Collision Avoidance Priority for Traffic Congestion in Multi-Agent Navigation

Mathias Della Giustina*, Ubirajara Franco Moreno†, Henrique Simas‡

*Universidade Federal de Santa Catarina
Departamento de Engenharia Mecânica
Florianópolis, Santa Catarina, Brasil

†Universidade Federal de Santa Catarina
Departamento de Automação e Sistemas
Florianópolis, Santa Catarina, Brasil

‡Universidade Federal de Santa Catarina
Departamento de Engenharia Mecânica
Florianópolis, Santa Catarina, Brasil

Emails: mathiasdella@hotmail.com, ubirajara.f.moreno@ufsc.br, henrique.simas@ufsc.br

Abstract— In this paper, we present an approach to manage crowds of autonomous agents when navigating around narrow areas of a scenario in a way that ensures every agent is able to reach its destination. In our formulation, agents that find themselves stuck in the crowd receive a priority number which signals nearby agents to steer away from them. This new technique will be tested on a simulated environment and compared to other algorithms that do not contain such crowd priority management.

Keywords— Navigation, Agent, Crowd, Pathfinding, Collision Avoidance

1 Introduction

Navigation algorithms for autonomous agents are codes capable of guiding agents step by step from their current position towards their goal position. To achieve this result, these algorithms must plan a path for the agent to follow, and as the agent follows the path, the algorithm must tell the agent how exactly it should move each step and how to avoid collisions with nearby obstacles such as nearby agents or nearby walls and buildings.

In order to plan paths for the agents, the algorithm must first have a geometrical representation of the environment in which the agents are navigating, so the algorithm needs some sort of a map containing the environment obstacles. These maps are traditionally represented with grids (Yap, 2002), but more recently, they have been represented with polygons (Leonard, 2014), (Pratt, 2014) and triangles (Chew, 1987).

Once there is a map representation of the scenario available, it is possible to plan paths for the agents using pathfinding algorithms such as (Kallmann, 2010) or (Demyen and Buro, 2006) in case of triangle representations or (Dechter and Pearl, 1985) for grid representations.

The final core part of a navigation algorithm is to guide the agent through its planned path while avoiding collisions with nearby obstacles. As these obstacles may be moving and their movement may be unknown or unpredictable, avoiding collisions with nearby moving obstacles is not as easy as it seems. There are many important researches on the subject such as Reynolds steering behaviors (Reynolds, 1999), ClearPath (Guy et al., 2009) and Relative Velocity Obstacles (Van Den Berg et al., 2011).

Additionally, navigation algorithms can have extra functions that are able to modify agents behaviors. There are researches on steering the agents with least effort approaches (Guy, Chhugani, Curtis, Dubey, Lin and Manocha, 2010), researches on giving agents group behaviors so that they stay grouped during collision avoidance (He et al., 2016) and attempts to copy humans movement behaviors (Guy, Lin and Manocha, 2010). More specifically in managing crowds, there are some recent researches that attempt to enforce crowd rules (Foudil and Nouredine, 2006) to solve some situations, as well as some work on long range movement predictions (Golas et al., 2014) that allows agents to predict and start avoiding from collisions a long time before they are a threat.
Unfortunately, there are still some cases where these algorithms are not capable of steering the agents towards their destination, and the agents will remain stuck in the crowd unable to move for a very long time, if not forever.

One of such cases is the case where there is a scenario with narrow pathways and multiple agents attempting to traverse through one of these narrow corridors at the same time. These agents will often find themselves stuck in the crowd and unable to reach their destination.

The collision avoidance priority algorithm presented in this paper detects agents that get stuck and assigns priority values to them as needed. Nearby agents will then steer away from agents that have priority and the agent with priority will find a path free of collisions towards his goal.

This paper contains a section "Problem Overview", with details about the problem with crowds and narrow pathways, a section "Collision Avoidance Priority", explaining the proposed technique to solve the problem, a section "Simulations and Results", which presents the simulations and the data obtained from them, and finally, a section "Conclusions", containing the conclusions obtained from the simulations.

2 Problem Formulation

The problem discussed in this paper is of a two-dimensional top-down scenario $S$ containing a set of static obstacles denoted $O$ and a set $A$ of multiple autonomous agents $a_i$ that navigate through the environment. Each agent $a_i$ is given a set of $n_p$ points of interest $p_i$ where they must successfully move to. In other words, each agent $a_i \in A$ starts at position $p_{i,0}$ and has the goal to move to position $p_{i,1}$, then $p_{i,2}$ then $p_{i,3}$ up to $p_{i,n_p}$, where $i$ is the index of the agent and $n_p$ the total number of points of interest the agent has.

Agents are described as circumferences with a certain collision radius $r$, a maximum movement speed $v$ and a maximum rotation speed $w$. Agents are only allowed to move forward.

The scenario is composed of multiple obstacles of arbitrary shapes and sizes, and the navigation map represents the scenario using triangles generated through a Constrained Delaunay Triangulation (Kallmann, 2014).

The problem is solved by planning a path for each agent individually and then steering the agent towards each path point while avoiding collision with other agents and obstacles. For a given problem of $n_A$ agents and $n_p$ points of interest per agent, each agent $a_i \in A$ plan and follow a path from its starting position $p_{0,0}$ to its first point of interest $p_{i,1}$, and then as it reaches $p_{i,1}$ it plans and follows another path from its current position $p_{i,1}$ to its next point of interest $p_{i,2}$, and then $p_{i,3}$ and so on until it reaches $p_{i,n_p}$. It is desired that all agents $a_i \in A$ reach all of their $n_p$ points of interest.

For this research, the scenario was represented using a Local Clearance Triangulation (Kallmann, 2014) and the path planning done according to Kallmann’s Shortest Paths algorithm (Kallmann, 2010). Agents follow their planned path and avoid local collisions with other agents and obstacles using the RVO2 algorithm (Van Den Berg et al., 2011).

Solving this problem exclusively with path planning and collision avoidance results in failures where some agents are not able to reach their points of interest due to crowded conditions at narrow pathways and an additional strategy is required to ensure agents are able to successfully achieve their goals.

3 Collision Avoidance Priority

Collision avoidance algorithms by themselves such as RVO (Van Den Berg et al., 2011) or Steering Behaviors (Reynolds, 1999) are not always able to properly steer the agents towards their destination, as these algorithms are primarily meant to prevent collisions. Cases where collision avoidance may fail to generate progress towards the agents destination can be easily seen when the only collision avoidance velocity the agent can calculate is zero, because the agent is completely surrounded by other agents.

In order to solve the crowd problem around narrow pathways and allow agents to successfully reach their destination, we introduce the collision avoidance priority algorithm.

Figure 1: Example scenario. Clear-space triangles are drawn in white, Obstacle-space triangles in black and agents are represented in red.
The collision avoidance priority algorithm consists of three steps:

- Assigning priority to agents
- Propagating priority information to neighbors
- Respecting other agents priority during collision avoidance

### 3.1 Assigning Priority to Agents

Let agent $a_i$ be an active agent on the simulation that has a maximum movement speed $v_{i,max}$ and current movement speed $v_{i,cur}$. The agent current movement speed varies according to the situation in which it is: it will be higher for situations where the agent is not in danger of collisions and lower when in crowded and dangerous situations regarding collisions.

If the agent ever becomes unable to move while avoiding collisions due to crowd around it, its $v_{i,cur}$ will become zero (or close enough to zero, as RVO algorithm tries to slow down the agent to prevent collisions instead of instantly stopping it). If the current velocity $v_{i,cur}$ of an agent stays zero for a certain period of time, then the agent is said to be stuck. Likewise, if an agents $v_{i,cur}$ is not zero for a certain period of time, then the agent is not stuck. The time it takes for an agent to consider itself stuck is denoted $t_{\text{stuck}}$, and the time it takes for it to consider itself not stuck is $t_{\text{unstuck}}$.

Once an agent becomes stuck, it receives a priority integer number denoted $s_{i,\text{self}}$ from the algorithm. The variable $s_{i,\text{self}}$ is the navigation priority of agent $a_i$, where agents with lower $s_{i,\text{self}}$ have movement priority over agents with higher $s_{i,\text{self}}$.

The algorithm has a global variable $s_{\text{counter}}$ that starts at 0. Every time an agent $a_i$ requests priority, this global variable is incremented by one and the resulting number is given to the agent as a value for his $s_{i,\text{self}}$. This way agents will always have unique values for $s_{i,\text{self}}$ and the priorities will be ordered by the order agents request them.

Additionally to requesting priority when stuck, if an agent remains stuck while holding priority, then it will give up its priority number $s_{i,\text{self}}$ and request a new priority number from the global counter, meaning this agents $s_{i,\text{self}}$ will become equal to $s_{\text{counter}}$.

### 3.2 Propagating Priority Information to Neighbors

In order for the priority algorithm to work properly, it is needed that all agents that are grouped up obey to the same priority number, that is, only one agent in a given group will have priority while all other agents of this group will have another variable that will arrange them so that the agent with priority in the group can move. This variable that arranges the group is denoted $s_{i,\text{value}}$, which is a real number. As agents are grouped up they also receive a variable $s_{i,\text{group}}$ which holds information of what $s_{i,\text{value}}$ value the group is currently obeying. Agents are considered to be in a group when they are within distance $g$ of any agent of that group.

At the beginning of every simulation step (every frame), each agent $a_i$ active on the simulation must have its priority variable $s_{i,\text{value}}$ set to infinity. Then, agents that have $s_{i,\text{self}} > 0$ will propagate their priority information to all agents that are grouped up together with them. After all propagation is complete, groups of agents will share a variable $s_{i,\text{group}}$ which tells what the lowest $s_{i,\text{self}}$ in the group is, and each agent of the group will have $s_{i,\text{value}} = s_{j,\text{value}} + \text{Distance}(a_i, a_j)$, where $a_j$ is the agent that gave the priority information to $a_i$ and $\text{Distance}(a_i, a_j)$ is the euclidean distance of agents $a_i$ and $a_j$.

In a group, the agent with $s_{i,\text{self}} = s_{i,\text{group}}$ is the agent in the group holding the current priority and it will have $s_{i,\text{value}} = 0$. All other agents in the group will have $s_{i,\text{self}} \neq s_{i,\text{group}}$, $s_{i,\text{value}} > 0$ and $s_{i,\text{value}}$ will be a larger number the further the agent is from the agent holding priority on the group.

Following, Algorithm 1 is the pseudo-code for resetting the agents priority variables before every simulation step and Algorithm 2 is the pseudo-code for propagating the priority of agents to agents that are grouped up.
Algorithm 1 Reset Priority ()
\[ a_i \leftarrow q.Pop() \]
for all \( a_i \in A \) do
\[ s_{\text{value}}(a_i) \leftarrow \infty \]
\[ s_{\text{group}}(a_i) \leftarrow \infty \]
end for

Algorithm 2 Propagate Priority (a)
\[ a \leftarrow \text{Agent propagating its priority} \]
\[ q \leftarrow \text{Queue of agents} \]
\[ g \leftarrow \text{Group distance threshold} \]
q.Push(a)
while !q.Empty do
\[ a_j \leftarrow q.Pop() \]
for all \( a_i \in A \) do
\[ d \leftarrow \text{Distance} (a_i, a_j) \]
if \( d < g \) and \( s_{\text{self}}(a_j) \leq s_{\text{group}}(a_i) \) and \( s_{\text{value}}(a_j) + d < s_{\text{value}}(a_i) \) then
\[ s_{\text{value}}(a_i) \leftarrow s_{\text{value}}(a_i) + d \]
\[ s_{\text{group}}(a_i) \leftarrow s_{\text{group}}(a_i) \]
q.Push(a_i)
end if
end for
end while

Algorithm 3 Compute New Velocity ()
\[ g \leftarrow \text{Group distance threshold} \]
\[ \alpha \leftarrow \text{Priority radius multiplier} > 1 \]
for all neighbors \( a_j \) of \( a_i \) do
\[ r_{\text{comb}} \leftarrow \text{Radius} (a_j) + \text{Radius} (a_i) \]
if \( s_{\text{group}}(a_j) \neq \infty \) and \( s_{\text{group}}(a_i) = s_{\text{group}}(a_j) \) and \( s_{\text{value}}(a_i) > s_{\text{value}}(a_j) \) and
Distance \((a_i, a_j) < g\) then
\[ r_{\text{comb}} \leftarrow r_{\text{comb}} \times \alpha \]
end if
\[ \ldots \text{continue with ORCA calculations for agent} \]
\[ a_j \]
end for

3.3 Respecting Other Agents Priority During Collision Avoidance

Once all agents have their \( s_{\text{value}} \) and \( s_{\text{group}} \) variables set for the current simulation step, it is possible to proceed and compute agents velocities for the simulation step.

The simulation step process is the same proposed by the standard RVO2 algorithm, with the addition that agents will consider neighbors with priority to have a slightly larger radius than they actually have. An agent \( a_i \) is considered to have priority over agent \( a_j \) if both agents are in the same group and \( s_{\text{value}} < s_{\text{value}} \).

Algorithm 3 shows the pseudo-code for this radius increasing operation. The algorithm is simply an adjustment to how RVO2 calculates the combined radius for a set of two agents \( a_i \) and \( a_j \). If \( a_j \) has priority over \( a_i \), then \( a_i \) will calculate a combined radius for these two agents multiplied by a factor \( \alpha > 1 \), which will result in a larger radius and \( a_i \) will steer away from \( a_j \).

4 Simulations and Results

It is desired to test the developed priority algorithm and compare it to the basic algorithm performance.

Because the root problem presented here was some agents not being able to reach their destination in certain cases, we surely want to evaluate how often this happens. Additionally, it is ideal that the agents reach their goal as fast as they can, so it also interesting to evaluate how much longer it takes on average for agents to reach their destination when affected by crowded scenarios.

To measure how often agents are unable to reach their point of interest (their destination), we’re going to use the number of times this failure happened \( n_{\text{fail}} \), divided by the total number of points of interests \( n_{\text{total}} \) and we’re naming it failure rate \( f = n_{\text{fail}}/n_{\text{total}} \).

Each agent \( a_i \) receives a number \( n_p \) of points of interest \( p_i \) during the simulation start, and they have to plan a path from \( p_{i0} \) to \( p_{i1} \), then \( p_{i2} \) up to \( p_{i,n_p} \), which means they have to move from point of interest to point of interest until they reach the final point (or fail to). For each pair of points of interest \( p_{i,j} \) and \( p_{i,j+1} \), a reference path \( \Psi_{i,j,\text{ref}} \) is planned. The reference time \( t_{i,j,\text{ref}} \) it takes for the agent \( a_i \) to follow this reference path \( \Psi_{i,j,\text{ref}} \) can be estimated by the length of the path \( \text{Length}(\Psi_{i,j,\text{ref}}) \) divided by the movement speed \( v_i \) of agent \( a_i \).

With the reference times defined, evaluating how longer it takes for the agent to follow a path due to crowd becomes as simple as dividing how long it actually takes for the agent \( a_i \) to follow the path \( \Psi_{i,j} \) during the crowded simulation by the estimated reference time for that same pair of points of interest \( (t_{i,j})/t_{i,j,\text{ref}} \). The resultant of this division is the time ratio \( t_{i,j} = t_{i,j}/t_{i,j,\text{ref}} \), and the average of the time ratio for all pair of points is \( tr = \frac{1}{n_{\text{total}}} \sum_{i=0}^{n_A} \sum_{j=0}^{n_p} t_{i,j}, \) where \( n_{\text{total}} = n_A \times n_p \).

4.1 Simulation Setup

To evaluate the desired variables for the simulation using the priority algorithm and also for the same simulation without using it, a arbitrary balanced map with multiple obstacles and multiple pathways to pairs of points was used. The map is 64x64 meters long with 2215 triangles.

Each simulation is composed of \( n_A \) agents and \( n_p \) points of interest per agent, resulting in a total of \( n_{\text{total}} = n_A \times n_p \) points of interest on each simulation. For each algorithms, we run the simulation on 3 different setups: 50, 100 and 200.
agents, and for each setup, 10 different simulations. Equal simulations for the two algorithms receive the same random seed, to ensure they have the same simulation input parameters.

For each simulation, each agent $a_i \in A$ is assigned with a random collision radius $r_i$ between 0.30$m$ and 0.50$m$, a random movement speed $v_i$ between 2.0$m$/s and 3.0$m$/s as well as a random starting point $p_{i,0}$ and a random set of $n_p = 5$ points of interest. During the simulation, if an agent takes longer than 60 seconds to reach its current point of interest $p_{i,j}$, then it is considered that the agent failed to reach its current point of interest. Once an agent fully completes its task of reaching $n_p = 5$ points of interest, it remains idle until the end of the simulation. Additionally, points randomized outside the main white area of the map were discarded and re-rolled until a valid point was found.

Algorithm 4 Agent Behavior

\begin{algorithm}
\begin{algorithmic}
\State $a_i \leftarrow \text{agent}$
\State $\text{Position}(a_i) \leftarrow \text{Current position of agent } a_i$
\State $\text{Velocity}(a_i) \leftarrow \text{Velocity of agent } a_i$
\State $dt \leftarrow \text{Simulation timestep}$
\For{$j \leftarrow 1 \text{ to } n_p$}
\State $\Psi_{i,j} \leftarrow \text{Plan path from } p_{i,j-1}\text{ to } p_{i,j}$
\While{$\text{Position}(a_i) \neq p_{i,j}$}
\State $\text{Velocity}(a_i) \leftarrow \text{ComputeNewVelocity}(a_i)$
\State $\text{Position}(a_i) \leftarrow \text{Position}(a_i) + \text{Velocity}(a_i) \times dt$
\EndWhile
\State Wait for next simulation step
\EndFor
\end{algorithmic}
\end{algorithm}

Algorithm 4 shows the behavior of agents during the simulation. For each point of interest $p_{i,j}$ of agent $a_i$, a path $\Psi_{i,j}$ is calculated, and the agents velocity is calculated and its position is updated every simulation step until it reaches the point of interest $p_{i,j}$. Then, it proceeds to plan the next path towards its next point of interest.

Because collision avoidance often deviates the agents from their originally planned path, while the agents are calculating their new velocity and updating their positions, they also periodically re-plan their path to their current point of interest.

4.2 Simulation Results

Simulations were done as described previously, and the discussed variables of interest can be seen in table 1. The results point that as the number of agents active on the simulation increase, the number of failures to reach points of interest also increase. This means that these failures are more likely to happen on larger crowds. While most failures happened because of the crowd on narrow pathways, some failures happened for other reasons related to crowds, such as agents completing their tasks and remaining idle at an unfortunate position for the remainder of the simulation.

<table>
<thead>
<tr>
<th>$n_A$</th>
<th>$n_{total}$</th>
<th>$n_{fail}$</th>
<th>$f$</th>
<th>$tr$</th>
</tr>
</thead>
<tbody>
<tr>
<td>RVO</td>
<td>50</td>
<td>2500</td>
<td>146</td>
<td>0.0584</td>
</tr>
<tr>
<td>RVO</td>
<td>100</td>
<td>5000</td>
<td>1243</td>
<td>0.2486</td>
</tr>
<tr>
<td>RVO</td>
<td>200</td>
<td>10000</td>
<td>5067</td>
<td>0.5067</td>
</tr>
<tr>
<td>PRVO</td>
<td>50</td>
<td>2500</td>
<td>1</td>
<td>0.0004</td>
</tr>
<tr>
<td>PRVO</td>
<td>100</td>
<td>5000</td>
<td>1</td>
<td>0.0002</td>
</tr>
<tr>
<td>PRVO</td>
<td>200</td>
<td>10000</td>
<td>125</td>
<td>0.0125</td>
</tr>
</tbody>
</table>

Table 1: Table with simulation results. Lines labeled RVO are without the priority algorithm and lines labeled PRVO are the results using the priority algorithm. $n_A$ is the number of agents, $n_{total}$ the total number of points of interest, $n_{fail}$ the number of failures to reach the point of interest, $f$ the failure ratio and $tr$ the time ratio.

As seen in table 1, the RVO simulation that does not benefit from the priority algorithm has a total of 146 (5.84\%) failures for a simulation with 50 agents, whereas the same simulation for PRVO, which uses the priority algorithm, only 1 (0.04\%) of the points of interest was not successfully reached. Additionally, the simulations with the priority algorithm yield a time ratio closer to 1.0, which is the ideal result. The failure rate greatly increases with the increase in number of agents for the RVO algorithm, but it remains stable for 50 and 100 agents for the PRVO case, increasing only when simulating with 200 agents. On top of less failures, the PRVO algorithm also yields lower time ratios than the same simulations using the RVO, which means the agents are able to reach their points of interest faster on average.

5 Conclusions

For both real and virtual applications of navigation algorithm for autonomous agents, it is highly desired that the agent is capable of reaching its destination reliably.

Unfortunately, there were cases where agents were not being able to achieve their navigation goals, and existing algorithms were not satisfying enough or even reliable to use on applications that demanded success.

In this paper, we presented a new algorithm that assigns priority to agents that can find their way around crowded situations at narrow pathways. This priority creates a flow of movement that allows agents to move and solve these situations at narrow passages.

The goal of the developed algorithm was to reduce the number of failures that agents have when trying to reach their destination, and improve autonomous navigation algorithms to make them more reliable. As seen in the simulations, the number of failures greatly reduce with the im-
Figure 3: Map used for the simulations. It contains many narrow corridors that fit only two or three agents at a time, as well as many wide corridors that many agents can traverse at the same time without problems.

The simulations also report an improvement in the average time it takes for agents to reach their destination. This happens because even in situations where agents could eventually reach their destination before, they are now able to reach it with ease.

References


