Advancing the Layered Approach to Agent-based Crowd Simulation

Bikramjit Banerjee, Ahmed Abukmail and Landon Kraemer
School of Computing
The University of Southern Mississippi
118 College Drive #5106
Hattiesburg, MS 39406-0001
{Bikramjit.Banerjee, Ahmed.Abukmail, Landon.Kraemer}@usm.edu

Abstract

We adapt a scalable layered intelligence technique from the game industry, for agent-based crowd simulation. We extend this approach for planned movements, pursuance of assignable goals, and avoidance of dynamically introduced obstacles/threats, while keeping the system scalable with the number of agents. We exploit parallel processing for expediting the pre-processing step that generates the path-plans offline. We demonstrate the various behaviors in a hall-evacuation scenario, and experimentally establish the scalability of the frame-rates with increasing number of agents.

1 Introduction

Crowd behavior simulation has been an active field of research [5, 15, 8, 2, 3] because of its utility in several applications such as emergency planning and evacuations, designing and planning pedestrian areas, subway or rail-road stations, besides in education, training and entertainment. In agent-based crowd simulations, where each pedestrian is modeled as an autonomous agent, a tradeoff is commonly made between the complexity of each agent and the size of the crowd. This is because, by common wisdom "simple characters are more efficient to evaluate, but complex characters can capture more realistic crowd behaviors" [11]. The assumption underlying the above quote is that realistic crowd behaviors are hard to achieve with simple agent models. Although we only focus on navigational behaviors in this paper, we show that it is possible to model complex behaviors realistically (such as static obstacle avoidance, separation, collision avoidance, approaching assignable goals, and avoidance of dynamically introduced obstacles/threat) with an extremely simple agent model, leading to a scalable simulation system. The main idea is to distribute the intelligence in the terrain [14] rather than accumulating it into a complex/bulky model that each agent must follow. Although this idea of *smart terrain* is not new, to the best of our knowledge, this is the first application of this idea to crowd simulation. More importantly, we advance this approach to incorporate new behaviors that are specific to crowd simulation.

In this paper, we focus on crowd movement on a 2D surface. We use the layered AI framework [14] to create an efficient platform for agent movement, that is also easily expandable to incorporate more and more complex behaviors at will, by simply adding more layers. We first create a flow-field for basic agent movement, avoiding static obstacles in the world, using the Markov Decision Processes (MDP) [12] framework. We show that realistic behavior in this context needs a refinement that *Semi-Markov* Decision Processes (SMDP) offer. We also show how the combination of (S)MDPs and layered AI allows us to easily handle the assignment of different goals to different agents. This means an agent is not limited to approaching the *nearest* goal, but an *assigned* goal, unlike what the (S)MDP framework alone offers. We also extend the layered AI framework to handle the dynamic introduction of new obstacles/threats. One limitation of our approach is the preprocessing time for creating the initial flow-field. We show results that this step can be parallelized to reduce the precomputation time. Finally, we show the frame-rates resulting from our implementation, which clearly establishes the efficacy of our scalable approach in modern crowd simulation.

2 Layered Intelligence

We consider crowd behavior in an environment created on a 2-D surface. We divide the surface into square grids, where each cell has a sufficient area to hold no more than one person of average size. We have used the concept of layered AI from the game industry [14] for crowd simulation in this environment. The basic idea is to distribute ter-

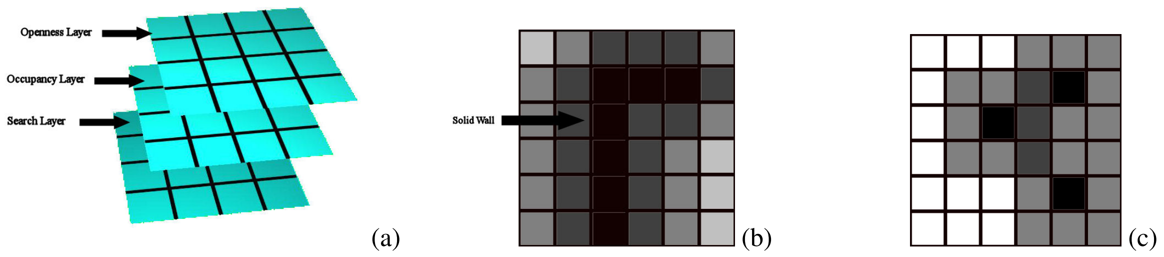


Figure 1. (a) Several informational layers overlay the underlying physical grid, (b) The openness/obstacle layer, (c) The occupancy layer. Black cells are occupied, progressively lighter cells are more easily walkable. These figures are adapted from [14] to illustrate the layered intelligence approach.

rain and other navigation-related information into several layers and have an agent make simple navigation decisions based on a combination of these layers. For instance, there could be a single layer called *occupancy layer* where each agent enters its current position. When an agent makes a decision of which cell to move to next, it will need to consult this layer and omit any neighboring cell that is already occupied by other agents. Once its decision is made, it will need to update its position on this layer, for other agents to avoid collision with this agent. Similarly there could be an *obstacle layer*, which contains information about all static obstacles in the environment. When deciding which neighboring cell to move to next, an agent must also consult this layer to omit cells that are blocked by obstacles. Rather than binary (blocked or available) values, the layers usually contain values from a continuous range to indicate proximity to agents/obstacles. The approach is illustrated in Figure 1.

Essentially, each type of information that is relevant to navigation is captured in a separate layer. In games, informations such as which cells are easily visible and hence open to enemy fire, which cells have the enemies just searched and are not likely to search again anytime soon, etc, are captured in separate layers, called *openness layer, search layer* etc. [14]. Normalized values (i.e., in the range $[0, 1]$) are stored for each cell in each layer, reflecting its value from that layer's perspective. Let $layer_i(x, y)$ be the value of cell (x, y) in layer i, with a total of L layers, $i = 1 \ldots L$. An agent at location (x, y) needs to simply look up the values of all cells in the neighborhood of (x, y), i.e., $N(x, y) = \{(p, q)|(p, q) = Neighbor(x, y)\}$, *from all layers* and pick the best next-cell as

$$(p, q)_{best} = \arg \max_{(p,q)\in N(x,y)} \prod_{i=1}^{L} layer_i(p, q) \qquad (1)$$

In this paper, we use simple formulae to compute openness/obstacle and occupancy layers (Figure 1 (b) and (c)). The obstacle layer is simply binary (0's occupied by walls, and 1's open) in contrast to Figure 1 (b) which shows a larger range of values (grey levels), while the occupancy layer is computed as (conforming to Figure 1 (c))

$$layer_{occupancy}(x, y) = \begin{cases} 0 & \text{if agent at } (x, y) \\ 0.5^k & k = \text{ number of agents} \\ & \text{in } N(x, y) \end{cases}$$

The above formula for the occupancy layer is actually implemented as a constant-time process per agent, and encourages slight (just one cell-deep) separation among agents (as shown in Figure 1(c)), unless they are pressed in a congestion. In every new frame, an agent only needs to update the neighborhood of its new location, and re-adjust the neighborhood of its previous location, besides the two successive locations.

The complexity of the decision-making process (equation 1) is $O(|N|L)$. Hence with a fixed sized neighborhood (say the 9 cells surrounding and including (x, y)), the decision would be constant time. With n agents in the environment, the total time complexity for generating the next frame of agent positions would be $O(n)$, which is the best possible speed we can hope for in a distributed simulation. The layered approach to intelligent decision-making abides by the following principle that game AI developers value for scalability and efficiency: *put the intelligence in the data, not in the code*.

3 Path planning

In this paper, we extend the layered approach to include path planning to allow planned movements as opposed to reactive movements. We also allow different destinations for different agents, and build a *path-plan layer* for *each* destination. An agent will thus consider only the path-plan layer for its desired destination and ignore the other path-plan layers. As an illustration consider an agent egressing a building to approach one of four surrounding parking lots, where he has actually parked. We can have a path-plan layer for each of the four parking lots, and have many agents approach their respective parking lots concurrently by consulting the appropriate layers. The success of this approach obviously depends on the number of destinations being small. The advantage is *open design*, i.e., we can extend and complexify the scenarios arbitrarily, simply by building extra layers.

In games, path planning is usually done offline (i.e. precomputed) with Floyd-Warshall technique, or online with A* for dynamic path-finding. In the layered approach, we are primarily concerned with minimizing run-time processing; hence we completely eliminate the need for A* with the intent of handling dynamic changes to the optimal path entirely within the layered framework. Although the Floyd-Warshall technique would give us path plans for all start-end cell pairs, the format of the output is not quite conducive to the layered approach. We need a path-planning technique that will produce a flow-field that tells an agent which neighboring cell it should move to, given its current cell location. Furthermore, these decisions should be based on real numbers that can be meaningfully combined (using equation 1) with other layers to formulate a more informed movement decision that is also based on occupancy, static obstacles and possibly dynamic obstacles. Another major limitation of the Floyd Warshall algorithm is that it only gives the optimal decision from any cell, but fails to offer an alternative if that optimal move is impossible to make (e.g., because another agent is currently occupying that cell). These limitations force us to look beyond Floyd-Warshall technique.

3.1 Markov Decision Processes

We view movement on the 2-D grid as a Markov Decision Process (MDP). Formally, an MDP is given by the 4-tuple $\langle S, A, T, R\rangle$, where S is the set of environmental states that an agent can be in at any given time, A is the set of actions it can choose from at any state, $R : S \times A \times S \mapsto \Re$ is the reward function, i.e., $R(s, a, s')$ specifies the reward from the environment that the agent gets for executing action $a \in A$ in state $s \in S$ leading to state s'; $T : S \times A \times S \mapsto [0, 1]$ is the state transition probability function specifying the probability of the next state (s') in the Markov chain consequential to the agent's selection of an action (a) in a state (s). An MDP solver agents goal is to learn a policy (action decision function) $\pi : S \mapsto A$ that maximizes the sum of discounted future rewards from any state s,

$$\begin{aligned} V^\pi(s) &= E_T[R(s,\pi(s),s') + \gamma R(s',\pi(s'),s'') \\ &\quad +\gamma^2 R(s'',\pi(s''),s''') + \ldots] \end{aligned}$$

where $s, s', s'', s''', \ldots$ are samplings from the distribution T following the Markov chain with policy π, and $\gamma \in [0, 1)$ is the discount factor.

```
1:  Input real $\epsilon$
2:  Initialize $V(s) \leftarrow 0, \forall s \in S$
3:  **repeat**
4:      $\Delta \leftarrow 0$
5:      **for** each cell, $s \in S$ **do**
6:          $v \leftarrow V(s)$
7:          $V(s) \leftarrow \max_a[R(s, a, T(s, a)) + \gamma V(T(s, a))]$
8:          $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
9:      **end for**
10: **until** $\Delta < \epsilon$
```

Table 1. The value iteration algorithm from [12]

In context of the 2-D grid, S is the set of all grid-cells. The action set A is the set of 9 cells that an agent at location (x, y) can move to (including the action of staying put in cell (x, y)). Some of these neighboring cells may be blocked by static obstacles, in which case the corresponding actions are unavailable, and excluded from set A (or equivalently, the corresponding transition probabilities $T(.,.,.)$ are set to zero). We use the information from the static obstacle layer to compute the MDP solution, so that the resulting path-plans avoid them in a realistic manner [1]. We use value iteration [12] algorithm to solve the MDP induced by the 2-D grid with the reward function defined simply as

$$R(s, a, s') = \begin{cases} 1 & \text{if } s' \text{ is a goal cell} \\ 0 & \text{otherwise} \end{cases}$$

and transition function (T) is deterministic, meaning an action a in state s leads to a unique next-state (s') everytime. Therefore for our purpose, we can redefine T as

$$T : S \times A \mapsto S$$

i.e., $T(s, a) = s'$ in accordance with the above. The value iteration algorithm is shown in Table 1. The result (i.e., $V(s), \forall s \in S$) of this process can be readily used by an agent to select an appropriate action in any state by

$$\pi(s) = \arg\max_a V(T(s, a))$$

If several actions have the same maximum value, then teh agent can pick one of these at random.

3.2 Semi-Markov Decision Processes

Oftentimes, different actions take different amounts of time to complete. For instance, when an agent wants to

[1]This means an agent starts avoiding an obstacle before he actually encounters it in his immediate neighborhood, simulating vision-based avoidance, much like A*

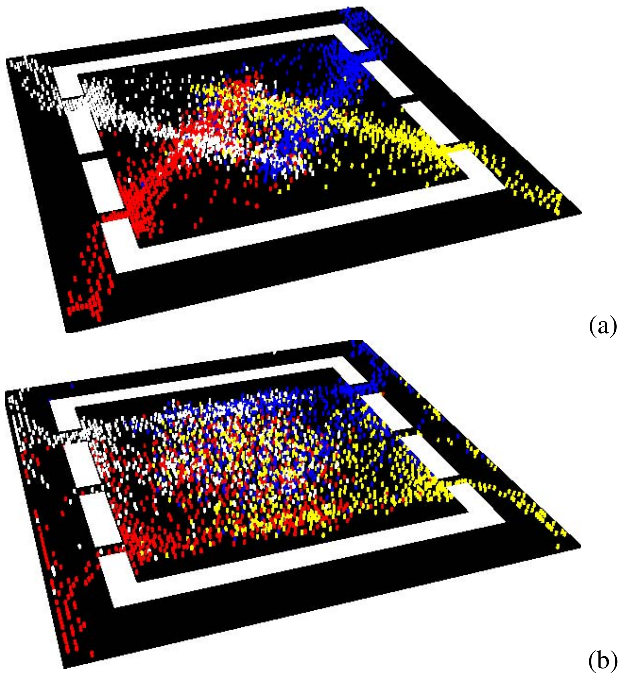


Figure 2. Crowd behavior without (a) and with (b) differing digonal cost taken into account

move from cell (x, y) to cell $(x + 1, y)$ (or any of the four non-diagonal neighboring cells), the distance covered (say centroid to centroid) is less than if the agent wanted to move to $(x + 1, y + 1)$ (or any of the diagonal neighboring cells). In particular, if the former distance is 1 unit, the latter is $\sqrt{2} = 1.414$ units, assuming square grid cells. Hence the agent would take longer to execute a diagonal step than a Manhattan step. This discrepency in action times can be neatly incorporated in the MDP formalism to produce *Semi-Markov Decision Processes* , where the step 7 in the algorithm in Table 1 needs to be modified to

$$V(s) \leftarrow \max_a [R(s, a, T(s, a)) + \gamma^{t(a)} V(T(s, a))]$$

where $t(a)$ is the time taken to execute action a. Figure 2(a) shows the result of ignoring the difference in action execution times (i.e., $t(a) = 1, \forall a$), while Figure 2(b) shows the result of taking these differences into account. In this simulation, 500 agents are placed in a hall with 6 exits, in a grid-world of size 137×137. The 4 corners of the grid-world are 4 different goals that an agent could go to. Each agent is colored by the goal that is randomly assigned to it. This simulation uses 4 path-plan layers (one for each corner goal) and an occupancy layer, but since we have used a binary static obstacle layer (in contrast with Figure 1(b)), this layer can be eliminated after the information needed to handle static obstacles is captured in the path-plan layers (e.g., notice that the path-plan layer in Figure 3 contains binary static obstacle information). In Figure 2(a) the agents line-up in the direction of their respective goals (which we believe is an unrealistic artifact), and also fail to use the middle exits. In Figure 2(b), both of these problems have been eliminated by using the SMDP formulation. Figure 3 illustrates the difference between the flow fields produced by the two methods. In this figure, the map has a lower

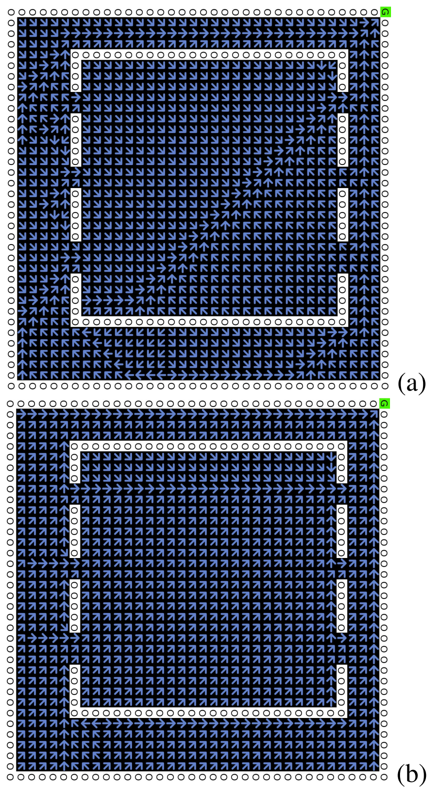


Figure 3. Navigation flow-field produced by the algorithm in Table 1 (a), and that produced with the SMDP adjustment (b), for the path-plan layer with goal at top-right corner

resolution (36×36) for the sake of clarity, and the static obstacles are all one cell deep (unlike in Figure 2), marked with 0's. The band of 0's around the periphery serves to establish the world's boundary. The goal cell is shown in green at the top right. It is worth noting from Figure 3 that while the raw MDP method pushes the agents away from the right-hand exits when in their vicinity, the SMDP adjustment nudges them toward those exits. This is the main reason for the agents being able to use the right-middle exit when in it's vicinity.

One important limitation in this regard is that the agents may be unable to use the middle exits if they are far from them, while the other exits are temporarily congested with other agents. While the occupancy layer enforces some separation among agents and as a by-product may be sufficient (in some cases) to push the agent toward such unused (or scantily used, hence less crowd producing less separation force) exits, we would like to solve this problem without banking on separation. As a first-cut approach, this scenario calls for replanning which is expensive with the layered technique. One possibility is to treat a congestion as a dynamically placed obstacle (see next section for how we handle such obstacles), but the key difference is that while we may be told where a dynamic obstacle has been placed, the location of a congestion must be autonomously determined. This is an important future direction.

4 Dynamically placed obstacles

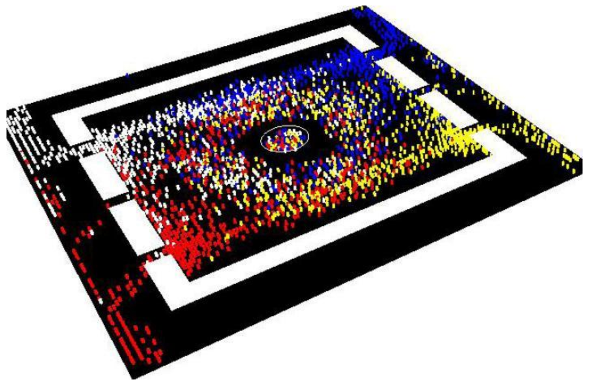
A second extension to the layered approach that we propose, is to handle obstacles that are added dynamically. We use the word "obstacle" in a broader sense, to refer to anything that an agent would want to avoid during navigation. For instance, rubbles created by an explosion, agents that died in stampede, a raging fire etc. are all considered obstacles that a user adds after a simulation starts. As mentioned in the previous section, we would also like to treat a temporary congestion at a desired exit to be an obstacle that agents (with that exit on their path) must avoid, but a user will not be expected to locate it. All such obstacles necessitate dynamic changes to the optimal path plans. The layered framework enables a simple solution to this problem, albeit in a limited way. The idea is to create a new layer for a dynamically added obstacle, instead of modifying the path plan layer. The new layer will create a *a trough* of values in and around the location of the obstacle, so that when combined with the other layers (equation 1), the agents will avoid bumping into it. In our implementation, for the purpose of simplicity an obstacle is characterized by 5 parameters:

- (c_x, c_y): the coordinates of the center of the obstacle
- r_i: inner radius, that is the physical extent of the obstacle. For obstacles that are arbitrarily shaped, this is the radius of the bounding sphere.
- r_o: outer radius ($> r_i$), that is the extent of the influence of the obstacle. Normally all obstacles extend their influence as far as visual distance, i.e., an agent that can see it will repath to avoid it. However, for some obstacles the influence could extend further; for instance, the effects of a fire or an explosion can be perceived from locations that are further away, and hence have larger r_o.
- a: avoidance intensity, that specifies how strongly an agent would want to avoid stepping close to the obstacle. If $a = 0$, an agent can walk right by the obstacle (without stepping on it), whereas for higher a, an agent would want to avoid it from a larger distance, e.g., a fire.

In this *dynamic obstacle layer*, the value of a cell (p, q) is computed as

$$layer(p,q) = \begin{cases} 0 & \text{if } d \le r_i \\ \left(\frac{d-r_i}{r_o-r_i}\right)^a & \text{if } r_i < d < r_o \\ 1 & \text{if } d \ge r_o \end{cases} \qquad (2)$$

where d is the distance between (p, q) and (c_x, c_y).


(a)

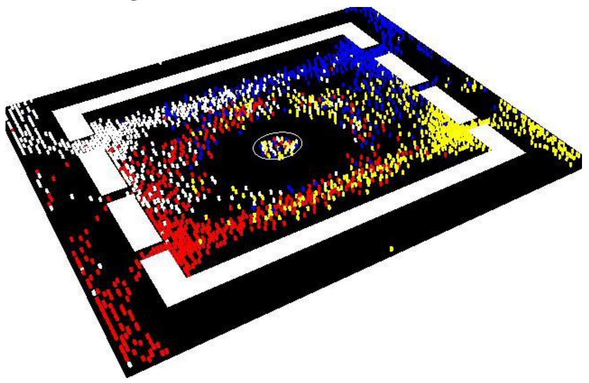

(b)

Figure 4. Two successive snapshots taken every 2 seconds after an obstacle is dynamically placed

Figure 4 shows two successive snapshots (a and b, roughly 2 seconds apart) from placing an obstacle at the center of the room. We have developed this tool to allow a user to place a circular obstacle with an adjustable r_i, at an arbitrary location with a mouse-click. All agents located within this obstacle immediately become immobilized, simulating dead agents [2] from whatever hazard (explosion, fire etc) originated the obstacle. In this figure, $r_o = 4r_i$, and

[2]This is implemented quite naturally in the layered framework, since unless an agent within the obstacle is in the periphery, it will be surrounded by 0 valued cells, according to equation 2, and hence unable to move.

$a = 1$. It should be noted that larger values of a enforces r_o more strongly making the agents form a more sharply-defined circular pattern of avoidance. In contrast, a lower value (here $a = 1$) makes the circular outline more diffuse which we believe is more realistic. It is also noteworthy that Figure 4(a) is a snapshot roughly 2 seconds (run on a Lenovo Thinkpad 2.6 Ghz dual core machine with 3GB RAM) after the obstacle was placed, which includes the time taken to compute the new dynamic obstacle layer. On the said machine, the delay from this computation is nearly imperceptible on a 137×137 grid.

A major advantage of a separate dynamic obstacle layer is the ease of handling temporary obstacles. Recall that one of our future goals is to use this technique to handle temporary congestions at bottlenecks. As such, when a temporary obstacle disappears (e.g., a fire dies down, or rubbles are removed by emergency response personnel, or a bottleneck crowd dissipates), the agents must resume their original path plans. In the layered framework, it amounts to simply deleting the dynamic obstacle layer, since the original path-plan is never modified. However, this also leads to a major limitation of this approach, during the period that the obstacle exists. Since the path plan is not modified taking the existing static obstacles (such as walls) into account, it is possible that even though an agent is on the other side of a wall relative to the obstacle, it still diverts along a curve if the avoidance field extends that far. In order to avoid this undesirable effect, it is necessary to modify equation 2 to take the static obstacles into account. We leave this enhancement for future work.

A minor limitation is that when an obstacle disappears, the (presumed dead) agents located within r_i will resume movement. This can be prevented simply by tagging all agents immobilized within r_i, as dead agents. A dead-flagged agent will never move at any future time. Another limitation in case of large maps is the fact that an entire layer gets created for an obstacle that occupies only a small area. One possible solution could be to create a sub-layer for only the area covered by the influence field of an obstacle, and combine it with other layers much like applying a filter in image processing. We intend to evaluate this approach in the future.

5 Parallelizing the Pre-Processing Step

The pre-processing step to produce the flow-field for each goal in a separate layer (i.e., the algorithm in Table 1 applied once for each goal), is computationally intensive and therefore it can benefit from improvement. In order to improve the pre-processing time, we have parallelized the pre-processing step. The parallel algorithm is quite simple, yet it shows significant improvements, especially as the size of the map grows. The idea is to allow for the path-plan for each goal layer to be processed separately on each processor in the parallel platform, whether a multi-core machine or on a cluster of machines. The advantage of having multi-core machines is that communication cost is kept to a minimal . However, if the implementation is done on a cluster of machines, communication of the results would be slightly more expensive. Since we have used four goal layers in

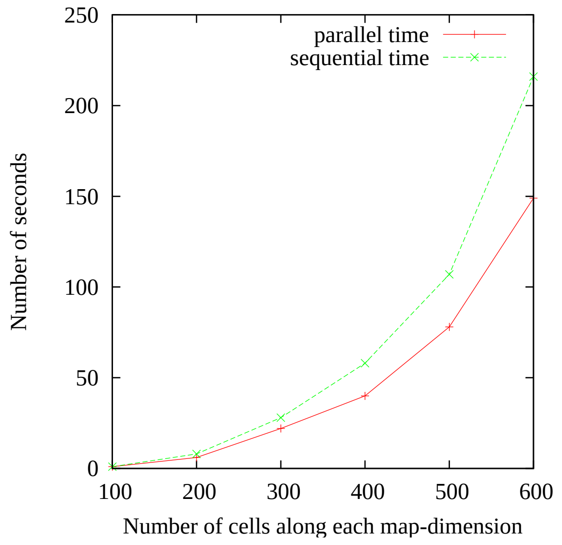


Figure 5. Plot of runtimes for the path-plan layers for all four corner goals (Figure 2), run sequentially and in parallel, for various map sizes

our implementation, we required four separate processors, each handling a separate goal. The computation for each of the layers is independent of the others, because the path plan for one goal is independent of the other goals. With the availability of quad-core machines and large amounts of memory, there was no need to use a cluster of computers to implement our algorithm. We distributed the load equally among all four CPUs. Since each one of the CPUs can access the memory, each CPU was able to write its results to a its own memory segment. As a result of all processors being on the same machine, communication cost was extremely low.

In our implementation, we used a cluster of machines in our Advanced Visualization lab. However, we only utilized a single machine, since all we needed was a quad-core machine. We used an implementation of the Message Passing Interface (MPI) to implement our parallel algorithm.

Figure 5 shows the number of seconds taken to genenerate the four path-path layers, using the parallel and the plain sequential methods, on varying map sizes. The maps were of the same form as the previous figures, but the sizes of the obstacles grew proportionately with the map size. Firstly, Figure 5 verifies a roughly cubic trend that is expected, because the loop 5 in Table 1 is $O(m^2)$ (m being the number of cells along each dimension of the map), while the loop 3 has an average complexity of $O(m)$ [3]. We also notice in Figure 5 that the amount of computation required by each processor was *not* reduced by 75% (as one might expect) of the sequential algorithm. This is mainly due to the $O(m^3)$ complexity of the main loop, and also the dominance of the loop overhead. However, the increasing benefit of parallelizing the pre-processing step is apparent from Figure 5, when the size of the map grows. To put in perspective, a 1200×1200 map covers approximately a quarter of a square mile according to the proportions used in our system, which roughly corresponds to the size of a football stadium. As Figure 5 shows, an area one quarter of that size (i.e., 600×600, or one quarter of a stadium) takes 2.5 minutes pre-processing time which is a compelling factor in favor of the layered approach, and this is made possible by parallelization. Sucar [10] discusses how an MDP can be broken into subtask-MDPs that can be solved in parallel with limited communication across processors for synchronization. We believe that this technique will improve the speed-up of the parallel version, since we will be parallelizing not only the path plans for different goals, but also the path plan layer for individual goals.

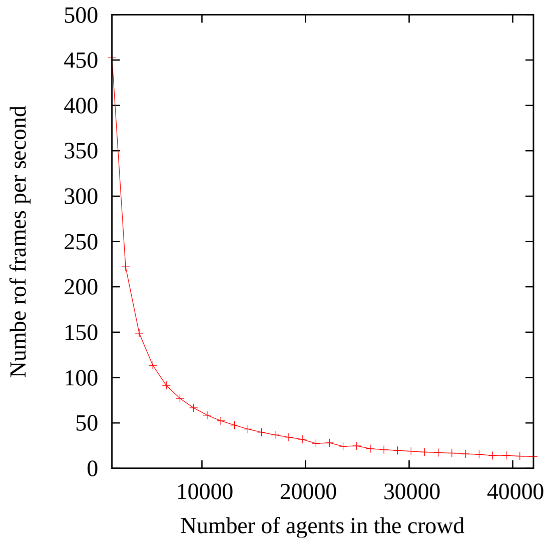


Figure 6. Plot of frame-rate against the crowd size

[3]Consequently, the Value Iteration algorithm is no more expensive than Floyd-Warshall algorithm on average maps, i.e., unless the maps are unusual involving $O(m^2)$ (or worse) path lengths among various locations.

6 Evaluation

In order to establish the scalability of this approach, we have run experiments on a fixed grid of size 724×724, populated with varying number of agents. In each case, the area to be populated was chosen on the same map, and the density of the crowd was kept roughly fixed to isolate the effect of separation dynamics on the relative frame-rates. The resulting number of agents are seldom round figures. Figure 6 shows the frame-rates as a function of the sizes of the crowd, run on an HP Pavilion 1.6 Ghz laptop with 2GB RAM, running XP.

The frame-rates in Figure 6 are *without* the graphic rendering, i.e., these figures show the number of times the position update loop completes per second, for the given number of agents, in the domain of Figure 2. Each crowd-size was used in 10 independent experiments and the averages are reported. The standard errors are low in all cases. Since rendering cuts the frame-rate to roughly half, and 7 frames/sec is an acceptable cutoff for real-time visualization, the Figure 6 only shows frame-rates that are at least 14 frames/sec. This means the said machine can handle crowd-sizes upto roughly 40,000 agents, producing real-time movements. With more sophisticated hardware, it is possible to support much larger crowd-sizes in our framework.

7 Related work

In order to face the complex challenges that crowd simulation poses, primarily three approaches[4] have been traditionally used. One approach assumes that the individuals are passive entities that drift in the presence of forces – the so-called "social forces" model [5] and the associated variants of gaskinetic model proposed by Helbing. This model has been extended to include further individual-level details such as familial ties and altruism [2]. The idea of social forces is similar to our use of various layers imposing "forces" and the compund forces guiding each agent. But our approach is distinctly different from ODE based methods, is discrete, and relies on the simplicity of the process of composing forces in a distributed fashion.

The other approach is an agent-based model where individuals are modeled as intelligent agents with (limited) perception and decision-making capabilities. Some of the earliest applications of simple agent-based behaviors were seen in Reynolds' flocking model – the "boids" [9]. In this and related works, each agent is endowed with a mix of simple steering behaviors, that produce complex macroscopic (group-level) behaviors as *emergent phenomena*. The basic

[4]We do not discuss user-guided or scripted crowd behaviors since we are interested in autonomous crowd movement

idea of emergent behaviors has been extended to rule-based systems [15, 8] that offer the added advantages of efficiency and variety in behaviors. Although our approach is also agent based, we prefer a thin client where the intelligence emerges from the interaction of the agents with their environment. We do observe emergent group behavior that are realistic, similar to other agent-based approaches, but our approach is scalable with the number of agents, in contrast to most other agent-based systems.

With respect to integrating agents within a distributed simulation, some work has been done on Multi-Agent Systems (MAS) using the High Level Architecture (HLA) [16]. As for layered approaches, a layering of the social interaction on environmental simulation was proposed to model interaction between humans and natural environments [13]. The problem of congestion has also been addressed with respect to amusement parks by using "social coordination" to reduce the time wasted in congestion [7]. The issue of time management in MAS was discussed to alleviate some the problems that exist in the simulation by providing "Semantic Duration Models" to help developers [6].

Cellular automata [3, 1] underlie the third major approach, with recent improvements [4] for pedestrian room evacuation, similar to our evaluation domain. In contrast to this approach, our method does not require the user to identify "exits", and involves a much simpler agent model, leading to computational efficiency.

8 Summary

We have presented an extension of the layered intelligence technique that is popular in the game industry, for scalable crowd simulation. We have shown how several navigation behaviors can be implemented efficiently in this framework. The chief advantage of this framework is extendability, where new behaviors can be added by adding separate layers, without affecting the existing layers. We have empirically shown the frame-rates to be sufficient to handle large crowds in real-time. We have identified several aspects where this simulation system can be improved. Besides, in the future, we will also extend this framework to automatically select the maximal set of behaviors that can be handled at a minimal frame-rate, based on an assessment of the available computational resources.

9 Acknowledgement

We would like to thank the anonymous reviewers for helpful comments. This work was supported in part by startup grants from the University of Southern Mississippi.

References

[1] S. Bandini, M. L. Federici, S. Manzoni, and G. Vizzari. *Parallel Computing Technologies*, chapter Pedestrian and Crowd Dynamics Simulation: Testing SCA on Paradigmatic Cases of Emerging Coordination in Negative Interaction Conditions, pages 360–369. Springer, 2007.

[2] A. Braun, S. R. Musse, L. P. L. de Oliveira, and B. E. J. Bodmann. Modeling individual behaviors in crowd simulation. In *Proceedings of the 16th International Conference on Computer Animation and Social Agents (CASA*, pages 143–148, Los Alamitos, CA, USA, 2003. IEEE Computer Society.

[3] M. Fukui and Y. Ishibashi. self-organized phase transitions in CA-models for pedestrians. *J. Phys. Soc. Japan*, pages 2861–2863, 1999.

[4] B. Gudowski and J. Was. Some criteria of making decisions in pedestrian evacuation algorithms. In *Proc. 6th International Conference on Computer Information Systems and Industrial Management Applications (CISIM'07)*. IEEE, 2007.

[5] D. Helbing and P. Molnar. Social force model for pedestrian dynamics. *Physical Review E*, 51:42–82, 1995.

[6] A. Helleboogh, T. Holvoet, D. Weyns, and Y. Berbers. Extending time management support for multi-agent systems. In P. Davidsson and et. al., editors, *MABS 2004, LNAI 3415*, pages 37–48, 2005.

[7] K. Miyashita. Asap: Agent-based simulator for amusement park - toward eluding social congestions through ubiquitous scheduling. In P. Davidsson and et. al., editors, *MABS 2004, LNAI 3415*, pages 195–209, 2005.

[8] X. Pan, C. Han, K. Dauber, and K. Law. A multi-agent based framework for simulating human and social behaviors during emergency evacuations. In *Social Intelligence Design*, Stanford University, March 2005.

[9] C. Reynolds. Flocks, herds and schools: A distrtibuted behavior model. In *Proceedings of ACM SIGGRAPH*, 1987.

[10] L. E. Sucar. *Advances in Probabilistic Graphical Models*, volume 214, chapter Parallel Markov Decision Processes, pages 295–309. Springer, 2007.

[11] M. Sung, M. Gleicher, and S. Chenney. Scalable behaviors for crowd simulation. *Comput. Graph. Forum*, 23(3):519–528, 2004.

[12] R. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[13] D. Torii, T. Ishida, S. Bonneaud, and A. Drogoul. Layering social interaction scenarios on environmental simulations. In P. Davidsson and et. al., editors, *MABS 2004, LNAI 3415*, pages 78–88, 2005.

[14] P. Tozour. *AI Game Programming Wisdom*, volume 2, chapter Using Spatial Database for Runtime Spatial Analysis, pages 381–390. Charles River Media, 2004.

[15] B. Ulicny and D. Thalmann. Towards interactive real-time crowd behavior simulation. *Computer Graphics Forum*, 21(4), 2002.

[16] F. Wang, S. J. Turner, and L. Wang. Agent communication in distributed simulations. In P. Davidsson and et. al., editors, *MABS 2004, LNAI 3415*, pages 11–24, 2005.