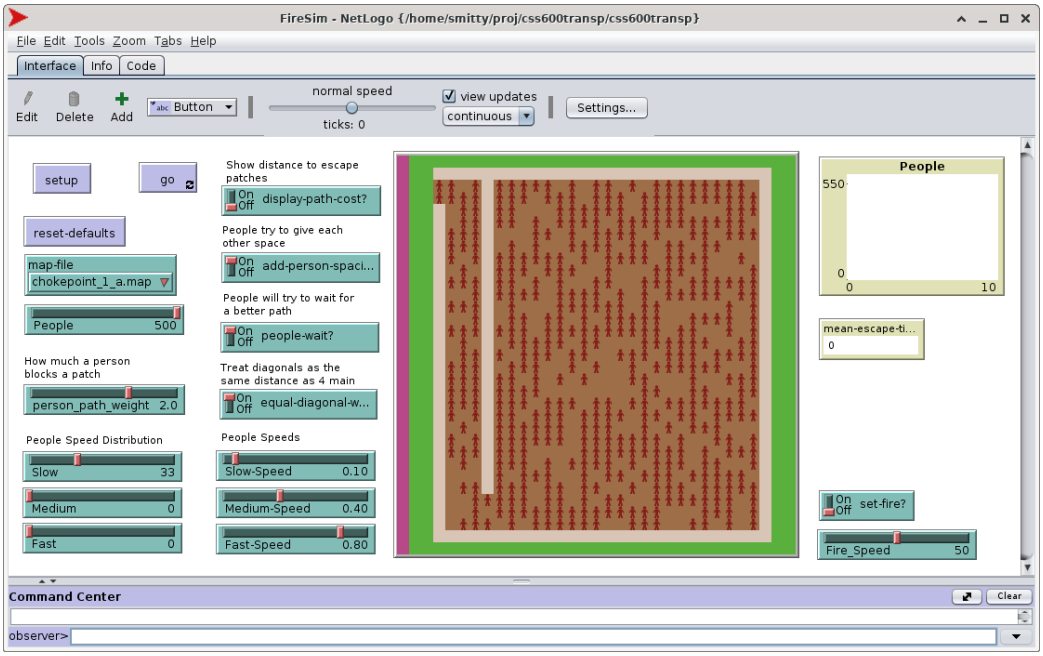


Figure 1: FireSim UI

3.1 Environment

This section talks about the simulation environment. therefor netlogo, the patch world, and loading patches

This paper talks about procedurally generating worlds [4]

need to talk about patch parameters

3.2 Movement Mechanisms

In this section we will go over the mechanisms which dictate how our agents move throughout the environment. Traditionally, in an environment where an agent is moving towards a goal, each agent's path to that goal will be calculated individually through the environment[1]. In our situation we implemented a simplified pathing algorithm which instead calculates global weights for each environmental patch in which all agents flow to the lowest weight. These weights grow in relation to their distance to the safety patches and as

[1]http://www.cs.us.es/ fsancho/?e=131

agents move over them by imparting their own weight to the patch. This simplified pathing allows for much quicker computation of routes and therefor quicker executions of simulations for our experiments.

The following will provide an overview of how the weighting mechanisms work and the choices agents make on whether to move. The foundation of movement is the weight of a given patch, or the cost as described below. The cost metric is defined as the distance from the patch to be considered P and the nearest safety patch P_s. While there may be many safety patches, for the sake of brevity, we assume that P_s is resolved *a priori* as the closest. We can confidently declare this since the weighting implementation iteratively grows from each safety patch such that when it reaches a candidate patch it has been reached by the closest one. This distance cost is also modified by the presence of an agent, $Agent(P)$, on that patch. This agent's weight, A_w, is configurable by the user. Of important note is that the weight is calculated independently of agents that may reside on patches between a patch and the safety patch. This is the simplified pathing mechanism at work in that only local environment calculations are conducted.

$$
\begin{aligned}
cost(P) &= distance(P_s, P) + Agent(P) * A_w \\
Agent(P) &= \begin{cases} 1, & \text{Agent Present} \\ 0, & \text{Agent Not Present} \end{cases}
\end{aligned} \tag{1}
$$

The distance measurement is not as simple as calculating the Euclidean distance. Since our environment is a grid world that is inhabited by blocking obstacles we can only consider movement in either the 4 cardinal directions or the 8 neighboring directions. These two options, which are additionally configurable by the user, determine how the distance between patches are measured. The distance between 2 points P_1 & P_2 is optionally defined by the flag $equal - diagonal - weight$?. If false the distance is the Manhattan distance [2]. If true the distance is the Chebyshev distance [3]. While this algorithm for distance works in an open world it doesn't account for the blocking obstacles. This is another side effect of how the actual implementation works. By growing outward from the safety patches and around obstacles the actual distance calculation is only ever considered for patches that are next to each other. This makes our implementation a narrow case of the algorithm below

[2]https://www.sciencedirect.com/topics/mathematics/manhattan-distance

[3]https://en.wikipedia.org/wiki/Chebyshev_distance

in that all distances we calculate equal 1 and the real choice is how we choose the next patches as either the Manhattan or the Chebyshev neighborhood.

$$distance(P_1, P_2) = \\ equal - diagonal - weight? = \begin{cases} max(|x_1 - x_2|, |y_1 - y_2|), & True \\ |x_1 - x_2| + |y_1 - y_2|, & False \end{cases} \quad (2)$$

Now that we have our patch weights to inform our movement decisions it is time to actually make the decision of where to move to. For this choice we simply choose the neighbor patch that has the lowest cost, where the neighbors are defined as the 4 patches above, below, right, and left of our current patch. Once we know which patch is lowest we take the vector at that patch and multiply it by our agent's speed. So, for a given agent A the vector to move is given by the vector of the lowest cost neighbor patch times our agent's speed.

$$move(A) = V(min(cost(neighbors_4(A_p)))) * S_a \quad (3)$$

$$neighbors_4(P) = \{patch(P_x - 1, P_y - 1), patch(P_x + 1, P_y - 1), \\ patch(P_x - 1, P_y + 1), patch(P_x + 1, P_y + 1)\} \quad (4)$$

We have an additional parameter to account for situations where the cost of all neighboring patches is greater than the current patch. The flag $people - wait?$ allows simulators to decide whether or not to stay put and wait for a better patch or to always move even if the new patch has a higher cost. This flag redefines the $move(A)$ function as $move(A)$ if the flag is false and if the cost of the current patch is less than $move(A)$ then to not move.

$$people - wait? = \begin{cases} min(cost(A_p), move(A)), & True \\ move(A), & False \end{cases} \quad (5)$$

Another phenomena we wished to capture is the ability for agent's to give each other space. This is configurable through the $add - person - spacing?$ flag. When the flag is true the cost of a patch is redefined to be the normal cost plus the sum of all the neighbors with agents times the agent's weight divided by 10. This scaling factor of 1/10 has been determined through

trial and error and is something that could be exposed to user control in the future. For ease of implementation and increased computation time the actual calculation is done from the perspective of the agent in that each agent has their scaled weight applied to its patch neighbors.

$$add - person - spacing? = \begin{cases} cost(P) + \sum Agent(neighbors_4(P)) * A_w/10, & True \\ cost(P), & False \end{cases} \tag{6}$$

The grid world, while maybe too much of a simplification of real world dynamics, enables us to implement complex behaviors through simple straightforward algorithms. These complex behaviors have enabled us to conduct a few interesting experiments described in the following sections.

4 Experiments

This should be about the underlying approach to devising and goals of our experiments

4.1 Experimental Environment

This section describes our experiment harness that we built using NetLogo's Behavior Space functionality.

Behavior Space supports supplying simulation arguments via an XML document, invoking the NetLogo engine via a script pointing to the XML, and then capturing the results via the standard output.

This lends itself to scripting via Python [4]. The goal had been to extract the initial Behavior Space XML content directly from the .nlogo file, and then craft a SQLAlchemy [5] model on the fly that would support storing results in an RDBMS, e.g. SQLite [6]. That proved out of reach due to the advanced nature of SQLAlchemy, so a hand-crafted model was used, which makes alterations to the underlying model more maintenance intensive.

[4] https://www.python.org/
[5] https://www.sqlalchemy.org/
[6] https://sqlite.org/index.html

While we stored the results in a local SQLAlchemy file, a mere update to the connection string would allow storing results in an enterprise RDBMS to good effect.

Another Open Source tool that was used extensively was PyTest [7]. Billed as a unit testing framework, PyTest supports breaking the problem down into granular fixtures and then combing them in a Lego-like fashion that lends itself to the problem space. For example, while Behavior Space allows stepped alteration of numerical parameters in a model, swapping out map file names is not directly supported. Implementing a Python function to generate the map file names and then re-writing the XML was far more convenient than having distinct experiments for each map.

And then Python's data science facilities are well-known [8], supporting arbitrary visualization pipelines.

4.2 Experimental Parameters

For the map files in use, there were 10 runs against the map files, capturing mean-escape-time for the agents.

VARIABLE	MEANING / VALUE
map-file	Name of map file to set up.
People	Number of people. Held constant at 500.
person_path_weight	Agent blockage factor for patch
Slow	Percentage moving at this rate, which we set to 100
Medium	Set to 0
Fast	Set to 0
Slow-Speed	0.3 patches
Medium-Speed	0.75 patches
Fast-Speed	1.0 patches
add-person-spacing?	true
equal-diagonal-weight?	true
display-path-cost?	false
people-wait?	true
set-fire?	false
Fire_Speed	50
mean-escape-time	output

[7] https://docs.pytest.org/en/stable/

[8] https://www.scipy.org/index.html

4.3 Experiments Based on Layouts

This section will describe the different layout based experiments. How we set them up to narrow parameters to just measure the layouts impact

4.3.1 Chokepoint Experiment

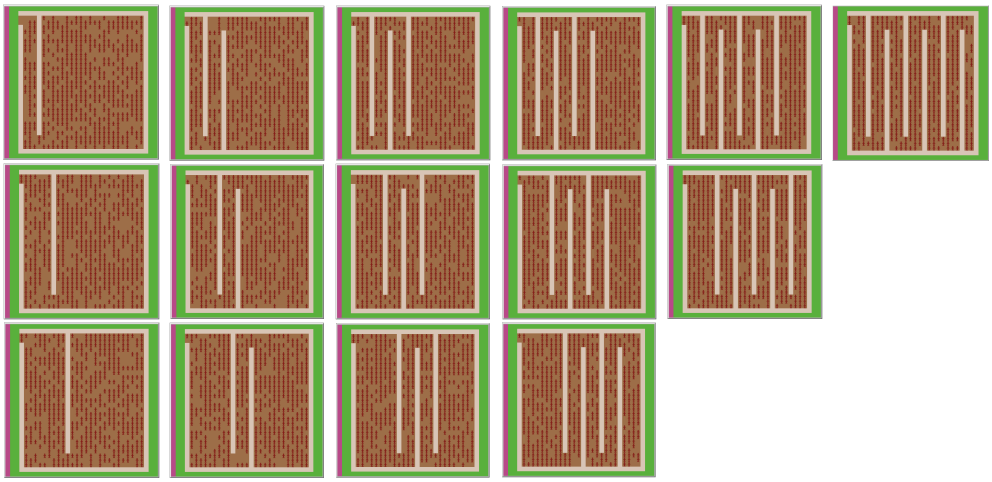


Figure 2: Chokepoint Maps

4.3.2 Exit Dimensions

4.4 Experiments Based on Agent Features

This section will cover our experiments based on modifying the agent's speed distributions.

4.5 Experiments Replicating Emergent Crowd Behaviors

- mainly from this paper [1] . This section will cover how we hope to see these emergent behaviors in our environments even though we use a simplified pathing strategy. This could lead to simpler models with less resource requirements.

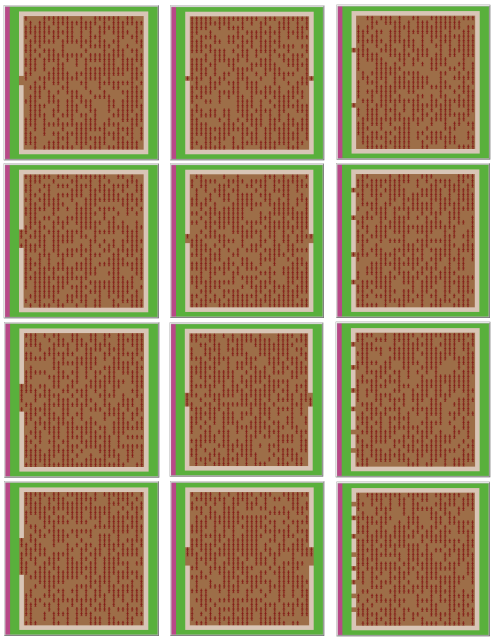
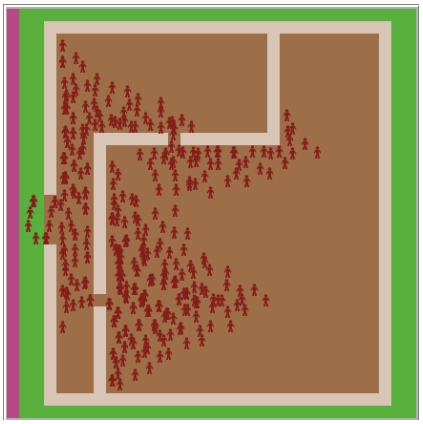
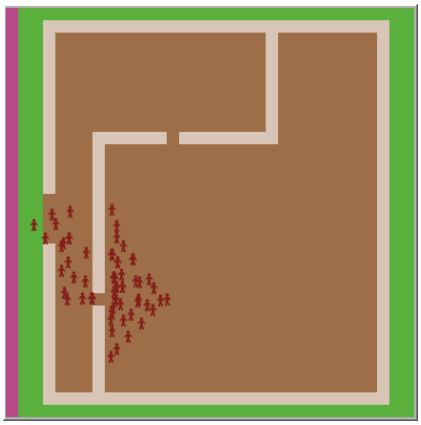


Figure 3: Exit Dimensions Maps

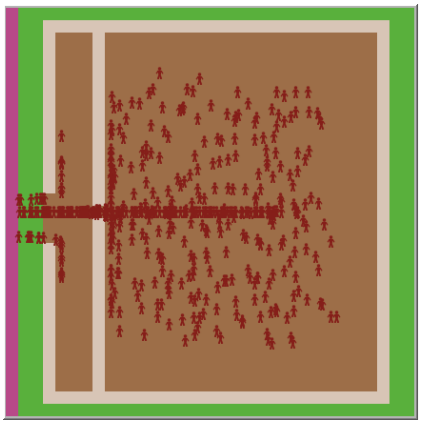


(a) Evacuation with no herding

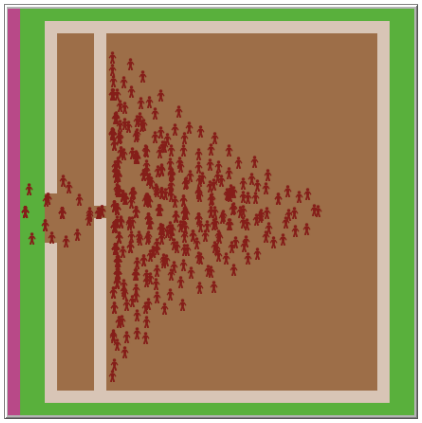


(b) Evacuation with herding (i.e. ignoring viable exit)

Figure 4: Herding behavior replicated through residual escape pathing from agents that never considered the alternative route.



(a) No clogging due to people not blocking paths



(b) Clogging due to increased weight per person on a patch

Figure 5: Clogging behavior replicated through a weight attribute calculated for each patch an agent is on as well as a configurable additional weight for each agent's neighboring patch.

if we can show that we achieve similar results even though we use a simplified pathing algorithm and abm environment i think that would be insightful

5 Results

The results showed that adding more length to the chokepoint pipeline did not result in a linear increase in the exit time. Rather, the Actors resembled a fluid dynamics problem, reaching a uniform flow and making good their escape.

More exits is better, especially if not on the same side, but there is not a linear increase with additional exits.

	map	MIN	AVG	MAX	VAR
	a	966.4	1006.0	963.0	995.6
	b	1155.6	1190.0	1149.0	1181.6
chokepoint1	c	1207.0	1243.0	1198.0	1234.9
	d	1217.5	1234.0	1210.0	1231.1
	e	1202.4	1243.0	1200.0	1226.0
	f	1183.1	1212.0	1179.0	1206.3

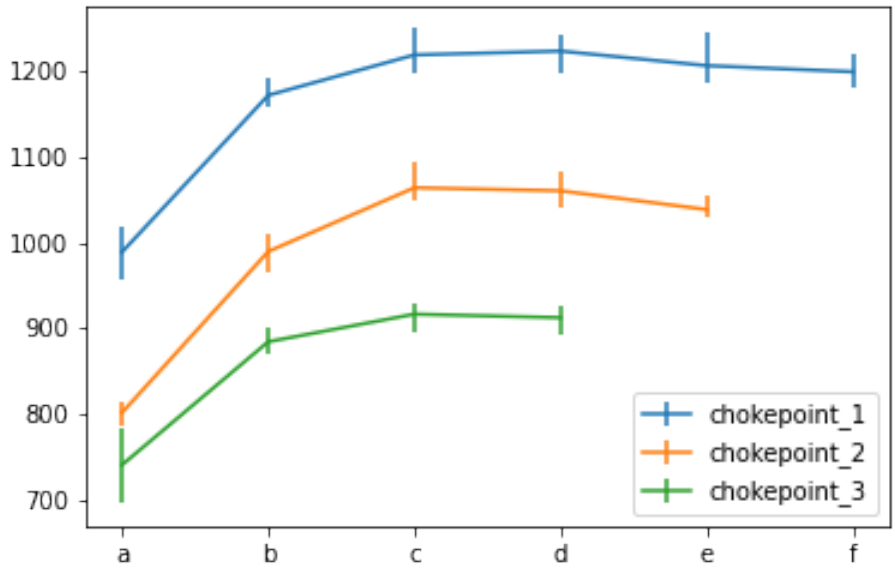


Figure 6: Chokepoint Results

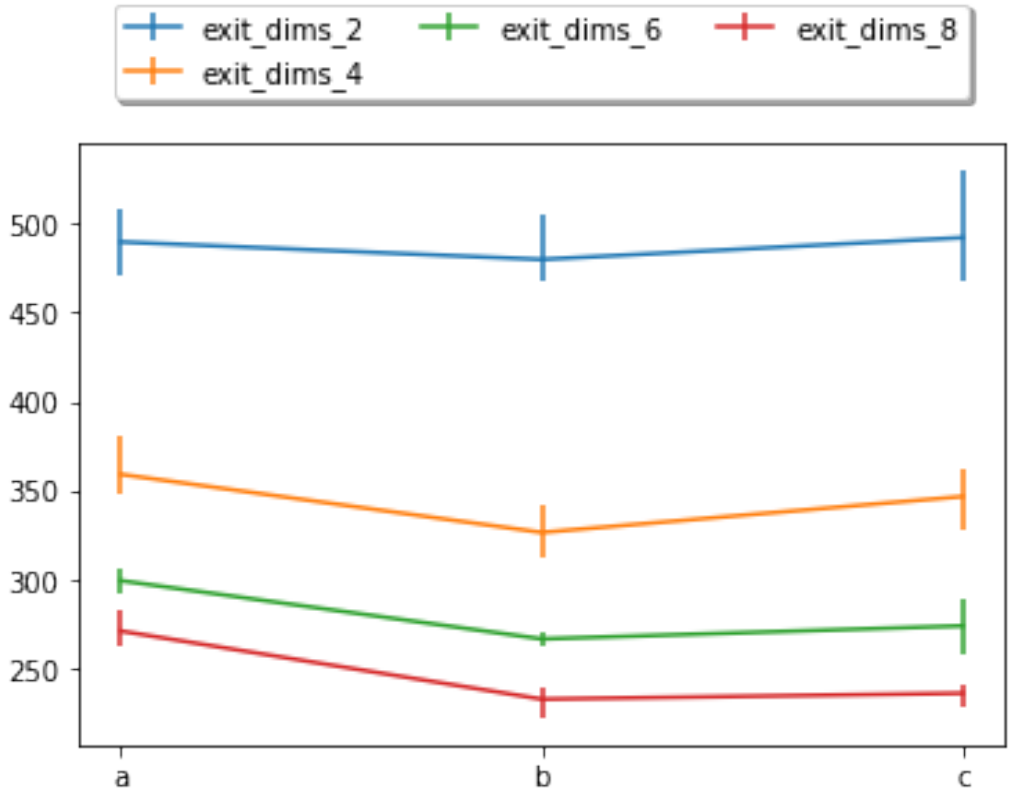


Figure 7: Exit Dimensions Overview/Results

chokepoint2

map	MIN	AVG	MAX	VAR
a	790.8	833.0	781.0	819.8
b	975.7	1025.0	970.0	1015.7
c	1036.3	1071.0	1017.0	1071.5
d	103 9.3	10710	134.0	165.5
e	10 34. 8	1055.0	102.0	105.2

chokepoint3

map	MIN	AVG	MAX	VAR
a	717.6	758.0	704.0	751.9
b	886.9	919.0	880.0	911.2
c	905.2	952.0	900.0	937.3
d	902.3	925.0	894.0	919.2

extdims2

map	MIN	AVG	MAX	VAR
a	480.6	516.0	472.0	502.5
b	482.9	522.0	474.0	510.0
c	481.3	524.0	473.0	510.6

exitdims4

map	MIN	AVG	MAX	VAR
a	348.4	368.0	342.0	367.3
b	319.2	344.0	317.0	336.3
c	334.3	365.0	328.0	358.8

exitdims6

map	MIN	AVG	MAX	VAR
a	292.0	302.0	287.0	301.7
b	262.0	285.0	258.0	277.9
c	265.8	290.0	265.0	280.3

exitdims8

map	MIN	AVG	MAX	VAR
a	264.7	279.0	263.0	276.2
b	230.4	243.0	228.0	240.3
c	231.4	250.0	231.0	242.7

6 Future Work

Here we can expand on improvements we would make, additional features we could add (smoke multi-floor etc.), additional experiments, and porting to other environments

7 Conclusion

The simpler the layout, with more and diversified exit options, the more optimal the evacuation results.

References

[1] João E. Almeida, Rosaldo J. F. Rosseti, and António Leça Coelho. Crowd Simulation Modeling Applied to Emergency and Evacuation Simulations using Multi-Agent Systems.

[2] E.D.Kuligowski and S.M.V.Gwynne. The need for behavioral theory in evacuation modeling.

[3] Angelika Kniedl, Dirk Hartmann, and Andre Borrmann. Using a multi-scale model for simulating pedestrian behavior.

[4] Maysam Mirahmadi and Abdallah Shami. A Novel Algorithm for Real-time Procedural Generation of Building Floor Plans.

[5] Fangqin Tang and Aizhu Ren. Agent-based evacuation model incorporating fire scene and building geometry.

[6] Eileen Young. Prioritevac: An agent-based model of evacuation from building fires.

[7] Jibiao Zhou, Yanyong Guo, Sheng Dong, Minjie Zhang, and Tianqi Mao. Simulation of pedestrian evacuation route choice using social force model in large-scale public space: Comparison of five evacuation strategies. 14(9):e0221872–e0221872.