Utrecht University

Master's thesis

High Level Planning in Crowd Simulation

ICA-5811678

Wouter van der Waal

Supervised by
Dr. Roland J. Geraerts
Co-supervised by
Dr. Frank Dignum

October 8, 2020

Abstract

One of the main challenges in Crowd Simulation is that a problem does not only need to be solved, but it often needs to be solved for the large number of agents that are simulated. In this project we want to let all agents determine what steps they need to perform to reach their personal goal. While Artificial Intelligence has many methods to determine such steps, most of them are not suited to do this for the large number of agents in Crowd Simulation.
In this thesis we show that it is possible to use an Hierarchical Task Network to solve high level planning problems in Crowd Simulation. Compared to the lower levels of planning, the relative time required to plan this for a Train Station scenario is very small ($< 0.1\%$). In addition, we show that by using the proposed Agent Profiles approach, we can solve extremely complex planning problems for large numbers of agents (up to one million) in a time that is short enough for a real-time simulation.

Contents

1 Introduction

The earliest large scale crowd simulation has been used in movies. Batman Returns and Disney's the Lion king used boid simulation techniques to simulate swarms of bats and a stampede of wildebeests respectively. Later animated movies such as AntZ or Bugs Life used a combination of physics based techniques, flocking behaviors and finite state machines to simulate the behavior of over 60.000 background characters. (Musse & Thalmann, 2001) This pregenerated type of crowd simulation can be easier, because it does not have to perform in real-time. The main goal for these applications is to make crowds look good, but not necessarily realistic.

If crowd simulation software actually can simulate realistic crowds, businesses, such as airports, train stations or shopping malls, can use it to design their buildings to optimize passenger or customer flow. Events that attract large crowds, like music festivals, can use it to minimize crowd density, improve the comfort of visitors and minimize risk of crowd disasters. Commercial applications, such as Pathfinder (ThunderheadEngineering, 2020), Pedestrian Dynamics (InControl, 2020) and MassMotion (Oasys, 2020) try to provide exactly these tools.

In order to create realistic scenarios, agents have to behave realistically both on low-level path planning and on high-level task planning. This thesis will focus on the latter. High level decision making has been studied in the artificial intelligence field for a long time, although applying this to crowd simulation can be challenging.

In general, the existing methods have been created to create optimal plans for single robots or agents. The main difference between these situations is that in crowd simulation we want to be able to plan and follow paths with thousands of agents simultaneously, so both memory and time efficiency become very important.

Most scientific approaches for complete crowd simulation packages solve this efficiency problem by using a type of Finite State Machine to describe agent behavior. (Musse & Thalmann, 2001; Sung et al., 2004; Shao & Terzopoulos, 2007; Paris & Donikian, 2009; Curtis et al., 2016) While this approach is definitely usable to let large numbers of agents select their next goal, it does not allow them to actually plan their behavior. That is, they effectively only follow a path predefined by the designer. While this does allow for complex behavior, it can difficult and time-intensive for the user to create such complex plans for all agents.

A different approach for goal planning, outside of crowd simulation is to split the complex goal into less complex goals (Fikes & Nilsson, 1971; Sacerdoti, 1975). These sub-goals are split, until only simple primitive actions remain that can be executed by an agent or robot. This way, complex solutions to reach a goal can be determined by the program instead of the user. However, since a lot more has to be calculated, this approach can be challenging to use for large crowds in real-time.

1.1 Goals and Contributions

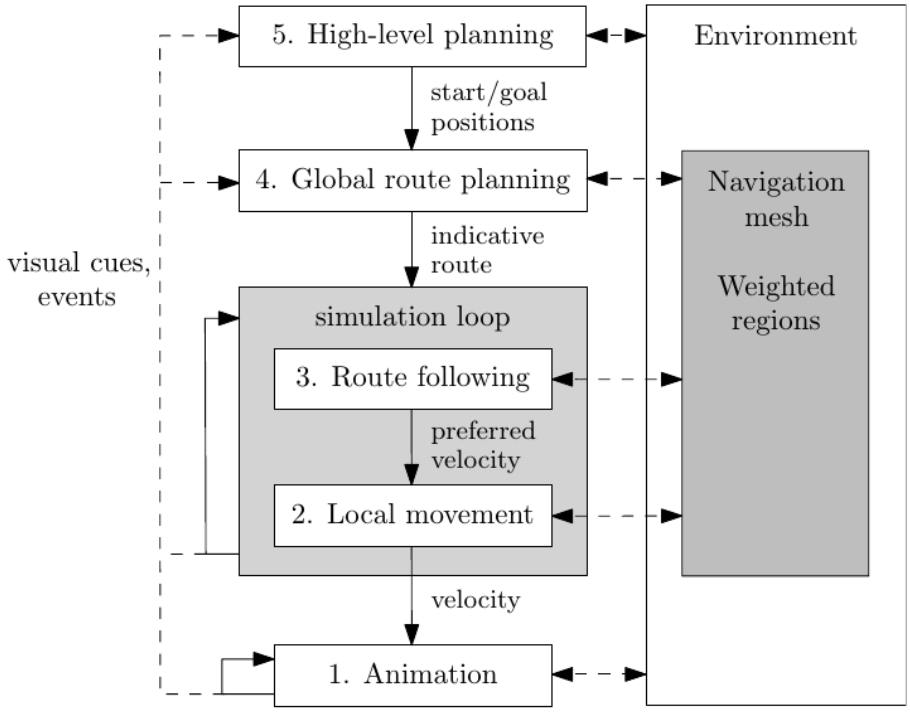


Figure 1: The simulation structure of the UUCS engine. (van Toll et al., 2015)

The goal of this thesis is to propose a goal planning architecture that is efficient enough to be used in a crowd simulation environment. The architecture is designed to work with the "Utrecht University Crowd Simulation" simulation engine (UUCS) using a separately developed authoring tool "SimCrowds". In this approach for a complete crowd simulation tool, the simulation is split into different steps, as seen in Figure 1. The goal planning architecture will focus only on the top level "5. High Level Planning". As such, it will provide a goal for each individual agent. Since the goal planning will be somewhat separated from the rest of the engine, the proposed architecture can be used by any other crowd simulation tool or project.
The authoring tool allows users to flexibly and interactively design scenarios. At its current state, agents move towards a single goal inside a defined goal-area. When they reach that area, they either receive a new goal or agents are removed from the simulation. Any proposed goal planning architecture will need to be able to work in this interactive environment. If the scenario changes, optional goals might be added or removed and agents need to be able to adapt to this.

Our research question is: What high-level goal planning architecture allows for complex and scalable agent behavior in interactive, real-time, large-scale crowd simulation?

There are multiple notable parts in the research question. First of all, we are interested in goal *planning* not goal following. The program should determine the plan based on (simple) user input. This way, users do not need to understand the more complex parts of the plans, which would be the case for goal following approaches, such as Finite State Machines.
Secondly, we are interested in *complex* behavior. In this context, we want agents to solve problems that may require multiple steps. For example, traveling to a train requires the agents to buy a ticket before entering the station and to enter the station using a ticket gate.
Additionally, we want the resulting behavior to be *scalable*. This means that we want the computations required for an agent not to be highly complex. If the approach is simple enough in terms of calculations, it can still be used if the number of agents increases a lot.
Furthermore, we want to perform in *real-time*. This means that we cannot pause or slow the simulation at any point to determine the next steps for agents.
Finally, we are interested in *large-scale* crowd simulation. This means that we want to be able to simulate large scenarios. Simulating a single building can be considered small scale. We want to be able to simulate multiple areas at the same time. In this case, agents might have vastly differing goals to pursue.

Section 2 will show the related work used in this thesis. Specifically, Section 2.1 will show previous goal following methods in crowd simulation and goal planning methods from artificial intelligence and Section 2.2 will show which actions agents need to be able to perform. Section 3 will explain the existing software used for this project. Section 4 will propose a high-level, goal-planning architecture, analyze its theoretical complexity and give some details about the implementation itself. Section 5 will show the practical performance impact of the architecture. Finally, Section 6 will conclude this thesis and talk about future work.

2 Related Work

2.1 High Level Planning Methods

Most previous work considering high level planning in crowd simulation makes use of Finite State Machines. A Finite State Machine describes all possible states an agent can be in and the corresponding behavior. (Tozička et al., 2014) Given enough states, agents can exhibit complex behavior. This method can be very efficient, but all states do have to be defined beforehand, so more complex behavior will require more effort from the user.

A Finite State Machine by itself can be powerful enough for crowd simulation (Curtis et al., 2016), but the method can be adapted to allow for some specific behavior. Combining Finite State Machines with explicit rules, which can target large groups of agents to adapt their behavior, can be used to create complex scenarios, such as evacuations (Ulicny & Thalmann, 2002).
The Finite State Machine itself can be adapted by using probabilities instead of conditions to determine the next state or goal (Sung et al., 2004). This will add some variation to agent plans, so agents will not exhibit exactly the same behavior. This approach does assume that individual agent behavior is irrelevant in crowd simulation.
A somewhat different approach using Finite State Machines can be done by using a Hierarchical Finite State Machine (Paris & Donikian, 2009). This splits difficult tasks into smaller ones, so agents can determine which smaller tasks to complete to reach a complex goal. The addition of priorities to each (sub)goal allows agents to plan for multiple goals at the same time.

An alternative approach is to use a directed graph to determine behavior (Shao & Terzopoulos, 2007). This way, multiple conditions can be linked together easily to determine the next goal for an agent. By using tokens or variables, agents can remember and determine what they have already done and what they want or need to do, aside from their main goal. This allows agents to wander around while waiting until their goal is available (e.g. getting a drink while waiting on the train).

Outside of crowd simulation, artificial intelligence has researched goal planning extensively. In the following section we will consider the main ones. To limit the number of methods, we will ignore the more specified variants of the methods. So, for example, we will look at Neural Networks, but not at Deep Neural Networks.

Neural Networks
A neural network can be used to mimic human behavior if enough data is available. Using this method, agents can express extremely complex behavior. (Jolly et al., 2007) It can be used to model scenarios that are too complex to logically describe. It is, however, completely dependent on data. Agents will only learn whatever behavior is in the data. In general, it requires a lot of training data. Processing this data and training the network can take a lot of time and resources. In addition, it can be difficult to understand why resulting behavior is expressed.

BDI
A BDI-model uses beliefs, desires and intentions to model individual agent behavior. (Dignum et al., 2000) Users will have to describe all beliefs, desires and intentions and the links between them, which can take a lot of time and effort. Resulting behavior can be very complex and still understandable. Since it does model complex individual behavior, it is quite computational expensive.

2APL

A special mention for the BDI-approach is 2APL (A Practical Agent Programming Language) (Dastani, 2008). This approach is designed to simulate multi-agent systems and later versions can work for about 100000 agents at the same time and includes repairing plans, making it usable for an interactive environment. This approach does solve the high computational costs of general BDI-models, but it still requires a lot of user input to work.

STRIPS Planner

A STRIPS planner links behavior using pre- and post-conditions to reach a desired goal state. (Fikes & Nilsson, 1971) As with the Finite State Machines, each "state" has to be defined by the user. However, the links between the states do not have to be defined, but can be automatically solved. This method is computational efficient, allows great flexibility and complex behavior. It is mostly designed for problem solving, but by linking pre- and post-conditions, it can be used for task following.

Hierarchical Task Network

An Hierarchical Task Network (HTN) links tasks an agent can perform in an hierarchical way. (Sacerdoti, 1975; Kelly et al., 2008; Kaelbling & Lozano-Pérez, 2010) A task can be defined as a combination of sub-tasks or it can be a primitive action that can be performed. Similar to the Finite State Machine and the STRIPS planner, each task has to be defined beforehand. With an HTN, a complex task consists of multiple smaller, less complex tasks. Alternatively, given the right pre- and post-conditions, larger tasks can be automatically "solved" into smaller tasks.

Depending on the efficiency requirements, a later task in each HTN can be evaluated while an agent is performing a former task. If efficiency during runtime is important and behavior is not too complex, the HTN can even be built fully beforehand. Efficiency wise, this would effectively result in each agent following a linear task list.

If more complex behavior is preferred, like conditional tasks dependent on location, a shorter horizon evaluation can be applied, evaluating tasks whenever it is required. This can also be used to handle infinite search spaces, possibly resulting in agents performing tasks infinitely.

To summarize, changing the HTN evaluation horizon allows more complex behavior or more efficient behavior, depending on the requirements.

Similar to the STRIPS planner, the HTN is designed for problem solving, but can easily be used for task following.

Task List

The simplest way to follow tasks is using a task list. Multiple tasks are performed sequentially in a predefined order. More complex behavior, like distributions over different tasks can be done by skipping over tasks using a probability. This is similar to a Finite State Machine without conditional statements.

2.1.1 Scope

Typical high level planning required in crowd simulation software generally consists of finding the correct steps necessary to finish a single (complex) goal. A typical goal can be for an agent to evacuate an environment, catch a specific train or visit a (music) festival.
The first scenario only requires the agent to move to the nearest exit, so it does not require complex high level planning. Here, a single goal for each agent would suffice.
However, if an agent wants to catch a train, the plan requires multiple steps. He first needs a ticket, then needs to pass the ticketgate and finally needs to go to the correct platform. This scenario would require an agent to perform multiple steps in the correct order.
Finally, agents visiting a festival would have multiple smaller goals to perform, such as getting a drink or finding a bathroom, while waiting for specific events. In this scenario an agent would have to monitor and fulfill its own needs until a certain moment. Because of unforeseen circumstances, such as a longer than expected line for a ticket booth, the time it takes to fulfill the smaller goals can be unpredictable. Therefore, it will be near impossible to create a complete action plan or task list beforehand. For such complex scenarios goal following will not suffice and goal planning will be required.

2.1.2 Conclusion

Both Neural Networks and the BDI-model can result in very complex behavior, but are generally computationally expensive and require quite a lot of effort from the user. These methods are most useful when using a small number of agents with very complex behavior.
For goal following, Finite State Machines, STRIPS Planners, Hierarchical Task Network and simple Task Lists are all very efficient, all effectively letting agents follow a task list. They also have a similar approach, requiring the user to split the desired behavior into smaller blocks. The FSM and Task List both let users directly link primitive behavior. The STRIPS and Hierarchical Task planners both use symbolic reasoning to link the primitives, although with a proper user interface, users should not notice any difference between direct or symbolic links.
There are significant differences when we consider goal planning. In general, a finite state machine and simple task list do not allow automated planning. Both the Hierarchical Task Network (HTN) and STRIPS planner do. Both HTN and STRIPS can solve the same problems, assuming the HTN cannot create a task list of infinite size or cannot create cyclic plans (Lekavỳ & Návrat, 2007). While task lists of infinite size in itself is not very useful for crowd simulation, cyclic plans can allow agents to roam around while waiting for a specific condition to be fulfilled (e.g. the train arrives on its platform). This can be done splitting the (sub-)task *waiting* into sub-tasks that can be done while waiting, such as getting

a drink or finding a bathroom, and again waiting until a specific moment. This will result in agents following their individual needs until that specific moment. A HTN can therefore be the preferred architecture for crowd simulation.

2.2 Building Blocks

To solve a complex problem, methods like the Hierarchical Task Network, STRIPS, Finite State Machines and even simple task lists try to split the problem into smaller parts that can be more easily solved or are already solved. This process depends the available actions that can be executed by an agent without further high-level planning.
To increase the maximum complexity of solvable scenarios, we require specific building blocks which can be combined into complex tasks. To determine which minimal building blocks are useful, we used passenger flow charts. These describe which steps are executed by people in certain scenarios in Section 2.2.1. We have also looked at tutorials from manuals of existing crowd simulation tools to see what crowd simulation software is currently used for in Section 2.2.2.

2.2.1 Passenger flow

Passenger flow charts describe what actions people take to move through a certain scenario. If we would want to model such a scenario using crowd simulation software, agents would need to be able to perform the same actions. This section describes the passenger flow in a train station and in an airport.

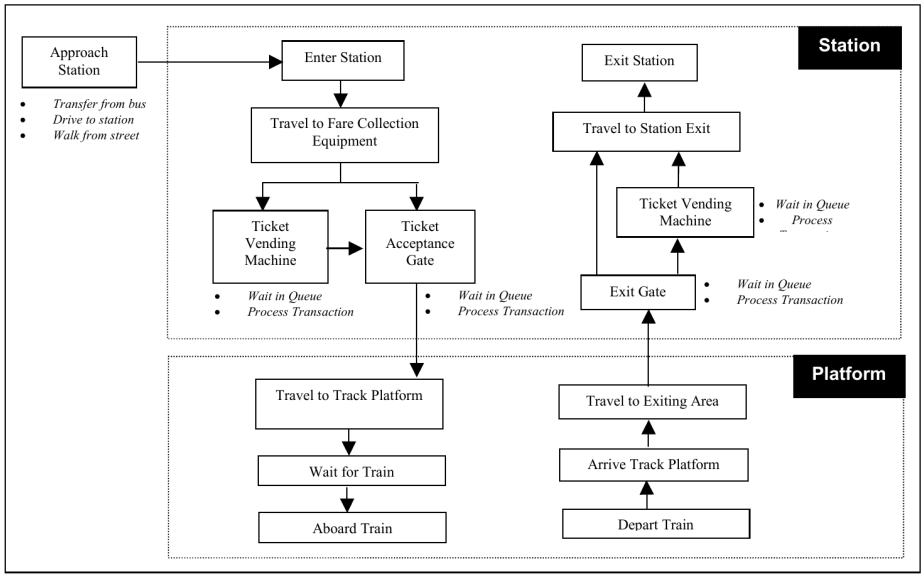


Figure 2: The passenger flow in a train station. (Li, 2000)

Train Station

To simulate the inside of a train station, we require the following steps (Li, 2000), starting from the approach of the station, as seen in Figure 2:

1. Enter the station
2. Set multiple sequential goals as an action plan, this is required throughout the scenario
3. Decide to go to the ticket vending machine or straight to the gate, based on personal availability of a ticket
4. Wait in queue, for both the vending machine and the gate
5. Pass the ticket gates
6. Wait for train
7. Choose between similar areas

To clarify the final point, agents will have to decide between different areas where a step can be solved. In this scenario, there probably are multiple ticket vending machines, ticket acceptance gates and waiting areas that can be used. Departing is mostly the same in reverse and contains the same actions. We can compare the capabilities of the software packs in Table 9 in the appendix.

Airport

According to Guizzi et al. (2009), the simulation of an airport can be split in four parts.

1. *Generating passengers*
   Common and Business class passengers have different behavior.
2. *Flights with common desks and flights with dedicated desks*
   Passengers need to decide between common or business/class check-in, depending on their identity.
3. *Check-in*
   Passengers wait in queue. Passenger processing time depends on the group size how much luggage they have.
4. *Security Control*
   Some people decide to wait passing the security checkpoint for other activities depending on time available, i.e. shopping or working. Passengers wait in queue.

If we translate these steps into building blocks, we require the following building blocks.

- Different behavior for different types of agents (e.g. common class vs. business class)
- Choose check-in desk based on agent identity
- Queuing
- Decide on activities before security, depending on time available (roaming until a specific time)
- Choose between similar areas

This scenario also requires agents to choose between similar areas, for example there are multiple check-in desks and waiting areas.
Capabilities of existing software packs can be found in Table 10 in the appendix.

2.2.2 Tutorials

Existing software packs usually provide practical examples to simulate in their tutorials. In this section we will take a look at these examples to find out what building blocks would be necessary to build them.

GOLAEM
The primary example in the GOLAEM tutorials is the creation of a wave in a stadium. GOLAEM approaches this problem by moving a plane over the stands with the visitors. Every person that is on that plane has a chance to participate in the wave.
If we would mimic the GOLAEM approach, we need a movable area in which agents perform an action. More generally, agents need to perform an action based on location and time, which could be caught in a simple function instead.

Pedestrian Dynamics
The tutorials of Pedestrian Dynamics (2017) contain three examples: evacuation, the ingress and egress of stands in a stadium and behavior in a metro station. Since we have already seen a train station scene in Section 2.2.1, we will not take a further look at the (less complex) metro station in these tutorials.

Evacuation
To turn any scenario into an evacuation scenario, we need a kind of trigger to notify all agents that the evacuation has begun. This can be done using a time trigger or an event trigger. This trigger would have to change the behavior of all affected agents to evacuate. In addition, changing agent speed could also be of use. Finally, the scenario map should be altered. For example, emergency exits need to open and uni-directional gates must become bi-directional.
Capabilities of existing software packs can be found in Table 11 in the appendix.

Stands
Ingress: During the ingress of the stands, agents spawn at the entrance of the stadium and move to their individually assigned seats over a period of ten minutes.
Each agent has a personal goal. If a general task list is used, an agent can select a random unclaimed seat, although that would require a controller to keep track of all seats. Alternatively, seats can be distributed beforehand and each agent will keep track of their own goal.

Egress: Agents spawn in their seats and leave the stadium within one minute. All agents have the same goal: leave the area. The moment when they decide to leave differs. This could be done by using a simple distribution to determine the start of their plan or assign a random waiting time below one minute to every agent.

2.2.3 Conclusion

To model advanced scenarios, we require some building blocks. None of the existing software packs supply all of them.

The first set of building blocks are blocks that are unlikely to be solvable by using high level planning. The blocks cannot be reasonably split into smaller tasks. To include them, they need to be solved during lower level planning, but they should still be supplied as separate blocks in the scenario.
Alternatively, they could be approached by using specialized areas. These areas would control all agents in it by setting very specific temporary goals. This would, however, be a quite inflexible and resource intensive approach.

- Pass ticket gates
- Queuing

This set contains building blocks that require conditional statements during high planning.

- Make decision based on personal situation (buy ticket or enter station)
- Distribution over different tasks
- Make choices based on identity

This set contains blocks that cannot be calculated beforehand. Roaming around until a specific time cannot be split into a finite number of subtasks, without knowing how long each task would take and how much time it would take to move between tasks. Since the latter is dependent on unpredictable traffic we assume pre-calculating would be impossible for crowd simulation.
Without pre-calculating, agents could roam until a specific trigger (e.g. roam

until 2 minutes before train arrives), although this specific time could be dependent on distance to the time-sensitive goal.

- Decide on activities depending on time available (e.g. roaming until train arrives)
- Specific action based on time and location (e.g. a wave in a stadium)

Finally, this set of building blocks cannot be simplified and are therefore necessary primitives for agents. This includes the previous sets that could be simplified.

- Set current goal
- Do nothing until a specific moment (waiting)
- Different task list for different types of agents
- Make decisions based on (individual) conditions
- Choose between similar areas

If a crowd simulation system can supply these building blocks for agent behavior, it can be considered complex (by industry standards). To design our goal planning architecture, we will need to make sure these building blocks are supported.

3 Preliminaries

This project is built on existing software. This section will describe the UUCS engine, the corresponding authoring tool (SimCrowds) and the Trigger, Action, Event system that already controls agents on a higher level.

3.1 UUCS engine

The Utrecht University Crowd Simulation (UUCS) engine (van Toll et al., 2015) simulates all individual agents. This C++ engine handles the low level crowd simulation, such as finding a path from one point to another and evading obstacles and other agents while traveling towards a location. This engine supports real-time editing of the scenario, which allows users to design scenarios more flexibly than other crowd simulation tools. This way, users can immediately see the effects of small changes in the scenario on the simulated crowds.
The authoring tool uses the UUCS Unity plugin. This plugin uses the C++ engine in the C# environment of Unity3D. So agents created in Unity3D will call to the UUCS Unity plugin for path planning tasks. The plugin will then call to the UUCS engine and send the solutions for all agents (agent direction, speed, etc.) back to Unity3D, which will use te information to determine the movement of the agents.
This allows both the efficiency of C++ for the large-scale low-level planning as well as the convenience and development speed of C# and Unity.

3.2 SimCrowds

SimCrowds allows users to easily create and simulate scenarios in real-time. Users are able to change the scenario during the simulation and agents will immediately adapt to those changes. The authoring tool supplies users with building blocks, which can be used to create a scenario. These building blocks include spawning zones in which a variable number of agents spawns, exit zones for agents to leave the simulation, way-points to travel to before going somewhere else, obstacles for agents to avoid, passageways in which agents can only move a single direction.
Different zones can be linked with each other. By linking a spawn zone to a way-point zone, agents spawned in the first zone have the second zone as their goal. When agents reach the way-point, that zone can give them a new goal. This way agents only ever have a single goal to reach. If a single spawn zone has multiple way-points or exit zones linked, the user can specify in what distribution agents choose each goal using a percentage or cost-based distribution. Agents are designated a flow-group. Each flow group can determine links and distributions separately, allowing for different behavior for different types of agents.
This approach has taken user interaction into account at its core. The design is based on how (less tech-savvy) users can use the tool to easily create scenarios. This approach will be continued in this thesis. The user should determine what is happening (an agent's final goal), then the program should determine how this is happening (the task plan).

3.3 Trigger, Action, Event System

In addition to the building blocks, there is also a "trigger, action, event" system being developed for SimCrowds. This logic system can be used to change both the scenario and agents. For example, during a fire alarm, all spawn zones can be closed and all agents can be triggered to evacuate to the exits. In addition, agents can be triggered to perform certain actions when entering a specific zone. For example, an agent can watch a street performer for a while and then return to its original goal.
While this system does allow reactive behavior in agents, it does not allow agents to actually plan ahead. The triggers in this system are usable for high level planning. If an agent needs to wait until the train arrives (as described in Section 2.2), the trigger system can be used to notify agents whenever they need to stop waiting/roaming and go to their platform. The logic system can also be used for scenarios with less complex high level planning where long term planning is not necessary, such as the aforementioned evacuation scenario.

4 Goal Planning

4.1 Architecture

4.1.1 Hierarchical Task Network Structure

As mentioned briefly in Section 2.1, a Hierarchical Task Network (HTN) takes a complex goal and splits this into smaller less complex compound tasks. This is done repeatedly, until the tasks are split into primitive actions an agent can execute. For this approach to work, the program requires two lists as the domain description: a list containing all compound tasks and a list containing all primitive actions an agent can take. Additionally, it requires a State description for each agent that keeps track of its current situation. Finally, it requires a Task Network, which keeps track of which tasks and primitives are planned already.

Definition 4.1. (State) Each agent requires a separate State S. A State describes the (current) situation using propositions. Each proposition p_i is either true or false and is used to remember multiple facts, such as "has this agent bought a ticket" or "has this agent entered the station".

$S = p_1 \wedge p_2 \wedge ... \wedge p_n$

Definition 4.2. (Task Network) A Task Network (TN) is a list of all planned tasks and primitives. Initially this network only contains the goal task(s), but as tasks are split into other tasks and primitives, the Task Network increases in size.

$TN = (h_1, h_2, ..., h_m)$, where h_i is the name of task i.

Definition 4.3. (Task) A (compound) task describes how it can be split into (different) tasks and primitives. Because there is not necessarily a constraint on subtasks referring to an earlier used task, loops can occur using tasks. For example (without loops), entering a train station could be done by first buying a ticket and secondly passing through the ticket gate. Formally, this means each Task requires the following definitions:

h: the name of the task

Pre: the preconditions of the tasks, which must all be valid in the agent's state before the task can be used

Sub: all (sub)task names, into which the current task can be split

Definition 4.4. (Primitive) A primitive action describes how and when a simple action can be executed. It is similar to a Task, but instead of a sequence of subtasks it describes the effects of the action.
For example, buying a ticket could be a primitive. This action would require an agent to have enough money. Performing this primitive action would result in the agent losing money and gaining a ticket.

h: the name of the action

Pre: the preconditions of the action, which must all be valid in the agent's state before the primitive can be executed

Eff: the effect(s) on the agent state when executing this action

Each planning problem requires its unique State and Task List. If agents have the same capabilities, the same task- and primitive-lists can be used for all solutions.
Each separate agent can have its own planning problem, although some agents can refer to the same solution, if goals and capabilities overlap.

4.1.2 HTN Solving

To solve fully the HTN-planning problems, we use the ordered task decomposition (OTD) algorithm (Algorithm 1) (Cheng et al., 2018). This relatively simple algorithm has no strong drawbacks, so it is a great option to test if using an HTN-planner is suitable for crowd simulation. To determine which other algorithm would be better suited for crowd simulation (if necessary), it is important to know what the limitations are when using an HTN-planner for crowd simulation.
For example, if performance needs to be higher in complex scenarios, the OTD algorithm can be swapped for the SHOP or SHOP2 algorithm (Nau et al., 2003). Other algorithms generally do have some constraints to limit complexity (Georgievski & Aiello, 2015). These constraints can be difficult in a crowd simulation environment, as seen in Section 4.2.1. Which algorithm to use is strongly dependent on what constraint can be made for the application and in which situations an increase of performance is necessary.
In addition, we use an adapted version of the OTD algorithm (Algorithm 2), to determine only the next sub-goal for an agent. Only line 12 has changed from returning the primitive + solved task network to returning the primitive + unsolved task list. State and domain are saved to be used for the next query.

Algorithm 1: OTD algorithm (Cheng et al., 2018)

Data: State S, TaskNetwork TN, Domain D
Result: A TaskNetwork containing only primitive tasks

if TN *is empty* **then**
  return *empty plan*;
end
Let t be the first task in TN;
if t *is a primitive task* **then**
  Find an operator op = (h, Pre, Eff) in D such that h unifies with t and S satisfies Pre;
  if *no such* op *exists* **then**
    return *failure*;
  end
  S ← S'. Let S' be S after adding Eff to S;
  TN ← TN'. Let TN' be TN after removing t;
  return [op, OTD(S, TN, D)];
else
  if t *is a compound task* **then**
    Find a method me = (h, (Pre_1, Sub_1), (Pre_2, Sub_2), ...) in D such that h unifies with t;
    Find the task list Sub_i such that S satisfies Pre_i and does not satisfy $\text{Pre}_k, k < i$;
    if *no such* Sub_i *exists* **then**
      return *failure*;
    end
    TN ← TN'. Let TN' be TN after removing t and adding all the elements in Sub_i at the beginning of TN;
    return OTD(S, TN, D);
  end
end

Algorithm 2: OTD-NextStep algorithm

Data: State S, TaskNetwork TN, Domain D
Result: A TaskNetwork containing a primitive tasks at the beginning

```
1  if TN is empty then
2  |  return empty plan;
3  end
4  Let t be the first task in TN;
5  if t is a primitive task then
6  |  Find an operator op = (h, Pre, Eff) in D such that h unifies with t
   |   and S satisfies Pre;
7  |  if no such op exists then
8  |  |  return failure;
9  |  end
10 |  S ← S'. Let S' be S after adding Eff to S;
11 |  TN ← TN'. Let TN' be TN after removing t;
12 |  return [op, TN];
13 else
14 |  if t is a compound task then
15 |  |  Find a method me = (h, (Pre_1, Sub_1), (Pre_2, Sub_2), ...) in D such
   |  |   that h unifies with t;
16 |  |  Find the task list Sub_i such that S satisfies Pre_i and does not
   |  |   satisfy Pre_k, k < i;
17 |  |  if no such Sub_i exists then
18 |  |  |  return failure;
19 |  |  end
20 |  |  TN ← TN'. Let TN' be TN after removing t and adding all the
   |  |   elements in Sub_i at the beginning of TN;
21 |  |  return OTD-NextStep(S, TN, D);
22 |  end
23 end
```

4.2 Complexity

We will split the complexity analysis of HTN planning in crowd simulation into two distinct parts. We will first consider the complexity of HTN planning in general. Secondly, we will consider high level planning in crowd simulation. While obviously dependent on the HTN planner, the second part of the analysis can be approached completely independently from it. In addition, the complexity of high level planning will not change if another planning method, such as STRIPS or even a finite state machine, is chosen instead.

4.2.1 HTN Planner Complexity

The complexity for general HTN planning is strongly dependent on the restrictions put on the planner. (Erol et al., 1994) Because two tasks can reference

each other, it is possible that cycles will happen in solutions. This behavior is exactly what we require for continuous agent plans while waiting, but it makes unrestricted HTN planning an undecidable problem.
The waiting part in itself is a undecidable problem, which is why this part cannot be solved beforehand or for multiple agents at the same time. We will consider agents waiting as a separate problem that will not be considered when discussing complexity.

Restrictions on non-primitive tasks	Must every HTN be totally ordered?	Are variables allowed?	
		no	yes
none	no	Undecidable	Undecidable
	yes	in EXPTIME; PSPACE-hard	in DEXPTIME; EXPSPACE-hard
"regularity"	doesn't matter	PSPACE-complete	EXPSPACE-complete
no non-primitive tasks	no	NP-complete	NP-complete
	yes	Polynomial time	NP-complete

Table 1: Complexity of HTN Planning (Erol et al. (1994))

There are three restrictions that influence the complexity of HTN Planning as seen in Table 1. Propositions can be used instead of variables, the HTN can be totally ordered or (non-primitive) tasks can be restricted to "regularity" (only link to at most one other task) or even no non-primitive tasks.
All restrictions will of course limit the strength of the HTN planning method. As such, some restrictions cannot be used in a crowd simulation domain.
Limiting variables to propositions can be used, since variables can generally be converted to a proposition before the HTN planner. (Check if the train has arrived before planning).
A totally ordered HTN could be used, but the exact order can be difficult to determine. The order would have to change between agents or agent profiles. For example, entering and exiting a train station would require similar steps in a reversed order.
No non-primitive tasks has the strongest effect, but this removes most options from problem solving. Only using "regular" tasks could work, since all non-regular tasks can be split into multiple regular tasks.
It is important to note that HTN planning complexity is mostly interesting for more complex problems. The problems that will be considered in a crowd simulation domain are generally more simple. As such even an exponential complexity would be solved in a very short time for most practical scenarios.
To illustrate, HTN planning has traditionally been used to find solutions in complex domains, such as blocks-world scenarios. In these complex domains, final plans with more than ten steps are common. (Gupta & Nau, 1992) This number generally increases quickly if the scenario gets larger by adding more blocks. If you consider the number of steps required for an agent to move through a train station, you find at most seven steps are required (Figure 2). Increasing the size of the scenario (by adding more platforms, ticket vending machines or ticket acceptance gates) does not increase the number of steps at all.

4.2.2 Crowd Simulation Plan Creation Complexity

In this section we will consider the complexity of creating high-level action plans for agents in crowd simulations. The complexity of the actual creation of the plan is irrelevant for this section. Most, if not all, agents will try to solve a similar problem with similar tools. The actual time it will take for each plan to be calculated will be similar, and, more importantly, only dependent on the number of tasks available. For the complexity analysis of the plan creation, we will assume creating a single plan will be done in $\mathcal{O}(X)$ space and time. Where X can be replaced by the actual complexity depending on the chosen constraints.

The main problem in crowd simulation is dealing with a huge number of agents. Methods that scale directly with the number of agents will become expensive if the number of agents increase. As such, minimizing scaling on the number of agents is vital for methods dealing with crowd simulation. We will consider three approaches to supply each agent with a plan.

1. Maximum Planning Horizon

The first approach is very straightforward. Each agent directly calculates and stores its own plan at spawn. This way, each agent can have its own variables and goals without difficulty. Clearly, this will take $\mathcal{O}(nX)$ time and space, with n being the total number of agents and X being the time and space required to calculate a single plan.
While this is relatively expensive, it is a simple straightforward approach.

2. Minimum Planning Horizon

The second approach does the opposite of the first one. Each agent still directly calculates and stores its own plan, but it only calculates its next step. This requires the same number of calculations, will give the same plan and has the same complexity of $\mathcal{O}(nX)$ time and space, but calculations are spread out over the whole simulation time of the agent, instead of heaving a large number of calculations at a single time.

3. Agent Profiles

The third approach adapts option 1. Plans are again calculated on spawn, but they are also saved for new agents to use. Every combination of goal and subset of all possible (spawn) propositions will have one plan, so the complexity will be $\mathcal{O}(g2^pX)$ time and space with g being the number of possible goals for an agent an p being the number of possible proposition.
While this complexity seems high when considering the exponential complexity for the number of propositions, in practice $g2^p$ cannot become higher than the number of agents n. Since each agent can at most create one plan the total number of plans will always be smaller or equal to n. As such, worst case complexity is still $\mathcal{O}(nX)$ in both time and space.
More importantly, simulations can be run indefinitely. g and p will be limited to any value, but n can potentially keep growing. Therefore, this approach cannot

be more expensive than the others, but depending on the situation, can be a lot cheaper.

4.3 User-centric Design

One of the challenges of using an HTN for users without a lot of programming experience is the amount of information the model requires. Describing primitives (name, precondition, effect) is relatively straightforward, since every primitive task is, by definition, simple and concrete.
The description of higher-level tasks can be a lot more abstract, but, more importantly, there can be a lot more tasks than are actually relevant for the scenario. This would require users to define a lot of (difficult) tasks that may or may not be actually used by the simulation.
To (partially) automate the definition of tasks we propose an automated task generator. The way the algorithm works is similar to how STRIPS-planning works. In STRIPS-planning, preconditions are satisfied by searching for actions that would turn that specific precondition to its required state. (Fikes & Nilsson, 1971)
HTN-planning in itself works differently, it searches for a (user) specified task name that may or may not change variables. To automate the process of defining tasks, there are tasks generated for each (user defined) primitive. This primitive does have some preconditions ("An agent needs a ticket, before passing the ticket gate") and some effects ("After buying a ticket, the agent has a ticket"). With this information we can generate two types of tasks. Precondition Checkers and Compound Tasks.

Definition 4.5. (Precondition Checker) A Precondition Checker is simply a task without any effects, with the precondition of itself. It is generated for each unique primitive. It is basically a Boolean check for each primitive where, if the primitive is true, nothing will happen. If it is false, this task will not be applicable, so another task needs to be chosen.

h: The name of the proposition that will be checked

Pre: A list containing only *h*

Sub: An empty list

Definition 4.6. (Compound Task) A Compound Tasks is a task that satisfies a single effect, by executing the designated primitive. A Compound Task is generated for each effect in each primitive. Each Compound Task is generated as follows:

h: The name of the proposition that will be satisfied by executing the primitive

Pre: A list of all preconditions that are (for any reason) left, because it could not be satisfied automatically. This only contains negative labels.

Sub: A list of all names of the preconditions in the designated primitive, followed by the primitive itself.

By generating these two types of tasks, primitives get linked through their primitives. If a primitive requires a precondition p to be true, the HTN-planner searches for a task named p. It should first find the precondition checker named p (if all precondition checkers are sorted before the compound tasks). If p is already true, the precondition checker will be a valid task, but since it has no effects, the task list will not be changed; no additional action should be taken if the primitive is already true.
If p is not true yet, the precondition checker will not be a valid task, so the HTN-planner will continue searching for a task and find a compound task named p, generated from a different primitive with p as an effect. It will then execute that task, which may contain more primitives to check, but will eventually turn the p true through the primitive designated to the task.

Using this approach, users only need to supply the simulation with concrete primitives and goals for agents. Any required steps in between will be taken care of by the HTN-planner.

4.3.1 Special Cases and Limitations

We consider three special cases for propositions. Global propositions, negative propositions and multiple options.
Global propositions describe scenario wide facts and cannot be altered by any

agent, such as if a train has arrived on the station or not. This can be used to make agents wait for specific events. The creation of a compound task for such a proposition is similar to the normal scenario, with the exception that is proposition is not (guaranteed) changed after performing the corresponding primitive. As such, the sub-tasks for the compound task requires the same task again. If the global proposition has changed, the precondition checker will prevent the compound task from being performed again, otherwise it will be repeated until the global has changed.
Note that global propositions cannot be calculated beforehand. Even in the longer-planning horizon approaches mentioned in Section 4.2.2, these global proposition cannot be evaluated beforehand and will be planned at a later moment, effectively using the minimum planning horizon to evaluate (only) these propositions.

Definition 4.7. (Global Compound Task) A Global Compound Tasks is similar to a normal Compound Task, with the exception that it repeats until the global proposition is satisfied (externally). Each Global Compound Task is generated as follows:

> h: The name of the precondition that will be satisfied by executing the primitive
>
> *Pre*: A list of all preconditions that are (for any reason) left, because it could not be satisfied automatically. This only contains negative labels.
>
> *Sub*: A list of all names of the preconditions in the designated primitive, followed by and the primitive itself and h again.

The second special case is negative propositions, which remove propositions from an agent's state, such as losing money when buying a ticket. These propositions can result in infinite plans or strange behavior. For example, if an agent gets money from an atm to buy a ticket, but loses it by passing through a toll gate, he might decide he needs money to buy a ticket and go back to the atm. Solving this problem is difficult and will increase the the overall complexity of the HTN, because regularity not be an option anymore. (Erol et al., 1994) In principle, negative propositions can be supported, but automatically creating compound tasks for them, while preventing infinite loops, is difficult and probably computationally expensive. As such, it is beyond the scope of this thesis.

If there are multiple options to change a proposition, the HTN will greedily select the first option it finds. As such, the way the complete list of tasks is sorted, will determine the final plan for an agent. This behavior is used for the precondition checks. As long as these are sorted before the compound tasks, a compound task will never be selected if the corresponding proposition is already true.
Because of this behavior, an HTN does not guarantee an optimal shortest path solution. It does, however, guarantee a solution if one exists, as long as there are no negative propositions. If the planning horizon is short, such as when

an agent only plans one step ahead, it always guarantees a next step, if doing anything is possible.

4.4 Implementation Details

4.4.1 Integration in the Tool

As described in the simulation structure of the UUCS engine (Figure 1), the high level planning is separated from the rest of the crowd simulation. This is done in a new AI-Manager class that assigns and keeps track of all goals.
This integrates quite naturally in the existing framework. Where agents originally asked the waypoint-areas for their next goal, they now ask the AI-Manager instead. This means the previously used method of making explicit connections between areas to direct agent paths cannot be used with this new method.
Because this approach has no other links to the existing code, it can be easily changed where necessary. If another HTN solving algorithm is preferred or if the whole HTN approach should be changed, this can also be done easily, because there are no dependencies or assumptions from other classes. As long as the UUCS simulation structure is used, any high-level planning method can easily be integrated with the rest of the simulation.

4.4.2 GUI

User can assign labels to areas if the areas are selected. Depending on the area-type, only certain labels are available.

Waypoints and Exits
As seen in Figure 3, if a goal or waypoint area is selected, an HTN tags element becomes available. Here, users can assign preconditions and effects to this area. All preconditions should be separated with commas.
Special tags can are recognized by the leading character in the tag. '-' is used for negative tags, which requires a tag not to be in a state for preconditions and removes an existing tag for effect. '#' is used to refer to global variables. It functions normally as a precondition. As an effect, the area is a designated spot to wait until the global variable has changed.
If two areas have the same preconditions and effects, they will have the same primitive task in the AI Manager. When a primitive with multiple areas assigned is executed by an agent, an area is selected randomly from all options by using a uniform distribution.
Note that there is no significant difference between waypoints and exits on a high level, aside from exits being seen as potential goals. If an exit would be chosen as waypoint to another goal, agents will still exit the simulation at the first exit. Users can avoid this by only using goal-tags in exit areas. Reversely, if a goal-tag is used in a waypoint area, agent will not exit the simulation after finishing their plan. Instead they will select a random exit to directly remove themselves from the simulation, but users should take care to avoid this.

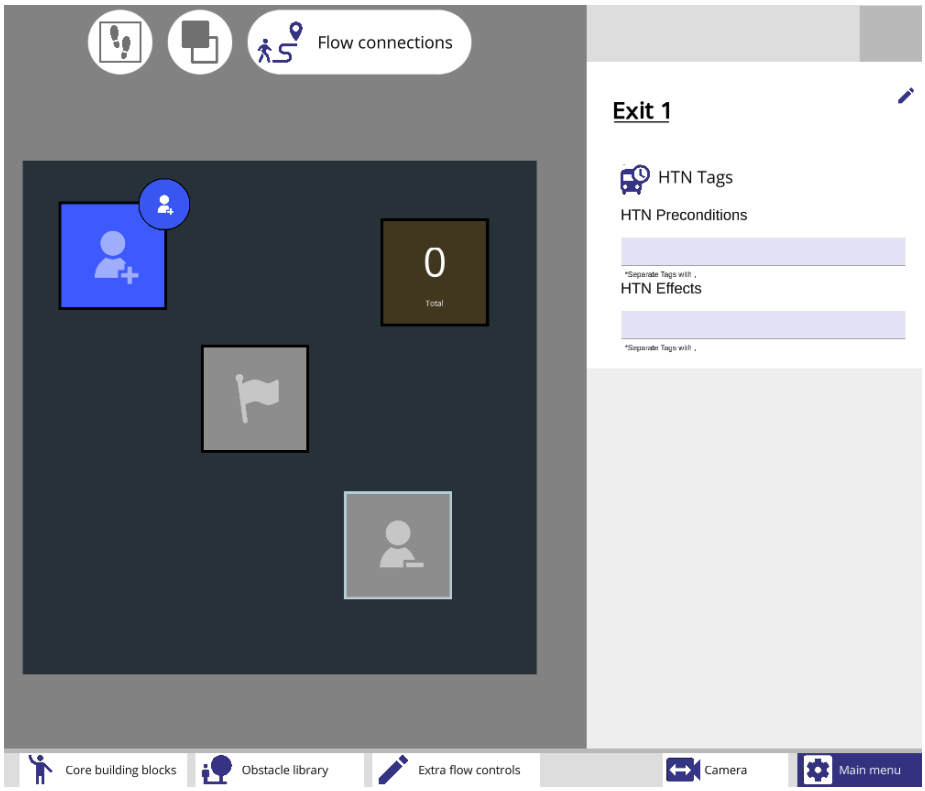


Figure 3: HTN GUI Elements for Goals and Waypoints.

Entries

Similar to the waypoints and exits, if an entry is selected, the corresponding HTN tags become available as seen in Figure 4. Users can supply agents that spawn from this area with goal tags, which should be separated with commas. If multiple tags are provided, each agent will uniformly choose one of the reachable tags. The use of the same tag multiple times is allowed and will change the odds. If a tag does not exist as a goal area effect, is is considered not reachable and will be ignored until such a tag is provided.

In addition, agents can spawn with some state propositions already true. Users should provide these starting states by entering all propositions that are true in that case. Multiple potential starting states can be provided by separating them with a '|'. Additionally, each state should be preceded by an integer. This integer gives the chance that state is selected. Each state i has a chance of $(stateChance_i / \sum_n (stateChance_n))$ to be selected.

For example, the string "1,HasMoney,Outside|2,HasTicket,Outside" would spawn 1/3 of the agents with the tags "HasMoney" and "Outside". The remaining 2/3 would spawn with "HasTicket" and "Outside".

Figure 4: HTN GUI Elements for Entries.

Global propositions Finally, global propositions can be changed by using a counter object as seen in Figure 5. This already existing object has been chosen, because there is an already existing link with simulation timers. This way, global tags can change based on simulation times. User can separate different tags with '—' and tags and timing should be separated with ','.
If a user only enters a tag, without correct timing information, this tag is considered true, until the user changes it. In other words, they are considered manual tags. By adding two or more integers after the tag, those tags will automatically alter based on simulation time in seconds.
For example, the string "trainArrived,5,25" will make the global variable "trainArrived" true for 5 seconds, then false for 25 seconds, then true for 5 seconds and so on. Leading with a "-" will start with that variable as false instead of true.

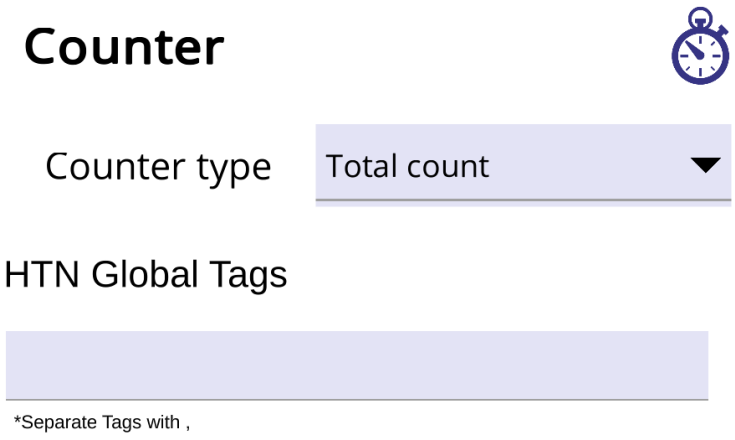


Figure 5: HTN GUI Elements for global propositions.

5 Experiments and Results

5.1 Experiment 1 - Train Station

5.1.1 Scene

The first experiment examines the performance of the HTN in a practical scenario; a train station. The design of the train station can be seen in Figure 6.

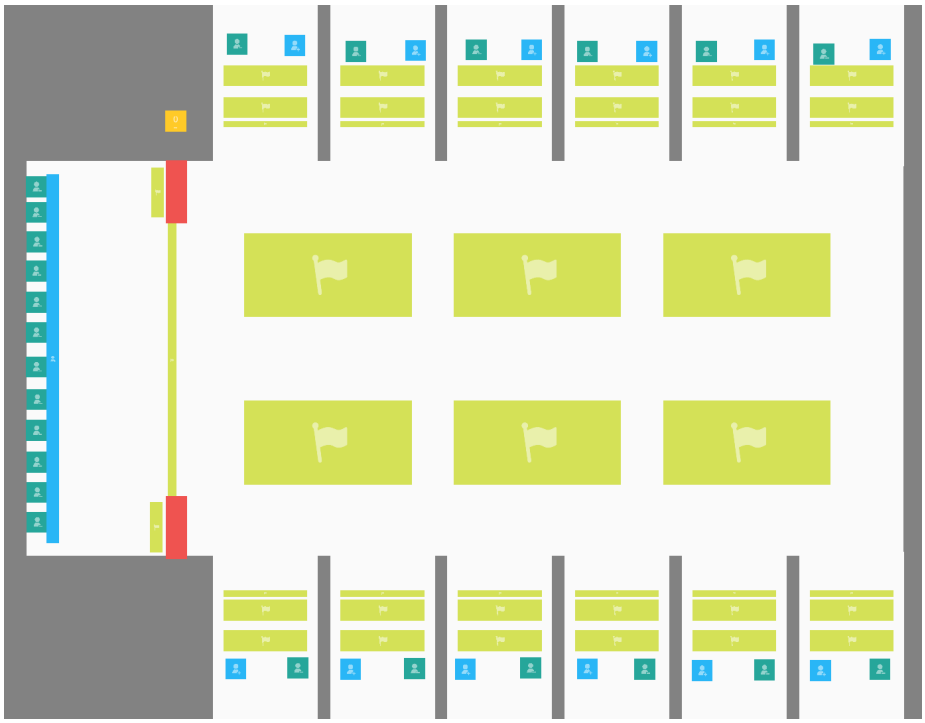


Figure 6: The design of the train station scene. Entrance (blue) and exits (green) are located to the left. The waypoints (yellow) directly left of the red obstacles are ticket booths. The waypoint between the obstacles is used to enter the (main) station. The six large waypoints in the middle are waiting areas. The twelve areas on the top and bottom of the waiting zone are track platforms.

In this scenario we consider two types of agents, one type spawns at the left entrance of the station and has one of the twelve track platforms as a goal. The other type spawns at one of the track platforms and has the left exit as a goal.

The defined waypoint areas are as follows:

- Ticket Booth (2 locations) - Agents can buy a train ticket if they don't have one
- Entry Gate - Enter the main part of the station, if agent has a ticket
- Waiting General (6 locations) - Agents can wait here until their train is close
- Go to Platform (12 locations) - Agents can move here if their train is coming soon
- Wait at Platform (12 locations) - Agents can wait here until their train arrives

- Wait till Deboard (12 locations) - Agents can wait here until the train they are on arrives at the station

The specific preconditions and effects used can be found in Section 7 in the Appendix. Solving the HTN for these waypoints results in the behavior as seen in Figure 7. In short, agents make sure they have ticket, enter the station, visit waiting areas until their train almost arrives, go to their platform, wait until the train has arrived and finally board the train.
Note that, depending on the planning method used, each agent calculates this diagram as it spawns, calculates only the next step or copies it from a previous agent with the same spawn conditions.

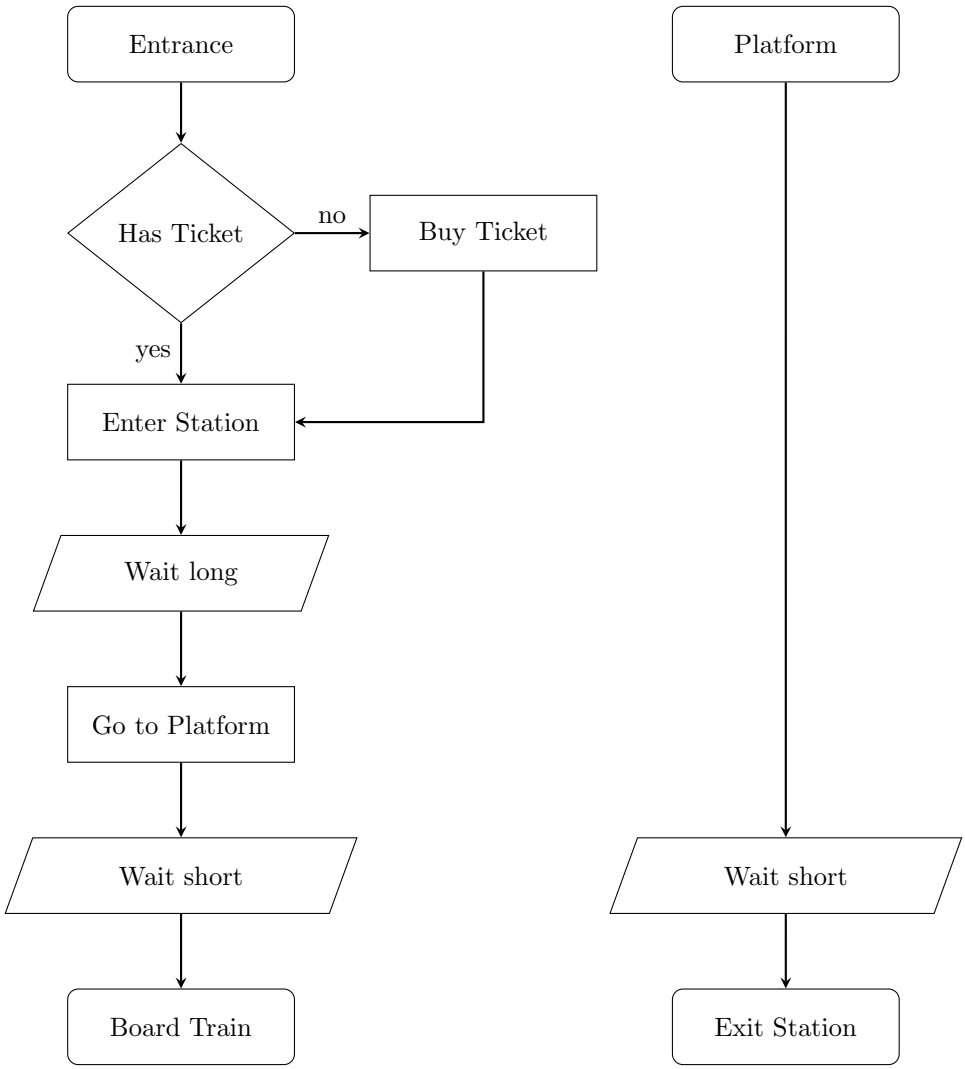


Figure 7: Left: The flow diagram of an agent trying to board a specific train. The waiting tasks are repeated until the train is close for the long one or arrived for the short. Note that this means agents can wait for a long time at the short waiting location if they initially just miss their train.
Right: The flow diagram of an agent trying to exit the station. They can only leave the train if it has arrived at the station.

5.1.2 Method

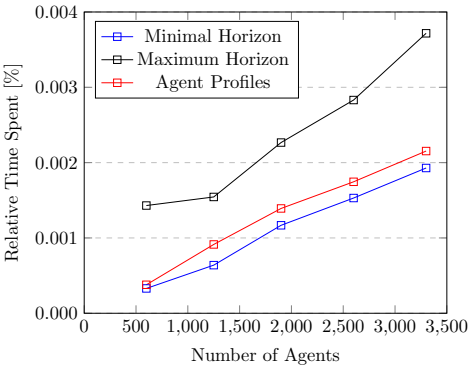
To determine how increasing the number of agents in the scene affects the high level planning times, we measured the amount of time spent on spawning all agents and the time spent on determining a next goal. The sum of this is the

total amount spent on high level planning. We measured the planning times each time for 60 seconds.
To vary the number of agents, we changed the spawn rates for all areas. We started at 0.25 agents per second for the agents arriving by train and 1 agent per second for agents arriving at the entrance. This was increased in steps of 0.25 and 1 respectively to a maximum of 1.25 and 5 agents per second. This was the limit of the testing system.
We measured the aforementioned planning times at a speed multiplier of x8. Before measuring, we let the simulation run for a while, to stabilize the number of agents in the simulation. This resulted in measurements for about 600, 1250, 1900, 2600 and 3300 agents at the same time in the simulation.
We did this for all three plan creation methods as described in Section 4.2.2; Maximum Planning Horizon, Minimum Planning Horizon and Agent Profiles. Each planning method had its own separately compiled executable. The raw data used for all graphs can be found in the appendix in Section 7. Detailed statistics can be seen in Table 2.

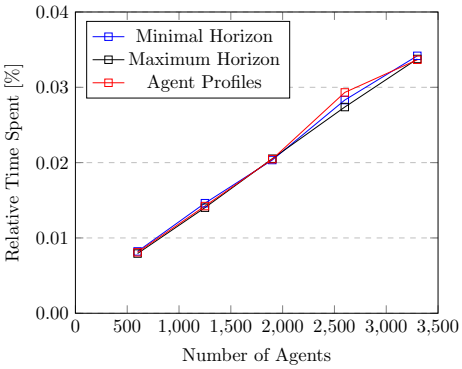
5.1.3 Results

As seen in Figure 8, increasing the number of agents increases the time spent planning, regardless of the planning method used. In all nine situations, we found a significant linear relationship ($p < 0.001$) between the number of agents and relative time spent on high level planning, using simple linear regression. Only in the case of using a Maximum Planning Horizon, does the planning approach differ from the others. As expected, a Maximum Planning Horizon takes more time spawning new agents.
We expected differences between the planning approaches at the other levels. We expect that this is because of the dynamic planning used in the train station scenario. The static plan (the part that agent can plan beforehand) is relatively small in this scenario (3-4 steps for boarding a train, 1 step for leaving the station). The dynamic part, i.e. walking around while waiting, is generally triggered a lot more (0-40 times per agent, depending on when an agent spawned). This means that in most cased, the part that cannot be determined beforehand makes up the largest part of the plan.
It stand to reason that we would not find a clear difference between the planning approaches, since they only differ on the static part.

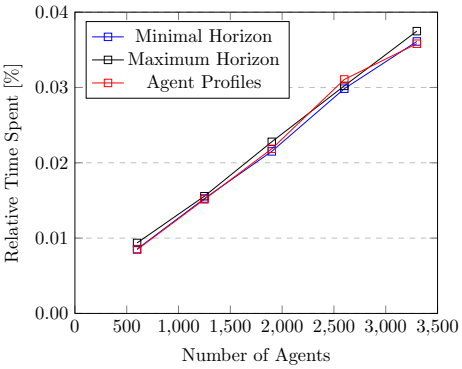
However, we did find that the relative time spent on high level planning is quite small and, more importantly, seems to scale linearly with the number of agents used. As such, an HTN can most likely be used to simulate larger number of agents using dynamic plans, which was the reason to actually use the HTN.



(a) Relative Time Spent on Spawning per Agent



(b) Relative Time Spent on Determining a Next Goal per Agent



(c) Relative Time Spent on All High Level Planning per Agent

Figure 8: Relative Time Spent on High Level Planning

Approach	Measurement	F	p	R^2
Minimal Horizon	Spawning	$F(1,4) = 1.67 \times 10^{-5}$	$p < .0005$	.998
	Goal Selection	$F(1,4) = 3.15 \times 10^{-5}$	$p < .0005$	.998
Maximum Horizon	Spawning	$F(1,4) = 7.32 \times 10^{-4}$	$p < .0005$	.981
	Goal Selection	$F(1,4) = 1.18 \times 10^{-6}$	$p < .0005$	.998
Agent Profiles	Spawning	$F(1,4) = 2.79 \times 10^{-5}$	$p < .0005$	.998
	Goal Selection	$F(1,4) = 3.79 \times 10^{-6}$	$p < .0005$	.997

Table 2: The predictability of high level planning times using simple linear regression.

5.2 Experiment 2 - High Complexity Plan

5.2.1 Scene

In experiment 1 we looked at the relative performance of the HTN, compared to the lower levels of crowd simulation in a realistic scenario. In this second experiment, we want to determine how well an HTN performs with high complexity plans.

5.2.2 Method

The complexity of HTN planning is based on the plan length, as such, we only need to create a long plan, not a plan where a lot of (conditional) choices need to be made. To obtain such a long HTN plan, we automatically generate it.
We do this by starting with a given number of primitives. Then, we create a second layer of compound tasks. Each compound task links two primitives. There can be one compound task that refers to a single primitive, if there is an uneven number of primitives. We continue adding new layers, until there is a layer with only one compound task. This compound task will be the goal task that will be evaluated.
Usually, a primitive can only be executed if there is at least one action linked to it. Since we are not interested in the lower planning levels for this experiment, primitives are instantly executed, not requiring lower level planning at all.
Note that we only use 'static' tasks for this experiment. As seen in the previous experiment, use of 'dynamic' tasks, such as waiting, effectively forces the planning approach to minimal horizon.

To compare the performance of the minimal horizon, maximum horizon and agent profiles planning approaches, we evaluate how much time is spent spawning agents and how much time is spent deciding on a next (sub)goal for each of the approaches. We do this for a varying plan length. (100/200/500/1000 primitives) To simulate a varying number of agents, we repeat it between 1 and at most 1 million times. (1/2/5/10/20/50/100/.../1000000) If a test with a certain number of agents would take more than 4 minutes, we won't be testing a higher number of agents.

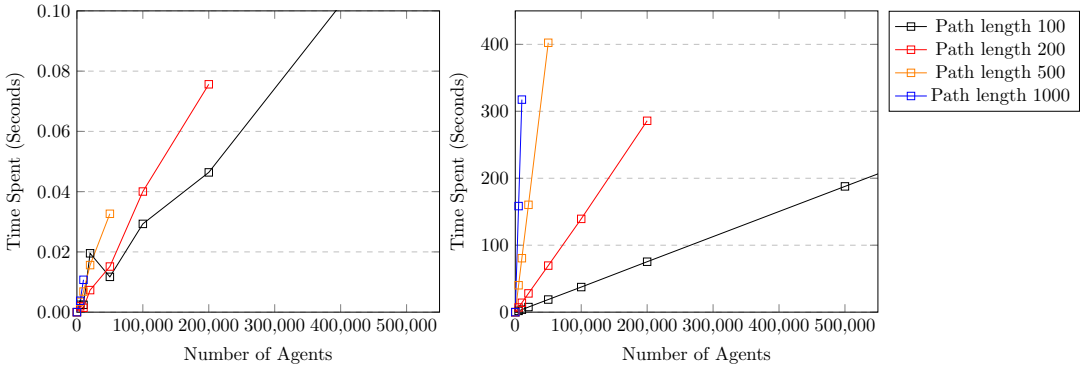
5.2.3 Results

The results from experiment 2 can be found in Figures 9, 10, 11. Raw data can be found in Tables 14 to 19 in the Appendix. From the graphs, we can clearly see that the time spent planning scales linearly based on the number of agents in all cases. As can be seen in Table 3, the number of agents and path length van be used to very strongly predict the time spent on high level planning.
For the minimal horizon approach, the path length did not add significantly to the spawning time prediction. We assume this is because spawning times were very low (at most 0.26 seconds), and they were often rounded from down from smaller than 0.00024 seconds to 0 by the timer. Path length did add significantly

for the goal selection time prediction, as well as the number of agents in both cases ($p < .05$).
For the maximum horizon approach, both path length and the number of agents added significantly to the prediction ($p < .05$).
Finally, for the agent profiles approach, the number of agents added significantly to both predictions ($p < .05$), but path length did not in both cases. As can also be seen in Figure 11, this means the agent profiles approach successfully eliminates the effect of the path length on high level planning times and, by extension, plan complexity.
While the 5 minutes spent on determining subgoals for 1 million agents in the Agent Profiles approach might seem long, this time would be spread out over the whole path of all agents in a real simulation. This means less than 0.0005 seconds would be spend per agent over its whole path on high level planning.

Approach	Measurement	F	p	R^2
Minimal Horizon	Spawning	$F(2, 62) = 1991.232$	$p < .0005$	.985
	Goal Selection	$F(2, 62) = 32.188$	$p < .0005$	.509
Maximum Horizon	Spawning	$F(2, 62) = 31.254$	$p < .0005$	.502
	Goal Selection	$F(2, 62) = 149.520$	$p < .0005$	.828
Agent Profiles	Spawning	$F(2, 62) = 102568.507$	$p < .0005$	1.000
	Goal Selection	$F(2, 62) = 49753481.57$	$p < .0005$	1.000

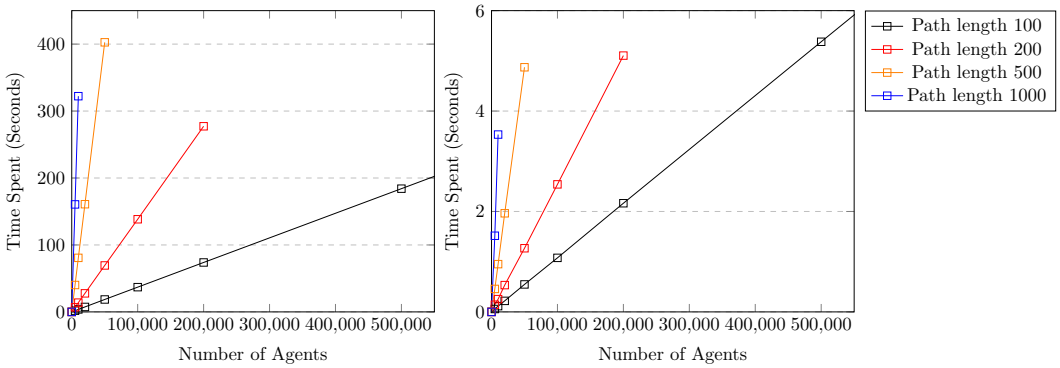
Table 3: The predictability of high level planning times using multiple linear regression.



(a) Time Spent on Spawning Agents

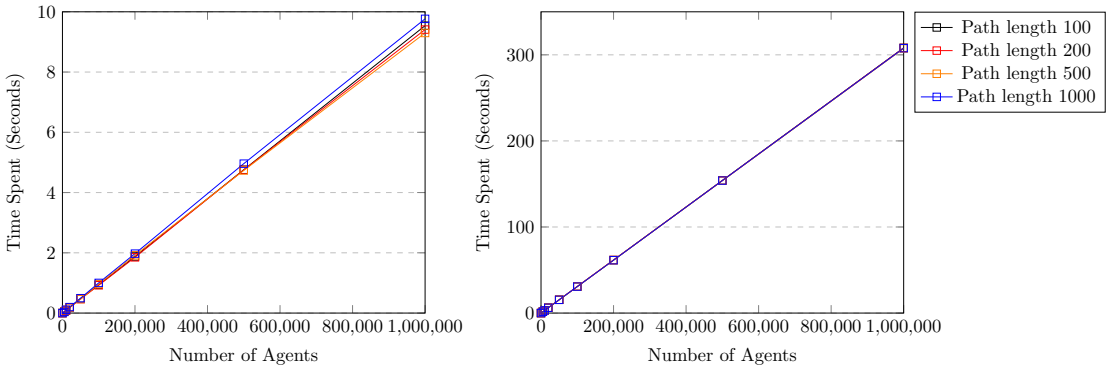
(b) Time Spent on Determining (Sub)Goals

Figure 9: Time spent on high level planning using the Minimal Horizon Approach



(a) Time Spent on Spawning Agents (b) Time Spent on Determining (Sub)Goals

Figure 10: Time spent on high level planning using the Maximum Horizon Approach



(a) Time Spent on Spawning Agents (b) Time Spent on Determining (Sub)Goals

Figure 11: Time spent on high level planning using the Agent Profiles Approach

6 Conclusion and Discussion

In experiment 1, we found that the relative time spent on high level planning is very low, when using the proposed Hierarchical Task Network. The impact for 3500 agents was lower than 0.04%, so this method can definitely be used to increase high level planning capabilities for crowd simulation software. Although the minimum horizon spawns agents a bit faster, when a lot of dynamic tasks are used, there is no efficiency difference in total planning time between the planning approaches. All approaches are forced to plan using a minimal horizon.
In experiment 2 we have looked further into complex tasks and the difference between the minimal horizon, maximum horizon and agent profiles approaches. As expected, we have seen that the minimum horizon spawns agents very quickly, but spends a lot of time planning subsequent (sub)goals. The maximum horizon swaps these times; long spawning times, but very short (sub)goal choices. In practice, the (sub)goal planning is spread over quite some time. There will most likely be a few seconds between each subgoal, although the exact times will differ vastly between scenarios. As such, the relatively long time planning (sub)goals in the minimum horizon approach would actually be spread out over time, possibly making the effective impact difficult or impossible to notice.
In both horizon approaches, we have seen that planning time scales linearly on both the number of agents and on path length. The newly proposed agent profiles approach completely eliminates the effect of path length on planning times. In addition, total planning times for the agent profiles approach are lower than those of the other approaches. If there are a lot of agents pursuing similar tasks, this approach can always be used, even if another planning method than the HTN is chosen.

6.1 Future Work

There are multiple ways to improve on the high level planning methods as proposed in this thesis. The current method of automatically generating compound tasks is limited. Adding support for negative labels would allow for higher complexity plans. This would require loop detection, which would be a performance hit.
Alternatively, manual compound tasks would also improve plan complexity. This would allow users to quickly define the steps an agent would have to take. Due to the structure of the HTN, users would be able to choose their preferred level of abstraction themselves. Ranging from selecting specific areas agents would need to visit to instructing agents to making a specific label true, depending on the existing tasks.

Using a minimal planning horizon, a next step in the plan could also be determined if the agent has not reached its goal yet. If this is done at a moment the system would be sleeping until the next tick for low-level planning, high level planning would not affect frame-rate at all.

Currently, if an agent can choose between multiple areas with the same results, a random area is chosen. This can be extended by letting agents choose the closest one or the otherwise most attractive area, depending on, for example, the density of other agents. This approach does not improve high level planning-time, -complexity or -quality, but it does improve agent travel-time. Additionally agent behavior would at least look more natural and could be considered more realistic.
Similarly, the OTD-algorithm can be improved by changing the order tasks are considered. This is currently done in a fixed order, but it could be sorted in a different way. It could be done based on expected chance a task is chosen, distance from where a task would be executed or density of other agents at or around the task or a random order, as done in the OTD-h method (Cheng et al., 2018). This would require further testing, but could improve planning times and could decrease the times agents would choose a wrong task, increasing plan quality.

Another way to improve plan quality and fix agents choosing a wrong task, would be by adding backtracking to the greedy OTD algorithm. This would improve plan quality and it could solve the problem of merging agents streams with different origins as described in section 4.3.1. Backtracking could definitely become expensive, but since the algorithm has proven to be quite cheap, a (small) hit in efficiency should be acceptable. Alternatively, a different algorithm could be chosen. For example, the SHOP2 algorithm (Nau et al., 2003) shows great performance and additionally allows quantifiers in the labels, allowing agents to get, for example, a varying amount of money, depending on their destination.

References

Cheng, K., Wu, L., Yu, X., Yin, C., & Kang, R. (2018). Improving hierarchical task network planning performance by the use of domain-independent heuristic search. *Knowledge-Based Systems*, *142*, 117–126.

Curtis, S., Best, A., & Manocha, D. (2016). Menge: A modular framework for simulating crowd movement. *Collective Dynamics*, *1*, 1–40.

Dastani, M. (2008). 2apl: a practical agent programming language. *Autonomous agents and multi-agent systems*, *16*(3), 214–248.

Dignum, F., Morley, D., Sonenberg, E. A., & Cavedon, L. (2000). Towards socially sophisticated bdi agents. In *MultiAgent Systems, 2000. Proceedings. Fourth International Conference on*, (pp. 111–118). IEEE.

Erol, K., Hendler, J., & Nau, D. S. (1994). Htn planning: Complexity and expressivity. In *AAAI*, volume 94, (pp. 1123–1128).

Fikes, R. E. & Nilsson, N. J. (1971). Strips: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, *2*(3-4), 189–208.

Georgievski, I. & Aiello, M. (2015). Htn planning: Overview, comparison, and beyond. *Artificial Intelligence*, *222*, 124–156.

Guizzi, G., Murino, T., & Romano, E. (2009). A discrete event simulation to model passenger flow in the airport terminal. *Mathematical methods and applied computing*, *2*, 427–434.

Gupta, N. & Nau, D. S. (1992). On the complexity of blocks-world planning. *Artificial Intelligence*, *56*(2-3), 223–254.

InControl (2020, January 20). Pedestrian dynamics. Retrieved from `https://www.incontrolsim.com/software/pedestrian-dynamics/`.

Jolly, K., Ravindran, K., Vijayakumar, R., & Kumar, R. S. (2007). Intelligent decision making in multi-agent robot soccer system through compounded artificial neural networks. *Robotics and Autonomous Systems*, *55*(7), 589–596.

Kaelbling, L. P. & Lozano-Pérez, T. (2010). Hierarchical planning in the now. In *Workshops at the Twenty-Fourth AAAI Conference on Artificial Intelligence*.

Kelly, J. P., Botea, A., Koenig, S., et al. (2008). Offline planning with hierarchical task networks in video games. In *AIIDE*.

Lekavỳ, M. & Návrat, P. (2007). Expressivity of strips-like and htn-like planning. In *KES International Symposium on Agent and Multi-Agent Systems: Technologies and Applications*, (pp. 121–130). Springer.

Li, J. P. (2000). Train station passenger flow study. In *Proceedings of the 32nd conference on Winter simulation*, (pp. 1173–1173). Society for Computer Simulation International.

Musse, S. R. & Thalmann, D. (2001). Hierarchical model for real time simulation of virtual human crowds. *IEEE Transactions on Visualization and Computer Graphics*, *7*(2), 152–164.

Nau, D. S., Au, T.-C., Ilghami, O., Kuter, U., Murdock, J. W., Wu, D., & Yaman, F. (2003). Shop2: An htn planning system. *Journal of artificial intelligence research*, *20*, 379–404.

Oasys (2020, January 20). Massmotion. Retrieved from `https://www.oasys-software.com/products/pedestrian-simulation/massmotion/`.

Paris, S. & Donikian, S. (2009). Activity-driven populace: a cognitive approach to crowd simulation. *IEEE Computer Graphics and Applications*, *29*(4), 34–43.

Sacerdoti, E. D. (1975). A structure for plans and behavior. Technical report, SRI INTERNATIONAL MENLO PARK CA ARTIFICIAL INTELLIGENCE CENTER.

Shao, W. & Terzopoulos, D. (2007). Autonomous pedestrians. *Graphical models*, *69*(5-6), 246–274.

Sung, M., Gleicher, M., & Chenney, S. (2004). Scalable behaviors for crowd simulation. In *Computer Graphics Forum*, volume 23, (pp. 519–528). Wiley Online Library.

ThunderheadEngineering (2020, January 20). Pathfinder. Retrieved from `https://www.thunderheadeng.com/pathfinder/`.

Tozička, J., Jakůbuv, J., & Komenda, A. (2014). Generating multi-agent plans by distributed intersection of finite state machines. In *Proceedings of the Twenty-first European Conference on Artificial Intelligence*, (pp. 1111–1112). IOS Press.

Ulicny, B. & Thalmann, D. (2002). Towards interactive real-time crowd behavior simulation. In *Computer Graphics Forum*, volume 21, (pp. 767–775). Wiley Online Library.

van Toll, W., Jaklin, N., & Geraerts, R. (2015). Towards believable crowds: A generic multi-level framework for agent navigation.

7 Appendix

Software Packs

Building Blocks

The following tables give an overview of what actions are available in current crowd simulation software. Reported actions are features any of the software packs report themselves.
If a pack supports an action it is indicated with a check-mark (✓). If it does not support it, a cross is used (×). Some of the software packs allow for custom scripting. If an action can only be performed using scripting, an s is used (S). Finally, if an action is supported, but in a roundabout way, it is indicated with a tilde after the check-mark (✓∼).
Information for Pedestrian Dynamics, Golaem, Mass Motion and Menge is retrieved from their user manuals. As PedSim is an open source project, its source code is available. Since the Legion's user manual is not freely available, not all information could be obtained, and some blocks are left empty. For information about Pathfinder a free trial version was used.

Agents Actions

These are actions that can be performed by a single agent, ways to work together with other agents or changing attributes of a single agent. The following actions are included:

Set Goal: Set a single goal location for an agent.
Add Temporary Goal: Adds a temporary goal for an agent. This can be done re-actively during path following, so it does not have to be predefined.
Wait: Let the agent wait until some condition is met.
Queuing: Agents can wait in line with other agents.

Select new action plan: Changes all high-level goals of the agent (i.e. evacuations).
Behavior in vehicles: Agents can behave differently in and out of vehicles.
Change Speed: Changes the (maximum) movement speed of a single agent.
Variable walk speed per agent: Different agents can have different movement speeds.
Variable agent size: Different agents can have a different size.

Follow another agent: Lets an agent follow another agent.
Flock: Lets agent move together using flocking behavior.
Formation: Lets agents move together in a formation.

Action	G	L	MM	M	PD	PF	PS	SimCrowds
Set Goal	✓	✓	✓	✓	✓	✓	✓	✓
Add Temporary Goal	S		✓	✓	S	×	✓	✓
Wait	✓	✓	✓	✓	✓	✓	✓∼	✓
Queuing	S		✓	×	S	×	×	×
Select new action plan	×		✓	×	✓	×	✓∼	✓
Behavior in vehicles	✓	✓	✓	✓	✓	×	×	×
Change Speed	✓		✓	✓	✓	✓	✓	✓
Variable walk speed per agent	✓	✓	✓	✓	✓	✓	✓	✓
Variable agent size	✓	✓	✓	×	×	✓	×	✓
Follow another agent	✓∼	✓	S	✓	S	✓	✓	✓
Flock	✓		×	×	×	×	×	✓
Formation	✓		×	✓	×	×	×	✓

Table 4: Available Agent Action for G:GOLAEM, L:Legion, MM:Mass Motion, M:Menge, PD:Pedestrian Dynamics, PF:Pathfinder, PS:PedSim and SimCrowds

Agent Tests

These are tests that can be performed on or by single agents.

Location visited: Agents can remember if they visited a location.
Specific area: Agents can be check if they are in a specific area.
Type of area: Agents can check if they are in a type of area.
Distance to area: Agents can check the distance to an area.

Test	G	L	MM	M	PD	PF	PS	SimCrowds
Location Visited	S		✓	×	✓	×	×	×
Specific area	✓		✓	×	S	✓	×	✓
Type of area	✓		S	×	S	✓	×	✓∼
Distance to area	✓		✓	×	S	×	×	✓

Table 5: Available Agent Tests for G:GOLAEM, L:Legion, MM:Mass Motion, M:Menge, PD:Pedestrian Dynamics, PF:Pathfinder, PS:PedSim and SimCrowds

Areas

These are areas that can be defined and environmental features that can be modeled.

(Dis)Comfort zones: Zones that agents prefer or avoid.
Directional passages: Passages that can only be passed in a certain direction.
Change directional passages: Change the direction of directional passages or

change them to bi-directional passages, while the simulation is running.
Entry/Exit flow speed: Limit the maximum flow speed for a passage. (I.e. entering a bus).
Entry/Exit capacity: Limit the maximum capacity for a passage. (I.e. entering a bus).
Stairs: Stairs to move from one level to another.
Spiral stair: Spiral stairs to move from one level to another.
Escalator: Escalator to move from one level to another.
Lifts: Lifts/Elevator to move between levels.
Ramps: Ramps to move between levels.
Secondary Facilities: Areas that allow agents to fulfill secondary needs (i.e. shopping areas, bathrooms, etc.).
Traffic: Moving objects like cars that agents generally try to avoid.

Area	G	L	MM	M	PD	PF	PS	SimCrowds
(Dis)Comfort zones	×		×	×	✓	×	×	✓
Directional passages	S	✓	✓	×	✓	✓	×	✓
Change directional passages	S		S	×	✓	✓	×	✓
Entry/Exit flow speed	S	✓	✓	✓	✓	✓	✓∼	✓
Entry/Exit capacity	S	✓	S	×	✓	×	×	✓
Stairs	×		✓	✓	✓	✓	×	×
Spiral stair	×		✓∼	✓∼	✓∼	✓∼	×	×
Escalator	×	✓	✓	×	✓	✓	×	×
Lifts	×	✓	✓	×	×	✓	×	×
Ramps	×		✓	✓	✓	✓	×	✓
Secondary Facilities	✓		✓	✓	✓	✓∼	✓	✓
Traffic	✓		×	×	×	✓∼	×	✓

Table 6: Available Areas for G:GOLAEM, L:Legion, MM:Mass Motion, M:Menge, PD:Pedestrian Dynamics, PF:Pathfinder, PS:PedSim and SimCrowds

Global

These are methods that can change the behavior of agents globally.

Distribute agents: Distribute agents over zones or goals, so those are visited equally (e.g. choosing between queues).
Vector fields: Change the behavior of agents using vector fields.

Tools

These are tools to combine the actions or tests in certain ways:

Area	G	L	MM	M	PD	PF	PS	SimCrowds
Distribute agents	×		✓	×	S	×	×	✓
Vector fields	✓		×	✓	×	×	×	×

Table 7: Available global controls for G:GOLAEM, L:Legion, MM:Mass Motion, M:Menge, PD:Pedestrian Dynamics, PF:Pathfinder, PS:PedSim and SimCrowds

Next block: Sequentially performs the next defined action.
Previous block: Revert to the previous block (Create a loop in the action plan).
Conditional block: Perform the next action only under certain conditions.
Probability for block: Perform the next block with a certain chance.
Distribution for set of blocks: Choose one of the specified blocks using a certain distribution.
Action Timer: Perform an action after a certain amount of time.
Events: Perform an action during an event (i.e. notice of a train cancellation).

Area	G	L	MM	M	PD	PF	PS	SimCrowds
Next block	✓		✓	✓	✓	✓	✓	×
Previous block	✓		×	✓	×	×	×	×
Conditional block	✓		✓	✓	✓	×	×	×
Probability for block	✓	✓	✓	✓	✓	×	×	×
Distribution for set of blocks	✓	✓	✓	✓	✓	✓	×	×
Action Timer	✓	✓	✓	✓	✓	✓	×	✓
Events	✓	✓	✓	✓	✓	×	×	✓∼

Table 8: Available tools for G:GOLAEM, L:Legion, MM:Mass Motion, M:Menge, PD:Pedestrian Dynamics, PF:Pathfinder, PS:PedSim and SimCrowds

Scenes

Train Station

1. None of the models use vehicles
2. All models can set an action plan, although PedSim creates cyclic repeating goals, which would not stop after boarding the train. This could be solved by deleting agents under certain conditions, but it is not very user friendly.
3. Deciding to get a ticket can be done in a few ways
   (a) Distribution over both tasks
   (b) Create a linear plan (... → buy ticket → go to gate → ...) with a chance to skip the buying of the ticket.

Action	G	L	MM	M	PD	PF	PS	SimCrowds
Use Vehicles	×	×	×	×	×	×	×	×
Action Plan	✓	✓	✓	✓	✓	✓	×	×
Ticket or gate decision	✓	✓	✓	✓	✓	✓	×	✓
Queuing	S		✓	×	S	×	×	×
Passing gate	×	✓	✓~	×	×	×	×	✓
Waiting	✓	✓	✓	✓	✓	✓	×	✓
Boarding	×	✓	✓	✓~	✓	✓	×	×

Table 9: Building blocks required to model a train station for G:GOLAEM, L:Legion, MM:Mass Motion, M:Menge, PD:Pedestrian Dynamics, PF:Pathfinder, PS:PedSim and SimCrowds

(c) Conditional choice (e.g. if the agent has a ticket → go to gate)

4. Only Mass Motion claims to implement queuing, by using a fixed action chain.

5. Only Mass Motion claims to implement passing ticket gates using "banking". This means that when agents must choose between gates, they ignore the distance from the gate to their final goal and only use distance to the gate and density of the crowd. This can solve the problem, but can also result in weird behavior in some scenarios.
   For example, if the area on the other side of the gates is crowded, an agent can decide to use to a (on the current side) less crowded, but farther away gate as a detour. After he passed the gate and has to move back to his original route, he has to move through the (unaccounted) crowded side, which would take more time.

6. Waiting can be done with all methods except PedSim.

7. Boarding a train has been implemented by Pedestrian Dynamics and Mass Motion, with speed and capacity limited goals. Menge could solve this too by using manual conditional logic to limit the number of agents boarding. Other methods do not have way to limit the number of agents trying to board, although this might not be necessary is normal situations.

If we ignore the use of vehicles, only Mass Motion can fully model the train station. Golaem, Menge and Pedestrian Dynamics cannot handle the advanced tasks; queuing and gate passing. Pathfinder cannot handle the gate passing.

Airport

Pedestrian Dynamics cannot handle groups. It also has trouble to let agents make decisions based on data instead of distributions, although this could be solved using conditional logic.
GOLAEM and Menge work a bit better with decision making using the finite

Action	G	L	MM	M	PD	PF	PS	SimCrowds
Grouping	✓~	×	×	×	×	✓	✓	✓
Multiple behavior types	✓	✓	✓	✓	✓	✓	✓	✓
Choose check-in desk	✓		✓	✓	✓~	✓~	×	✓
Queuing	S		✓	×	S	×	×	×
Decide on activities before security	✓		×	✓	✓	✓~	×	×

Table 10: Building blocks required to model an airport for G:GOLAEM, L:Legion, MM:Mass Motion, M:Menge, PD:Pedestrian Dynamics, PF:Pathfinder, PS:PedSim and SimCrowds

state machines, but still cannot handle queues. GOLAEM claims to be able to handle groups, but this is limited to setting an agent as a goal.
Mass Motion can handle the distribution over the desks and queuing using their fixed action chain to distribute agents and let them wait in queue.
PedSim can handle groups, but not much more than that.
Pathfinder can handle all blocks, although choosing check-in desks and activities before security cannot be handled straightforward.

Evacuation

Action	G	L	MM	M	PD	PF	PS	SimCrowds
Trigger	✓	✓	✓	✓	✓	✓	×	✓
Change Behavior	×		✓	×	✓	×	×	✓
Set Goal	✓	✓	✓	✓	✓	✓	✓	✓
Change Speed	✓	✓	×	✓	×	✓	×	✓
Change Gate Direction	S		S	×	✓	✓	×	✓

Table 11: Building blocks required to model an evacuation for G:GOLAEM, L:Legion, MM:Mass Motion, M:Menge, PD:Pedestrian Dynamics, PF:Pathfinder, PS:PedSim and SimCrowds

The trigger and behaviour/goal change is supported by all software packs except PedSim. Only Pedestrian Dynamics, Pathfinder and SimCrowds can change the direction of the gates. In evacuation scenarios all gates can open, making them bi-directional. They use the same technique to open emergency exists.

Scene Design

Train Station

The following settings have been used for each area in the train station for agents to board a train.

- Globals (Change global values based on a timer)
  - TrainXAlmost,10,50
  - -TrainXHere,5,5,50
    Both repeated for each platform
    Train is considered "almost" at the platform for the first ten minutes each hour.
    Train is considered at the platform for the second five minutes each hour.
- Entry
  - Goal: BoardTrain1, BoardTrain2, ... (One for each platform)
  - Start Tags: 50,HasMoney,Outside—50,HasTicket,Outside
    Always start outside. 50% chance to start with a ticket.
- BuyTicket (Same for each ticket booth)
  - Preconditions: Outside,-HasTicket,HasMoney
  - Effects: HasTicket,-HasMoney
- BuyTicket
- EnterGate (Same for each ticket booth)
  - Preconditions: Outside,HasTicket
  - Effects: InsideStation,-Outside
- WaitGeneral (Same for each waiting area)
  - Preconditions: InsideStation
  - Effects: #TrainXAlmost
- GoToPlatformX (Repeated for each platform)
  - Preconditions: InsideStation,#TrainXAlmost
  - Effects: AtPlatformX
- WaitAtPlatformX (Repeated for each platform)
  - Preconditions: InsideStation,AtPlatformX
  - Effects: #TrainXHere
- BoardTrainX (Repeated for each platform) - Exit Area
  - Preconditions: AtPlatformX,#TrainXHere
  - Effects: BoardedTrainX

The following settings have been used for each area in the train station for agents to leave the station.

- Entry (Repeated for each platform)
  - Goal: ExitX
  - Start Tags: 20,AtPlatformXExit
- WaitTillDeboard (Repeated for each platform)
  - Preconditions: AtPlatformXExit
  - Effects: #TrainXHereExit
- ExitX (Repeated for each platform) - Exit Area
  - Preconditions: #TrainXHereExit
  - Effects: ExitX

Raw Data

This section shows the raw data obtained from the experiments described in Section 5.

Experiment 1 - Relative Planning Time

Spawn Time

Number of Agents	Minimal Horizon	Maximum Horizon	Agent Profiles
600	0.01977	0.08582	0.02270
1250	0,03833	0,09265	0,05481
1900	0,07007	0,13599	0,08349
2600	0,09180	0,16986	0,10473
3300	0,11572	0,22302	0,12922

Table 12: Time (in seconds) spent on spawning agents while running the simulation for 60 seconds using the three planning approaches.

Next Goal Time

Number of Agents	Minimal Horizon	Maximum Horizon	Agent Profiles
600	0,49267	0,47583	0,48584
1250	0,87695	0,84119	0,85388
1900	1,22070	1,23108	1,22784
2600	1,69848	1,64271	1,75922
3300	2,05079	2,02575	2,01947

Table 13: Time (in seconds) spent on determining the next (sub)goal for agents while running the simulation for 60 seconds using the three planning approaches.

Experiment 2 - Complex Planning Time

Minimum Horizon
Spawn Time

Number of Agents	Path: 100	Path: 200	Path: 500	Path: 1000
1	0	0	0	0
2	0	0	0	0
5	0	0	0	0
10	0	0	0	0
20	0	0,00049	0	0
50	0	0	0	0,00003
100	0	0	0	0,00012
200	0	0	0	0,00031
500	0	0,00049	0,00049	0,0004
1000	0,00049	0,00098	0,00049	0,00067
2000	0,00049	0,00049	0,00098	0,00153
5000	0,00244	0,00195	0,00342	0,00378
10000	0,00244	0,00146	0,00684	0,01074
20000	0,01953	0,00732	0,01562	
50000	0,01172	0,01514	0,03265	
100000	0,0293	0,04004		
200000	0,04639	0,07563		
500000	0,12988			
1000000	0,25977			

Table 14: Time (in seconds) spent on spawning agents the given number of agents for a fixed plan length.

Goal Selection Time

Number of Agents	Path: 100	Path: 200	Path: 500	Path: 1000
1	0,00049	0,00146	0,00752	0,03119
2	0,00049	0,00293	0,01546	0,06068
5	0,00195	0,00684	0,03906	0,15857
10	0,00391	0,01367	0,07739	0,31631
20	0,00732	0,02637	0,15869	0,62085
50	0,01758	0,06885	0,41187	1,58371
100	0,03809	0,13965	0,80664	3,18198
200	0,07275	0,27832	1,62744	6,37524
500	0,19287	0,68848	4,0293	15,85733
1000	0,39014	1,42236	8,0874	31,92914
2000	0,74365	2,79492	16,04639	63,72787
5000	1,89111	7,00244	40,1665	158,4364
10000	3,729	14,02246	80,58545	317,4677
20000	7,62354	27,99463	160,3359	
50000	18,82422	69,53467	402,392	
100000	37,46631	139,3491		
200000	75,33301	285,8457		
500000	187,839			
1000000	374,7959			

Table 15: Time (in seconds) spent on determining the next goal for the given number of agents for a fixed plan length.

Maximum Horizon
Spawn Time

Number of Agents	Path: 100	Path: 200	Path: 500	Path: 1000
1	0,00049	0,00146	0,00781	0,03271
2	0,00073	0,00244	0,01562	0,06201
5	0,00171	0,00635	0,04199	0,15771
10	0,00342	0,01367	0,07764	0,31641
20	0,00708	0,02588	0,15576	0,64795
50	0,01758	0,06641	0,40332	1,61523
100	0,03491	0,1333	0,81055	3,19593
200	0,07227	0,29028	1,62891	6,46484
500	0,19287	0,71753	4,05225	16,03223
1000	0,37646	1,42773	8,04932	32,02686
2000	0,75049	2,78003	16,09375	64,31348
5000	1,84277	6,90332	40,18262	160,561
10000	3,68066	13,85889	80,74072	322,2154
20000	7,41431	27,70117	160,8555	
50000	18,55029	69,37305	402,6683	
100000	36,9519	138,3691		
200000	73,8269	277,2556		
500000	184,074			
1000000	368,1316			

Table 16: Time (in seconds) spent on spawning agents the given number of agents for a fixed plan length.

Goal Selection Time

Number of Agents	Path: 100	Path: 200	Path: 500	Path: 1000
1	0	0	0	0
2	0	0	0	0,00098
5	0	0,00049	0	0,00195
10	0	0	0,00195	0,00293
20	0,00024	0,00098	0,00098	0,00488
50	0,00049	0,00122	0,00244	0,01465
100	0,00146	0,00269	0,00781	0,03096
200	0,00269	0,00659	0,01953	0,05566
500	0,00513	0,00781	0,04102	0,15918
1000	0,01172	0,02808	0,10205	0,31689
2000	0,01978	0,04541	0,18408	0,59619
5000	0,05762	0,14307	0,45752	1,5166
10000	0,1145	0,25415	0,95117	3,05319
20000	0,21973	0,53369	1,96387	
50000	0,54712	1,26758	4,87189	
100000	1,07642	2,53979		
200000	2,16406	5,10461		
500000	5,37988			
1000000	10,72339			

Table 17: Time (in seconds) spent on determining the next goal for the given number of agents for a fixed plan length.

Agent Profiles
Spawn Time

Number of Agents	Path: 100	Path: 200	Path: 500	Path: 1000
1	0	0	0	0,00001
2	0	0	0	0,00001
5	0	0	0	0,00002
10	0	0	0	0,00002
20	0	0,00024	0,00024	0,00006
50	0,00049	0,00024	0,00024	0,00009
100	0,00049	0,00024	0,00073	0,00015
200	0	0,00073	0,00073	0,00044
500	0,00122	0,01196	0,00122	0,00117
1000	0,00244	0,00171	0,0022	0,00262
2000	0,0166	0,01538	0,01489	0,0166
5000	0,04541	0,05005	0,04761	0,03821
10000	0,08716	0,09009	0,10571	0,11057
20000	0,18848	0,18286	0,18335	0,19305
50000	0,48486	0,46289	0,4646	0,4935
100000	0,93652	0,92334	0,96289	1,00119
200000	1,87646	1,84961	1,91907	1,97287
500000	4,75684	4,75098	4,73352	4,95941
1000000	9,53906	9,40967	9,297	9,76593

Table 18: Time (in seconds) spent on spawning agents the given number of agents for a fixed plan length.

Goal Selection Time

Number of Agents	Path: 100	Path: 200	Path: 500	Path: 1000
1	0,00024	0,00049	0,00049	0,0005
2	0,00073	0,00049	0,00073	0,00061
5	0,00146	0,00146	0,00146	0,00145
10	0,00293	0,00293	0,00293	0,00296
20	0,0061	0,00586	0,00586	0,00589
50	0,01489	0,01562	0,01514	0,01508
100	0,03052	0,03003	0,03442	0,02991
200	0,06177	0,06006	0,06128	0,06013
500	0,15259	0,1543	0,1543	0,15036
1000	0,30518	0,31152	0,31226	0,31577
2000	0,61792	0,61597	0,61646	0,61475
5000	1,5564	1,53076	1,53491	1,54138
10000	3,08032	3,07153	3,09448	3,06894
20000	6,17017	6,18164	6,18506	6,1651
50000	15,41553	15,40967	15,43652	15,39822
100000	30,73584	30,90186	30,81042	30,76254
200000	61,55786	61,63086	61,5765	61,45029
500000	154,1125	153,9253	153,7647	153,8959
1000000	308,1072	307,9922	307,4349	307,733

Table 19: Time (in seconds) spent on determining the next goal for the given number of agents for a fixed plan length.