AI for Autonomous Agents in Games and Simulations

Student Name: James King

Supervisor Name: Magnus Bordewich

Submitted as part of the degree of BSc Computer Science to the

Board of Examiners in the School of Engineering and Computing Sciences, Durham University

April 30, 2014

Abstract — **Context:** Designing the core autonomous behaviour for characters in a video game, where those characters are expected by the player to perform tasks for themselves, poses many interesting challenges. This paper explores the tasks faced when developing a capable and somewhat convincing set of Artificial Intelligence routines within a simulated zombie epidemic as the foundation of a real-time strategy game with player allocated goals. **Aims:** The primary goal for this project was to develop a successful method of controlling a large number of agents in a hazardous environment to prioritise survival through cooperation. The algorithms must not be computationally expensive to allow for many agents to act in real-time and also express superficially convincing human-like behaviours. **Method:** Two conceptually distinct architectures for character behaviour were implemented, a subsumption stack and Beliefs-Desires-Intentions model. These were then compared in terms of the ability for agents to avoid threats, system resource usage and qualitative aspects of their behaviour. Advanced path finding and vision testing algorithms were also implemented. **Results:** A total of 480 individual simulations were performed to generate graphs describing population decline and execution speed for the subsumptive implementation and two versions of the BDI approach, which seemed to favour the BDI model for this scenario in all aspects. **Conclusions:** For this problem domain the BDI model appears to be a better fit due to its efficiency and power while also allowing for easy extensibility. A few avenues for additional research in the area of shared task planning are also identified.

Keywords — AI, BDI, cooperative, multi-agent, subsumption, task planning, video game

I INTRODUCTION

A Context

Video game worlds are often populated with computer controlled Non-Player Characters (or NPCS), the behaviour of which can make or break a game. A great deal of effort is required when designing the routines used by these characters if they are to satisfy the following major requirements. They should behave with approximate rationality, provide an appropriate challenge for the player and act in a believable way if the characters represent humans (or at least animals). For Real-Time Strategy (or RTS) games the difficulty is further accentuated by the necessity for the game to support perhaps hundreds or even thousands of these characters in an environment simultaneously. This combines the previous three requirements with a fourth: performance. Designing solutions for the first three key requirements is often more of a creative task than a purely methodical one and entails an element of subjectivity. The fourth requirement is easier to test, but the difficulty in

maintaining a minimum performance quality heavily depends on the complexity of the solutions for the first three goals.

B Problem Domain

This project aims to explore possible implementations of an NPC behaviour system designed to realise the central gameplay of a work-in-progress RTS game. The components of the game that existed prior to this project form a zombie epidemic simulation within a procedurally generated city environment containing buildings separated by streets constructed on a tile-based grid. Initially, a number of human characters are dispersed around the environment, a fraction of which are allocated the role of being a zombie. The simulation then begins, with the zombies navigating towards the nearest visible human and humans attempting to run away from danger if any is present (and otherwise just wandering randomly). If a zombie catches up to a human it may attack it, reducing a stored health value for that human and infecting that human with some probability. A human will become a zombie if they are infected when their health value reaches zero, while a zombie or non-infected human is simply removed from the simulation upon losing all their health.

The conflict is clearly one-sided, not least because the zombies are smarter than the humans. The humans have little regard for their surroundings other than the locations of visible enemies and so often run directly into the inner corners of walls. This would obviously not be classed as a sufficient behaviour as it does not meet the core requirements for a decent NPC. It fails at being rational, at providing an appropriate challenge (the humans don't stand a chance) and the agents certainly don't act like humans. Their only redeeming quality is that their simple AI isn't too computationally expensive, so thousands of them can be supported simultaneously.

C Project Aims

At the very least the artificial intelligence routines explored in this project should improve each human agent's survival ability. This may mean attempting to escape when cornered, deciding when it is rational to attack in self defence and implementing strategies to hide from danger. Later on in the project a player will be able to assign actions for the humans to complete, such as instructing a group to navigate to a specific location, or to construct barricades out of material found in buildings. These commands may conflict with an agent's necessity for self preservation, so the processes developed should intelligently decide when it is rational to neglect an order in favour of reacting to danger.

On the topic of player-specified tasks, an efficient path finding technique will need to be implemented that balances computation time with optimality of the path found. This algorithm will be essential for when agents are directed to a specified location by the player, but also useful for attempting to navigate when performing other tasks or to detect if a path exists to bypass some barricades (thereby establishing whether they are secure). Some tasks may not be assigned to a specific agent but will rather be goals to be achieved by the collective group. For example, the player will be able to instruct that a barricade should be built in a specific location. In this instance the human agents should automatically distribute sub-tasks between themselves in order to achieve the common goal efficiently.

The two core approaches to be compared are a Subsumption architecture (Brooks 1990) and a Belief-Desires-Intentions model (Rao & Georgeff 1995). A subsumption architecture features a stack of behaviour layers where each may in turn choose to either act or subsume control to the next layer, starting with the top of the stack. The first layer in the sequence to specify an action is heeded and the rest are ignored. This design usually relies on complex behaviour emerging from complimentary layers. A Belief-Desires-Intentions model is a radically different approach, where each agent records an internal representation of the world from which a set of attainable goals are found, a subset of which are committed to in an attempt to maximise expected utility.

D Deliverables

These following core elements will be created before the end of the project.

1. **Expanded Core Game** — The main environmental components that the AI system will interact with must be implemented. These include the creation of game objects that represent a resource to be used when building barricades, the system to support the construction of the barricades themselves and an expanded user interface which allows a player to select groups of agents and assign tasks. Additionally facilities to be used by the two proposed architectures should be provided, such as a path-finding implementation.

2. **Subsumption Prototype** — An agent design using a subsumption architecture will be developed, implementing the core requirements of the human NPCs. This prototype will be constructed by breaking up the humans functionality into a stack of behaviours, from which a successful strategy is designed to emerge. Relies on deliverable 1.

3. **BDI Prototype** — A distinct agent design using a BDI architecture will also be developed, meeting the same requirements as the subsumption approach. This prototype will use a personal abstracted representation of the environment for each human agent, from which a set of achievable goals are found using a goal evaluation algorithm. These goals are then further reduced into a set that is calculated to yield the most utility and are simultaneously possible. Relies on deliverable 1.

4. **Analysis & Final Report** — Both completed prototypes will be analysed and compared in terms of agent survival rate (without player intervention), efficiency at which goals specified by a player are achieved, prevalence of unnatural behaviour and system resource usage. The results of this comparison will be compiled into a report, along with the identification of the prototype that is most suited for the problem and an explanation for why that prototype was chosen. The report will also cover possible areas of additional research exposed by this project. Relies on deliverables 2 and 3.

II RELATED WORK

The problem of finding a shortest path between two positions in a virtual world often rears its head when dealing with video game agents. Solving this problem can be costly

when applied to complex environments or when using many agents if an optimal result is desired. The goal is to emulate the human ability to plan a route through a world (or an abstraction of one), usually with complete knowledge of the environment to be navigated. It is far more difficult to find an optimal path if aspects of the world are hidden from an agent while it is planning a path, so a trade-off can be made to only require a path that is *rational*. For games, a laxer requirement for the quality of path may even be beneficial, as the resulting path-finding agents appear more natural and human like, making occasional wrong turns along the way.

Computer games are real-time applications; they should respond to the actions of the player within a few milliseconds. The minimum requirement is to simulate and update the world between each consecutive pair of frames rendered and outputted to the screen. Conventional path finding algorithms such as A* will produce a complete and optimal path from the agent's position to the target position, a result that is quite costly and often unnecessary. *Real-time Heuristic Search for Pathfinding in Video Games*(Hart et al. 1968) describes and compares several algorithms that are designed to have a low time and memory footprint, and so are ideal for use in video games. The three algorithms analysed in detail are k Nearest Neighbours Learning Real-time A* (kNN LRTA*), Time-Bounded A* (TBA*), and Real-time Iterative-deepening Best-first Search (RIBS).

The first algorithm, kNN LRTA*, relies on a precomputed database of paths between *subgoals*. This allows the bulk of path calculation to be offset to when the game world is created, and the time needed to find a path for each agent is significantly reduced. A decent heuristic for deciding which nodes to precompute paths between would be *bottlenecks* in the game world, nodes that are more likely to be on paths between disparate regions of the world. For the on-line searching procedure, the main challenge is selecting a series of subgoals that would lead to the least deviation from the shortest path. In terms of optimality, the algorithm will trivially produce an optimal result if the starting and goal nodes are both subgoals with precomputed paths (assuming the precomputed paths are optimal - which they may well be depending on the algorithm used to find them). The article's performance analysis of the algorithm demonstrates the mean move time and sub-optimality of the algorithm as the size of the precomputed database increases. As would be expected, a larger database produces significantly more optimal results, although it appears to plateau at around 12 percent longer than optimal. A sub-optimality of this order is certainly sufficient for use in this project. The algorithm compares very favourably in both respects to the second algorithm, TBA*, and also in working memory used, but at a cost; a significant pre-computation time required, in the order of several days for database sizes that produce even acceptable results. This could be a major issue if the algorithm were to be used for this project, because game worlds should be generated and therefore paths precomputed whenever a player wishes to begin a new game. Usually this doesn't pose a problem for games with static worlds, as the subgoal database may be precomputed by the developer and the results distributed with the game. The problem may be less pronounced when applied to this project as the pathing node count will be significantly smaller than the worlds used in their study, but the precomputation time would still add an unwelcome delay to the already lengthy world generation process.

The next algorithm analysed in the article is TBA*, a simple modification of the ubiquitous A* algorithm that simply limits the execution time to a fixed threshold. If the

algorithm hasn't halted when the threshold is exceeded (a complete path has not been found), the immediate action that appears most promising is chosen. Another slight alteration is that the working data used by A* isn't discarded when an action is taken, and persists to the next simulation iteration to allow the agent to continue where it left off. For applications where there is adequate time to complete a full A* search every simulation step, this algorithm is clearly optimal. The time bounded nature of the algorithm essentially enforces a contract that the algorithm has constant worst-case time complexity, and so is suitable for real-time applications. However, the algorithm can occasionally reconsider which path to take and will produce movements that appear indecisive. This algorithm would certainly seem adequate for this project, and will be investigated further.

Finally, the article describes the RIBS algorithm. This method attempts to be more agent-oriented than the previous algorithm in that TBA* will often explore nodes at locations in the map away from the agent it seeks to find a path for. In contrast, RIBS will focus on the agent's local environment and so will be able to respond to nearby changes and will involve less random access of remote nodes; beneficial in the case of expensive memory access. This algorithm is much more involved than TBA*, and may only provide limited improvements in efficiency. The technique is also relatively untested and so may have unforeseen drawbacks.

The second article to be analysed is titled *Partial Pathfinding Using Map Abstraction and Refinement*(Sturtevant & Buro 2005), and details the Partial Refinement A* (PRA*) algorithm. This is another real-time path planning algorithm that relies on a tiered structure of the environment at different abstraction *levels*, and a heuristic function that can estimate the distance between two nodes at any abstraction level. The range of levels in this hierarchy spans from the original graph of path nodes in the world at the lowest level, to an unconnected graph with a node for each island of nodes from previous tiers with existing paths between them. Each tier is constructed such that within each abstracted node is a clique of nodes from the previous tier (although orphans are additionally attached to the group they connect to). A useful consequence of the construction of this hierarchy is the efficient ability to test if a path exists between any two nodes; simply check if they both belong to the same highest-level abstracted group. This can save a great deal of time because searching for a non-existent path can lead to a fruitless search through every node in the world. PRA* expands upon a simpler algorithm *QuickPath*, which recursively finds and refines paths through each abstraction level from the highest to lowest, arriving at a complete path from the start to the goal node at the lowest abstraction level.

The article points out that while *QuickPath* is guaranteed to find a path if it exists, there is little effort made to refine the path in an optimal manner and the result may be far longer than the shortest. The PRA* algorithm aims to improve the quality of the outputted path through several means. Firstly, the algorithm starts at a lower abstraction tier than the top layer, using A* to find a path of abstracted nodes. This abstracted path may then be refined recursively with subsequent A* calls. The method can be applied in an agent-centric way like RIBS, by only refining the path to the lowest abstraction near to the agent, saving computation time and allowing for a dynamic environment. This partial refinement can be compared to TBA* and RIBS from the previous article in that these algorithms don't attempt to produce a complete path from the agent to its destination, and simply do the best they can with the time they are allocated. However, PRA* should be

less inclined to stumble into dead-ends than TBA*. The algorithm also suits this particular application well, as the world is already abstracted into groups of nodes in the form of the blocks containing each building; an abstracted path may be found at the block abstraction level and then partially refined for each block when it is visited by the path finding agent.

The third article, *Real-Time Path Planning in Heterogeneous Environments*(Jaklin et al. 2013), explores path finding for agents with a preference for different classes of region in the environment. While not directly contributing to solving the problem of efficient real-time path finding for many agents, it would provide a basis for expanding the project if there is time available to make humans prefer to walk on pavements rather than roads, or as a means of selecting paths that avoid areas with a high zombie density.

For the second major issue of cooperative agent role allocation, identifying the specific class of problem would be an important step in finding a solution. *Multi-Agent Role Allocation: Issues, Approaches, and Multiple Perspectives*(Campbell & Wu 2011) provides an excellent description of the various classifications of role allocation problem. From this resource the following properties of this specific problem have been identified; it can be modelled as an *iterative Optimal Allocation Problem* (OAP), can be divided into *individualistic* tasks that require only one agent, communication between agents is *free*, and the roles are *explicitly defined*.

The issue can be classed as an instance of the iterative OAP because it satisfies the condition of each agent only performing one role at a time, and it is possible for a function to be formulated that provides a value representing the utility of an agent performing a given task. As the article describes, an optimal solution to the problem involves finding a set of agent-role pairings that maximises utility. For this application, a decent utility function would largely map a higher value to agents with a short path length to the location of the task. The presence of danger between the agent and the task location should also be taken into account to avoid agents assuming roles that could risk their safety.

The tasks are classed as being individualistic as only a single agent is required to complete them, and in this particular scenario no more than one agent can perform one of the tasks (two agents can't carry the same piece of barricade material to a stockpile or to a barricade under construction). Additionally, each task cannot be divided into subtasks that different agents can perform - the entirety of the task is to carry an item from one location to another.

Communication between agents inflicts no cost to either sender or receiver in the context of the simulation (processing time is assumed to be negligible), a property which fortunately simplifies the allocation problem as the volume of information transferred between agents doesn't need to be taken into account when allocating roles, although it should still be kept to a minimum for performance reasons. Communication in this context will be needed to ensure two agents don't claim the same role, and to allow agents to negotiate swapping roles when circumstances call for it.

Each role is explicitly defined; the actions required to complete the related task is independent from which agent assumes the role. This property, combined with freedom of communication, presents several possible solutions to the role selection problem as described in the article; *Negotiation-based*, *Threshold-based*, *Broadcast and Compute*, and *Token-based*.

Negotiation-based strategies typically involve a group of agents performing market-

like activities such as bidding for roles in order to construct an allocation set. This usually involves a coordinating agent to act as a *manager* to decide which agent wins a bid (the bids being related to the utility of an agent assuming that role), although distributed methods exist but have been found to be less effective. The article concludes that a negotiation based technique may be overly complex for simple problems with a large number of roles, qualities that describe this particular application and so other strategies will be considered.

The next method described in the article is the Threshold-based variety. This method requires that each agent independently keep track of a value for each available task, which will become prohibitive when many tasks are available with many agents able to claim them.

Broadcast and compute follows, which involves each agent broadcasting its current state and then selecting a role based on its own state and the received states of other agents applicable for that role. Perhaps counter intuitively, this method can result in *less* communication for situations where communication is costly because each agent will only need to communicate a small number of times per allocation. However, the method relies on each agent being aware of the environments of other agents, but this application is simulated to be partially observable and so this is not always possible.

Finally, Token-based role selection for roles with the individuality property involves each agent possessing at most one *token*, where each token maps directly to a role in a 1-to-1 relationship. Agents may trade tokens to swap roles if they deem it to be preferable, and through many such trades an optimal role allocation may be found. The method enforces the requirement that each role must be possessed by at most one agent, which happens to be a requirement of this application. This technique also requires no central processing and little communication. A Token-based method appears to be a good fit for this project, although care must be taken to ensure that the set of active tokens is maintained such that each role has exactly one available token at all times.

III SOLUTION

i Development Environment

The original game that this project is extending is written in C#, so that will be the primary programming language used. Microsoft's Visual Studio 2013 will be used as an integrated development environment for its C# compiler, debugging and profiling tools and Sublime Text 2 for any peripheral text editing. Development will mainly be on a Windows desktop PC (Intel Core i5, AMD 7850), using a Windows laptop (Intel Core i7, nVidia GeForce GT 540M) to test performance on a device with an nVidia GPU. The OpenTK library is used to expose OpenGL bindings for .NET and is the only non-standard library used.

ii High Level Architecture

Currently, after the environment has been generated a main control loop begins. During each iteration this loop will either perform a simulation step, redraw the screen, or both; depending on how much time has passed since those respective actions were last performed. While screen redrawing will occur as frequently as possible while maintaining a minimum period of 16.67ms, simulation steps attempt to maintain an average period of that same

frame-time. This may involve running slightly more simulation steps than the desired average to make up for a prior period of slow updates that fell below the average. Each simulation step cycles through every object in the game environment, including agents, and allows them to perform some individual logic. Following this, player inputs are interpreted and some additional processing will be performed as described in sections iii and vi.

iii Path Finding & Navigation

The problem of finding efficient paths through virtual environments is often faced when developing AI for games and so has been explored extensively. Generally the A* algorithm (Hart et al. 1968) or one of the many variants is used, although while it guarantees a high quality path it may require a significant amount of time to find it (depending on the complexity of the environment). When observing the structure of the worlds produced by the procedural city generation algorithm (See Figure 1) it is easy to spot aspects that could potentially be exploited to produce an A* adaptation which required less time to execute.

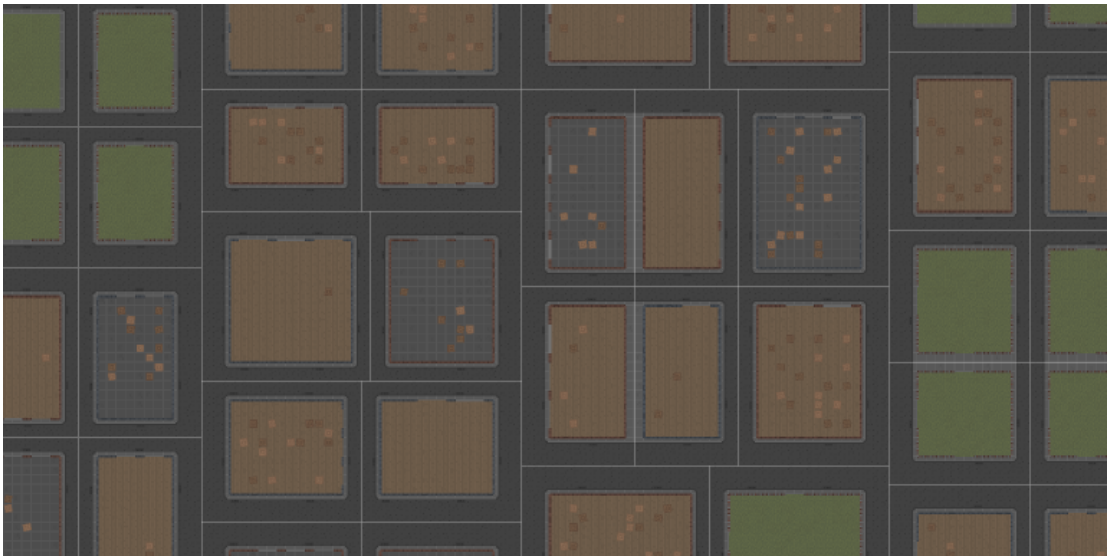

Figure 1: A diagram to show the structure of a procedurally generated city environment, featuring blocks of buildings separated by roads. Each block is bounded by a white rectangle.

When imagining how we would solve the problem of navigating to a remote position in a city in the real world, we don't think on a metre-by-metre basis. We plan our overall route street-by-street and only worry about how to navigate down each road when we reach it. A similar approach can be implemented here, where an agent first applies the A* algorithm on a graph representing the road network of the city and then applies a more refined A* instance along each street when it reaches it as described by Sturtevant & Buro (2005). A refined path is also found from the agent's initial position to the nearest street and then from the final street to the destination. For cases where the street-based route takes a corner, the refined A* instance should look for paths between the start of the street preceding corner to the end of the street following it, in case the block that they share a border with can be cut across.

One of the requirements of the system was for it to maintain a minimal amount of processing time per simulation step. This may still be exceeded if many agents calculate refined paths during the same step, so a queueing system should be implemented to limit the number of paths determined per iteration. Each agent that wishes to find a refined path will submit a request to the back of a shared queue while removing any of their previous requests. Once per simulation step this queue will be processed, dequeuing and servicing requests until some time limit is exceeded. This process should occur after all individual agent calculations are complete, so that this time limit can accommodate those agent actions. This may mean that several simulation steps will be required for a large group of agents to receive paths, but distributing this processing time over several frames is obviously preferable to the game pausing for a visible amount of time. As an additional positive side effect, seeing a crowd of instructed agents begin to move at different times over a few frames would appear more natural than them all moving simultaneously.

This algorithm should produce results very close in length as to when applying a single instance of refined A* across the entire route, with a far shorter execution time for all but the most trivial instances. If the implementation still leads to a performance bottleneck then it could potentially be improved by caching the paths found along individual streets, which can then be looked up instead of calculating the same paths multiple times. This will be particularly beneficial when a large group of nearby agents are travelling to the same destination.

iv Vision & Collision Testing

Visibility testing and collision detection are similar problems and can be largely solved with the same algorithm. For collision detection the program is testing if an object (simplified as its bounding rectangle) can be 'swept' along a vector representing the movement it wishes to make during the current simulation step. If the swept rectangle intersects with solid world geometry, the object's movement is pulled back to the furthest it could go without such an intersection. Similarly, when testing the visibility of an object you are essentially testing whether it can be swept towards the observer without being blocked by any solid geometry, thereby detecting if its image would be blocked by a wall.

This sweeping action is the costly part of the technique, an optimal solution will limit the number of intersections tested with this swept area to the absolute smallest number required. As in this scenario only walls can intersect the object, and walls are only present along the edges of tile, the distance along the movement vector that the object is swept can be incremented by amounts that exactly move it to the next tile boundary that would be met. This method is illustrated by Figure 2 and is an adaptation of the technique used in early 3D environment rendering (Permadi 1996).

This technique compares well against the usual naïve approach of sweeping a certain fixed distance along the movement vector each iteration. It is guaranteed to find an intersection if one exists and calculate the exact distance the object would have to travel to reach it rather than just the previous iteration location when using a constant step distance. It also requires fewer iterations, assuming a constant step distance approach uses a small iteration delta (which it would need to for accurate results).

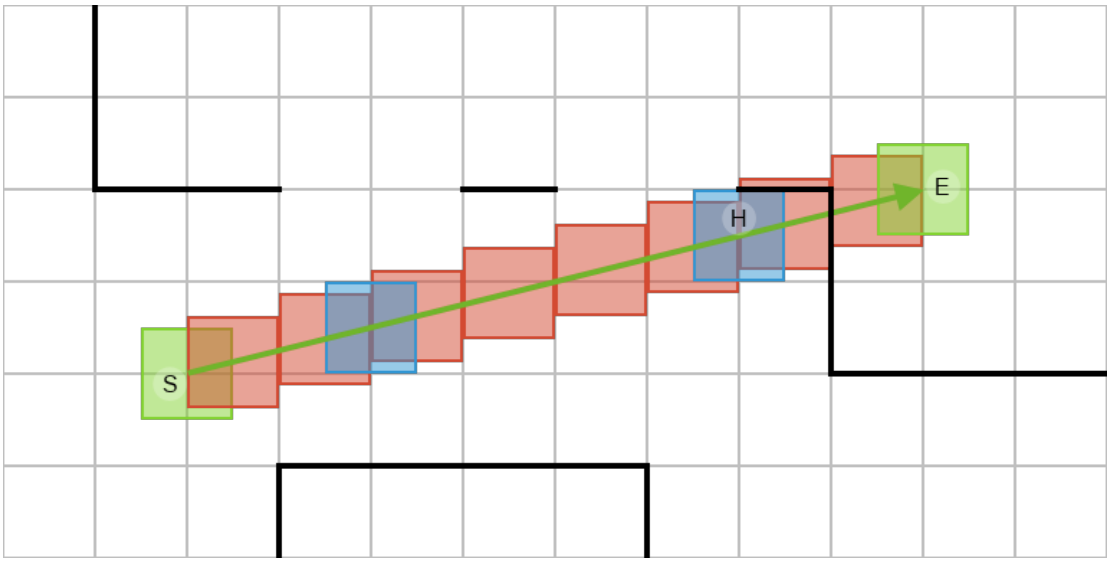

Figure 2: A diagram to show the world geometry intersection checks performed when sweeping an object's bounding rectangle along a trace line. The object starts at position S and its destination is position E. Black lines represent solid walls. The red boxes show iteration steps where a horizontal collision test is performed against possible obstructions to the right and the blue boxes are iteration steps where vertical collision tests are performed upwards. Position H is the projected box location where a collision is first detected and so will be the position the object is moved to.

v Subsumption Prototype

Central to a subsumption agent architecture is a stack structure upon which behaviour layers can be pushed. This is relatively simple to implement using an Object Oriented approach and an adaptation of the design used by Butler et al. (2001) will be used involving an abstracted behavioural layer class with a method to be overridden called when that layer should decide to act. This method will return $true$ if an action is performed by that layer, otherwise $false$. The highest layer in the stack will be invoked first and if that layer returned $false$ the next layer will be invoked. This continues until either a layer returns $true$, or the last layer has been invoked. The proposed layers to be implemented are listed below, with the top-most layer first.

- **DropResource** — Drop a carried resource if in danger or at a barricade under construction
- **SelfDefence** — Attack a pursuer if they are too close
- **Mob** — If there are many more humans than zombies nearby, head towards a zombie
- **VacateBlock** — If carrying a resource or in danger while inside a building, run outside
- **Flee** — Run away from nearby danger
- **FollowRoute** — Follow a route assigned to this agent
- **PickupResource** — If standing on a resource and not in danger, pick it up
- **FindResource** — If not in danger and not holding anything, move towards a resource

- **HarvestResource** — Attack a nearby object that yields resources when broken
- **SeekRefuge** — When outside and neighbouring a safe-looking building, run inside it
- **Wander** — Walk around randomly

This particular configuration of behaviours should satisfy each requirement of the AI prototype, using complimentary sets of layers to allow for more complex compound behaviours to emerge. The survival strategy is implemented with the $SelfDefence$, Mob, $VacateBlock$, $Flee$ and $SeekRefuge$ layers, $FollowRoute$ for travelling to player designated locations and the remainder (along with $VacateBlock$ and $SeekRefuge$ again) for constructing barricades autonomously.

The survival aspect functions with several emergent sub-strategies. The $SelfDefence$ behaviour will attack a zombie that is extremely close to the agent, which is expected to occur when the zombie catches up to the agent due to a movement speed advantage or if the agent has been chased into a corner. In either case, attacking the pursuer is the only rational thing left to do. The Mob behaviour actively directs an agent towards a nearby zombie if that agent predicts that a decent number of other agents will perform the same action, enough to overwhelm the zombie. When the agents are close enough, the $SelfDefence$ behaviour will take control and cause the agents to attack the surrounded enemy. This strategy should lead to zombies being eliminated in safe situations where it is unlikely for a human agent to take much damage. $VacateBlock$ and $Flee$ will help avoid situations where being in close proximity to a zombie is unnecessary and $SeekRefuge$ ensures agents aren't exposed outside.

Barricade construction relies on a similar interaction between several behaviours. The process usually starts with an agent wandering around outside, carrying no resource object. The $SeekRefuge$ behaviour will lead it to navigate inside a building, within which it may find objects that yield resource items when damaged when obeying the $HarvestResource$ behaviour. When the object breaks into its component resources, the $PickupResource$ behaviour is followed and the agent now holds a resource item. The $VacateBlock$ layer instructs the agent to leave the building if it is holding a resource and as soon as the agent leaves it drops the item as instructed by the $DropResource$ layer. The agent then attempts to $SeekRefuge$ again and the cycle repeats. Resource items are continually dropped onto tiles neighbouring an exit to the building, which pile up to form barricades. When the barricade becomes large enough, the tile it belongs to becomes solid and the agent automatically finds a different path to leave and re-enter the building, thereby dropping items onto a different tile.

This solution is quite trivial to implement due to the simplicity of each individual layer. Some behaviours, such as Mob and $Flee$, provide some scope for experimentation by testing different thresholds for when to perform that action. Additionally, it may be interesting to test whether agents survive longer if they all posses the Mob layer, or if having a subset of agents that deviate from that strategy will lead to that set lasting markedly longer than the longest surviving agents in the first instance.

vi BDI Prototype

For a BDI agent architecture we must design each of the three core components (Beliefs model, Desires identification and Intentions filtering) as individual systems (Rao & Georgeff 1995). After this, for each type of intention a planning mechanism must be implemented.

The Beliefs component involves designing a system for representing aspects of an agent's environment that could influence its actions, updating that internal representation as the agent perceives stimuli. The most direct structure for this model of the environment (while guaranteeing that potentially useful information is stored) would be to store every detail exactly as it is

perceived; but this would be difficult to parse, would hold unnecessary detail and could result in too much memory being consumed. A better approach would be an abstracted model, perhaps considering two levels of abstraction as with the path-finding solution (see section iii). The highest level abstraction will be at a city block scale, considering observed attributes of the contents of each block. Each agent will track a few key properties about each block; the last time they observed it, whether that block contains a building, the number of human agents seen within that block, the number of zombie agents and the number of barricade resources seen. Each agent will also track the locations of observed agents and resources for their local environment, information that will be discarded when they travel a specified distance from those locations to save memory. Agents will also record whether they are within a building or outside, their health value and whether they are holding an item.

Desire identification is the next component requiring planning, which involves designing a mechanism to parse the beliefs model to find a set of achievable goals that align with the higher aspirations of the agent. For this specific scenario, these aspirations will be to improve safety and achieve player-given tasks. These high level goals can be broken down into sub-goals as described by Figure 3.

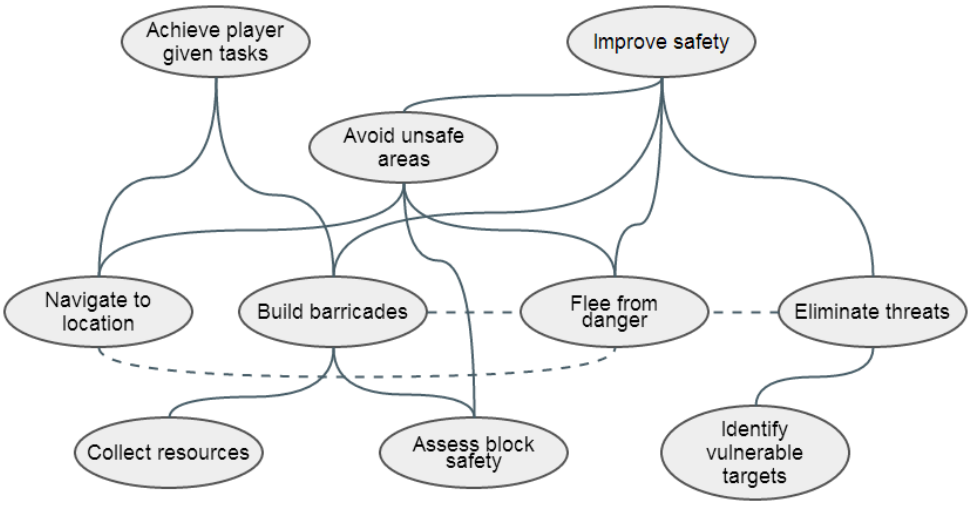

Figure 3: A diagram to show a breakdown of the main goals for a human agent into sub-goals, with possible conflicts signified by dashed lines.

The *Avoid unsafe areas* sub-goal obviously relies on being able to identify whether an area is potentially hazardous and more importantly whether there is another location worth travelling to due to increased expected safety. Implementing this can exploit the abstracted block-level representation of the world as defined in the Beliefs model by calculating a *danger* heuristic for each block. This heuristic would be a function of the remembered number of zombies, humans, barricade resources and taking into account uncertainty related to the amount of time since the block was last observed. The details for specifically how these values are manipulated into forming the heuristic will be a point for experimentation, as it will be difficult to discover the optimal way to factor in uncertainty without testing. A second heuristic should then be calculated for each block which finds the estimated cost of travelling there, determined with an A* instance that uses the values from the previous heuristic as edge weights for path segments travelling through each block. If any blocks are deemed to be safer than the one the agent currently resides in (using the first heuristic) and is also found to be worth the trip (using the second heuristic), the agent will desire to travel to it.

For the *Eliminate threats* sub-goal, agents need to rely on teamwork to safely and efficiently attack zombies. A one-on-one fight will lead to the human receiving a considerable amount of damage, whereas if a large group of humans attack an individual target the fight will be resolved quickly before much damage is dealt by the zombie. For this cooperative strategy to work, agents need to determine if enough other agents will be willing to pitch in for the battle to be worth undertaking. This can be achieved by each agent independently being empathetic towards other agents within a certain range of the prospective target, asking "if I was that agent, would I attack the target assuming all other nearby agents attacked too?" If it is estimated that a sufficient number would cooperate, a desire to attack the target would be included with the desire set. As a fail-safe for when guessing the intentions of other agents fails, if an agent finds itself considerably closer to a zombie than other agents it will forfeit the desire to attack it in favour of desiring to flee.

Forming desires related to the construction of barricades relies on there being a known local position requiring one, either through finding a constricted space that would require a small barricade to produce a large enclosed area, or identifying a location that has been requested by a player to have a barricade. The former situation has a trivial solution in that the entrances to buildings are easy to recognise for an agent, but an ideal solution would also eventually construct barricades across streets to enclose several safe city blocks as a human stronghold. This can be implemented through incrementally expanding existing enclosures; initially a building will be be barricaded across its entrance, then barricades are constructed across the ends of the street that the entrance looks upon (with the original barricade then being de-constructed) and then the barricade extends outwards to claim more streets that were previously exposed (see Figure 4). When the location for a barricade has been identified by an agent, it will desire to construct it. The barricades will be left on the cusp of completion, allowing for agents to travel through it, until a threat is detected near to the compromised section.

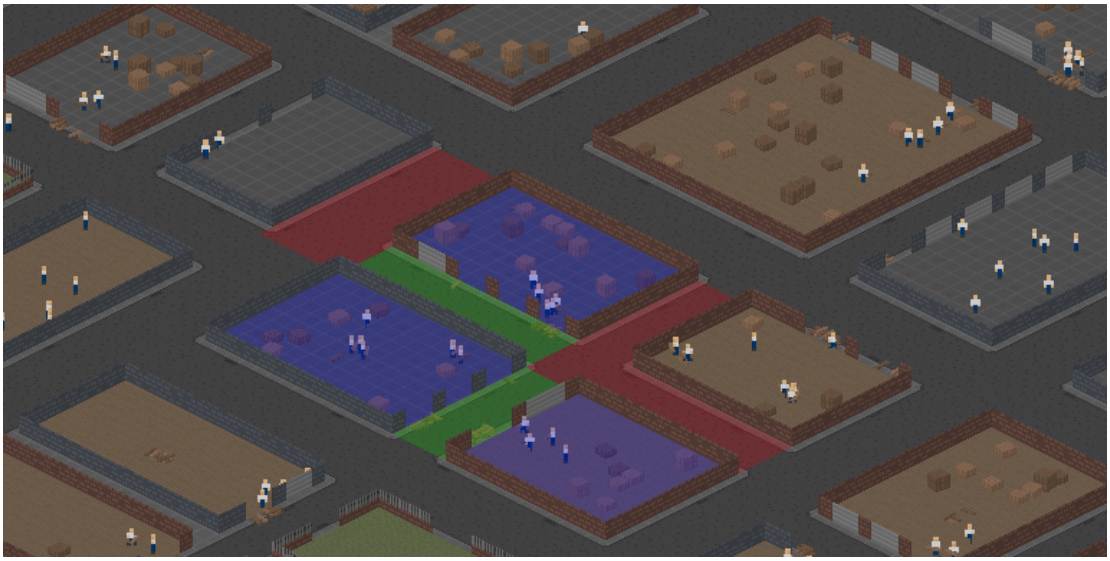

Figure 4: A diagram to show the proposed automated barricade expansion process. Barricades enclosing single buildings (blue) will initially be produced, followed by an expansion of the enclosure onto the neighbouring street (green) and then progressively outwards to adjacent streets (red). The process will continue in this manner while resources are available and the encroached area houses no threats.

The intention filtering process attempts to reduce the set of desires into a subset that max-

imises utility while leaving no conflicting goals (such as desiring to attack a target while also desiring to run away), from which agent actions are then derived. Filtering should prioritise agent survival over task completion and also prefer to travel to a location expected to yield more utility over remaining to build a barricade. Desires to avoid individual threats can be aggregated into a single desire to avoid many threats simultaneously; for example by desiring to leave a building within which are multiple hazards, or treating each enemy as a point charge repelling the agent and desiring to move in the direction of the resulting force vector. This process is expected to be the most performance intensive component of the BDI architecture, so a request queuing system like the one used for refined path calculation should be implemented. Intentions are mostly committed to by an agent for several frames at a time (Bonura et al. 2009), meaning determining them every simulation step for every agent will be extremely wasteful. Instead, a calculation to determine whether an intention is likely to have changed is expected to be faster to execute and would trigger a request being sent to a shared intention recalculation queue. This queue would be processed once per frame (as will the path calculation process) after all agents have acted, and would dequeue and perform the requested intention determinations until a time limit is exceeded. Obviously it would be more rational for every agent if they could recalculate their intentions immediately after they are believed to have changed, but the simulation step time limit is imperative. Helpfully the appearance of agents within a group "changing their minds" at different times from the same stimuli would give a more natural appearance to their behaviour, rather than them all reacting in the same instant.

vii Analysis Strategy

A comprehensive and fair analytical method is required for comparing the two agent architectures and this section will cover a set of experiments that satisfy both of these qualities.

Survival Rate will be tested by first selecting a set of initial simulation states (at least 10) and running the simulation using each once per agent architecture. Each simulation will be left to run for 5 minutes and the number of survivors and zombies at each moment in time will be logged. The data produced will be used to produce graphs of the agent counts over time which can be analysed in terms of population decline rates, time until a stable population is established and whichever other statistical features that may appear. This test will involve no player interaction, only the purely autonomous aspects of the agent designs.

Task Completion will analyse the efficiency at which each agent design can complete tasks assigned by a player. Similarly to the survival rate test, a set of initial states will be produced over which both architectures will run, the difference being that each initial state will include an instruction to build a collection of barricades at specific locations. The simulation will execute until each barricade is completed, the progress towards completion being recorded over time. This data will be graphed and compared, as with the survival rate data.

Performance of the two architectures is of obvious importance; a solution with optimal survival rate and task completion efficiency has little use if it can only support a handful of agents at a time without reducing the simulation to an agonisingly slow crawl. The main method for the survival rate test can be used here, but with frame-time (the time taken to calculate each simulation step) being recorded rather than survival rate. The main requirement for the architectures is to keep average frame-time below 16.67ms, with minimal dips above that threshold. This test is platform dependant and so will be carried out on at least two different machines. Additional more superficial tests will also be performed with world sizes spanning from 64x64 tiles to 256x256 tiles and a range of agent counts from 50 to 1000 with each world size, to analyse performance relating to dense vs sparse populations.

IV RESULTS

Quantitative results were produced by running the subsumptive architecture and two versions of the BDI implementation (one with frequent belief and deliberation updates, and the other with one quarter of the frequency) on a total of 160 different initial world states. These were divided into 8 different categories of world size and initial agent populations, designed to test different population densities, total agent counts, and ratios of survivor agents to antagonistic ones. For each of these categories 20 seeds were used to control the specific geometry generated and placement of agents, giving the total of 160 distinct environments. Table 1 lists each of the categories used to produce the final set of results.

ID	Size	Agents : Threats	Agent Ratio	Agent Density
A	64	48 : 16	3 : 1	1.172
B	64	96 : 32	3 : 1	2.344
C	64	120 : 8	15 : 1	2.930
D	128	96 : 32	3 : 1	0.586
E	128	120 : 8	15 : 1	0.732
F	128	128 : 128	1 : 1	0.781
G	128	192 : 64	3 : 1	1.172
H	128	224 : 32	7 : 1	1.367

Table 1: The 8 different initial environment categories, for each of which 20 distinct instances were used for testing. Agent density measures the mean number of survivor agents per 100 tiles.

Each of the 480 individual tests ran for 36,000 individual simulation ticks, which is equivalent to 10 minutes of simulation time at 60 ticks per second. Every time the agent population changed a datapoint would be logged; recording the current time, the number of agents of both types, and the average time spent per tick to simulate the agent AI. The data for each of the 20 instances in a category was then aggregated into a single graph of averaged data, with plots for each AI implementation for comparison. This provided a set of 16 graphs in total, six of which are on the following pages to illustrate points identified during their evaluation. Additionally, Table 2 lists the mean final population counts for the three algorithms tested, along with their mean frame time per agent.

	Average Agent Count			Average Frame Time (ms)		
ID	Slow BDI	Fast BDI	Subsump	Slow BDI	Fast BDI	Subsump
A	29.2	23.3	24.6	0.777	0.365	0.692
B	52.4	38.0	44.7	1.343	0.741	2.057
C	116.0	114.4	114.0	2.209	1.230	5.288
D	69.8	61.6	58.9	1.678	0.971	1.893
E	114.7	111.7	110.8	2.333	1.381	4.390
F	30.6	23.9	19.7	1.054	0.714	0.757
G	125.4	98.7	96.9	2.954	1.990	3.099
H	195.9	176.1	182.8	4.206	2.265	6.364

Table 2: Final agent counts and frame times for each algorithm on each initial state category (as specified in Table 1).

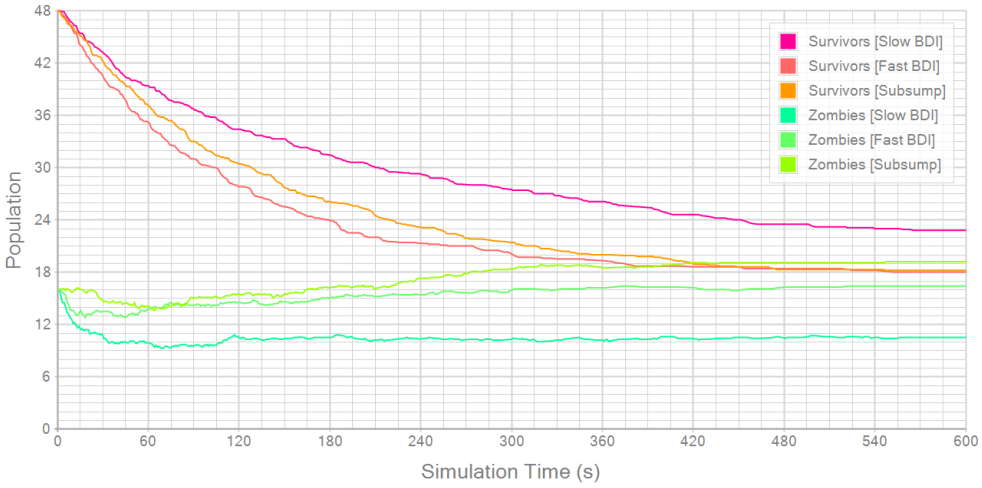

Figure 5: Average population curves for the 60 simulations on a world of size 64x64, with an initial 48 survivors and 16 zombies.

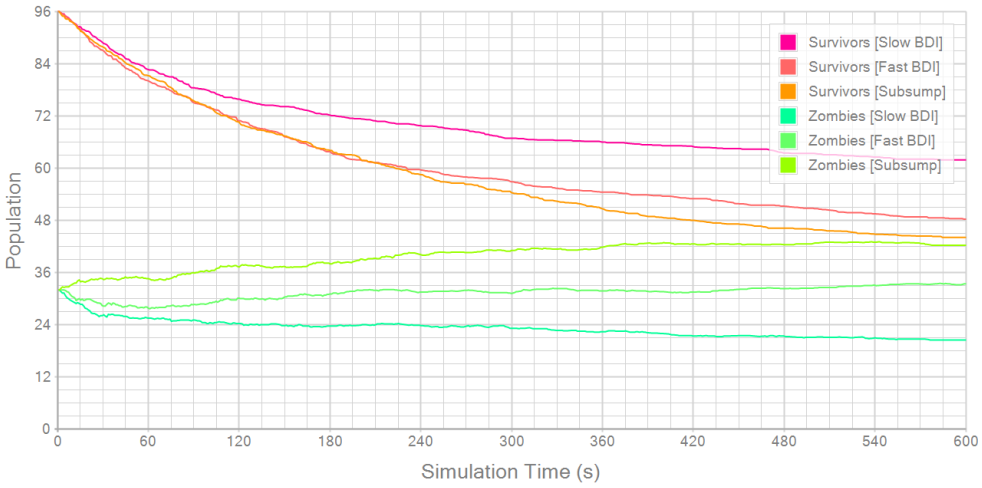

Figure 6: Average population curves for the 60 simulations on a world of size 128x128, with an initial 96 survivors and 32 zombies.

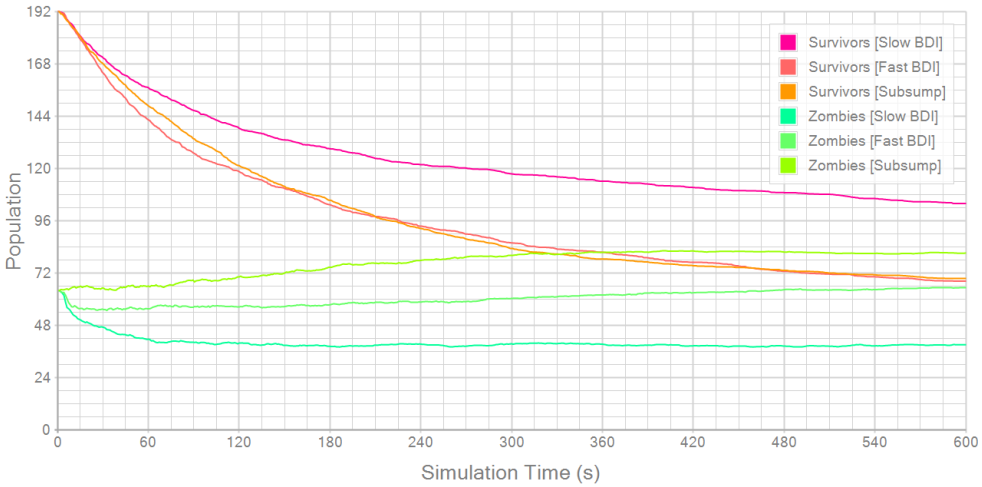

Figure 7: Average population curves for the 60 simulations on a world of size 128x128, with an initial 192 survivors and 64 zombies.

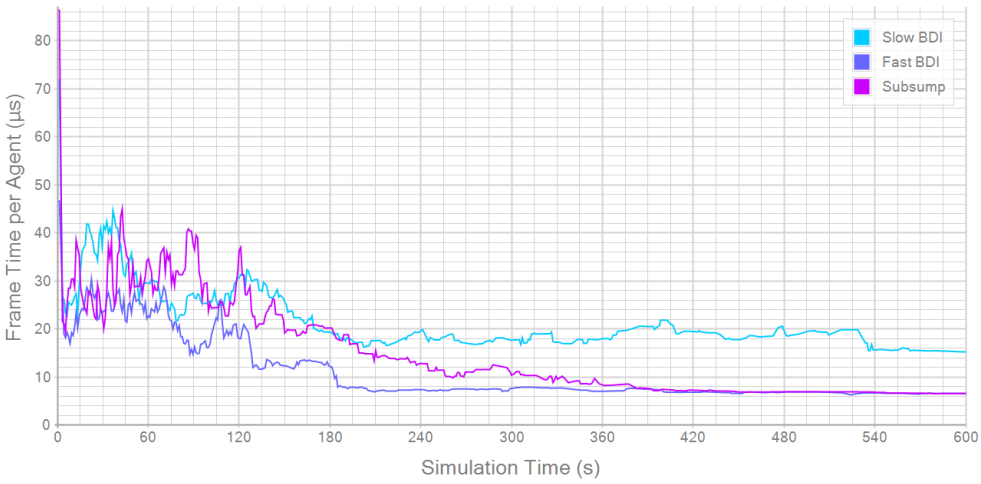

Figure 8: Average AI simulation performance for the 60 simulations on a world of size 64x64, with an initial 48 survivors and 16 zombies.

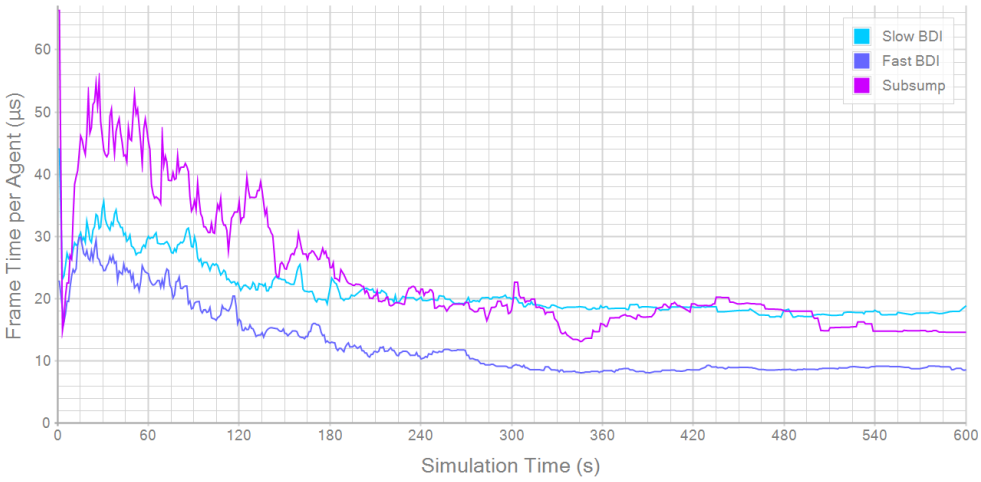

Figure 9: Average AI simulation performance for the 60 simulations on a world of size 128x128, with an initial 96 survivors and 32 zombies.

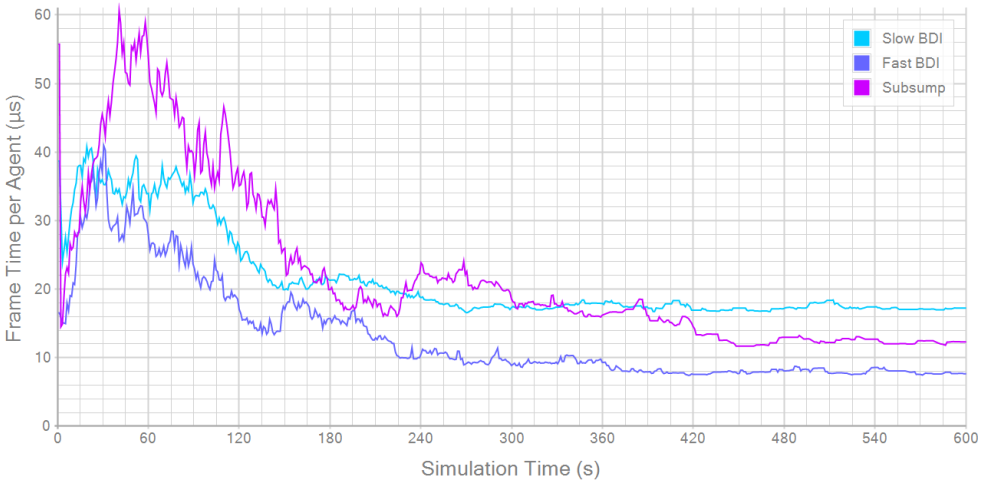

Figure 10: Average AI simulation performance for the 60 simulations on a world of size 128x128, with an 192 survivors and 64 zombies.

V EVALUATION

The survival rate plots clearly demonstrate that one implementation is the most successful at maintaining the agent population, with the more intensive BDI implementation outperforming the others significantly in all instances. The subsumption and less intensive BDI implementations appear to be very similar in terms of survival rate, with the fast BDI maintaining a slightly higher mean population count overall.

As the two BDI implementations differ only in the frequency that the individual agents update their belief databases and deliberate on new intentions, so the considerable difference in survival rates demonstrates that those higher frequencies are certainly advantageous. This will be because a higher belief and intention update frequency allows the agents to respond to a dynamic environment with a lower mean latency, or in other words they have faster reflexes.

Each of the behaviours in the subsumption architecture are polling their respective agent's environment with an equivalent or higher frequency compared to the high-frequency BDI implementation, so a difference in response latency is not the source of the disparity in survival rates between the two algorithms. However, when watching the two algorithms execute an obvious issue with the subsumption architecture becomes apparent that will be a large factor in reducing agent survival rates. There are clear instances where two behaviours will "fight" each other for control of the agent; for example a behaviour to leave a building will cause an agent to step outside, only for a subsumed behaviour to send the agent back inside during the next simulation step. This will continue until another behaviour breaks up the fight, like the fleeing behaviour causing the agent to run further inside. This will lead to neither behaviour achieving their respective goals, and would require either far more complex and therefore expensive logic in each behaviour or communication between behaviours which would violate one of the core principles of the architecture. The BDI implementations avoid such conflicts of interest by aggregating the actions requested by each current intention, placing precedence on whichever has the highest attached utility heuristic. Movement actions are normalised after aggregation, so an agent won't experience restricted movement if it has two intentions pulling it in opposite directions but will instead move at full speed in whichever direction has the highest utility. This is far more rational behaviour, and subjectively appears more natural than the occasionally indecisive subsumption approach.

While the population decline plots show relatively similar behaviour between the different approaches, the frame time data tells a vastly different story. The first thing to note is the large range of frame rates for the subsumption architecture in each graph, suggesting the frame time is wildly unpredictable and so unsuitable for a real-time game environment. The larger peaks near the start of each graph (which the BDI plots also exhibit, albeit in a far more muted form) will be caused by the relatively high density of agents near the start of the simulation, meaning each agent must consider a larger number of nearby objects. The subsumption architecture will suffer more from this as each behavioural layer individually examines its local environment without the processing being shared between them. In comparison, the BDI approach only samples its environment during beliefs updates, the results of which are shared between all desire and intention types that may require them.

Some more trivial observations include the apparent significant benefit in reducing belief and deliberation update frequency for the BDI approach in terms of running time, although admittedly at the cost of agent survival rate. This very simple modification to vary the algorithms efficiency would be very useful in a game setting, where a player may wish to adjust the AI's quality to a level that performs well on their device. The subsumptive architecture cannot be adjusted in such a simple way, and would require each behavioural layer to accommodate a variable quality

level independently. Another observation is that running time per agent for the BDI approaches settles down to a near constant value after the initial peak of high population density. Such a steady and predictable processing time would equate to a steady display update rate, meaning a player won't be subjected to intermittent periods of low frame rates that may disrupt their experience.

From a programmer's perspective, the subsumptive architecture itself was relatively trivial to implement, and each individual behavioural layer is extremely simple as they each are only concerned with one sub-problem of an agent's overall agenda. However, the main difficulty lies in deciding which layers to add and in what order, as complex interactions between individual behaviours are relied on for complex compound behaviours to emerge. Additionally, while certain constructive interactions between the layers may be anticipated and utilised, unexpected destructive interactions may arise such as two competing layers leading to an indecisive agent.

Conversely the implementation of the BDI architecture was much more involved; with quite a bit of effort involved in designing the beliefs database, the desire capturing, filtering and ordering process, intention deliberation, and action aggregation. Thankfully the process of designing and implementing each desire and intention type was far simpler after the underlying architecture was in place, with fewer concerns about how the desires could potentially interact and instead relying on an automatic conflict resolution system between desires and actions. This system has the additional benefit of being easily extensible, whereas the process of adding additional behaviours to the subsumption model would require careful consideration about how potential new layers would interact with existing ones.

Overall, this implementation of the BDI architecture seems to fit this particular scenario well, meeting each of the core requirements of increasing survival rate, performing complex tasks like barricade construction, being computationally inexpensive, and navigating the environment rationally while avoiding unnatural looking behaviour. The subsumption architecture was not as successful in all of these aspects, and additionally was more difficult to implement due to the reliance on complex interactions between behavioural layers.

VI CONCLUSIONS

Performing this analysis of the comparative merits and shortcomings of the subsumption architecture and BDI model for this particular problem instance has certainly been worthwhile, as the results deviate from what may be expected in several respects.

Due to its bulky underlying architecture it may be assumed that the BDI approach would suffer from poorer performance, but it is now evident that this bulk offsets calculations that would otherwise be performed by each individual behavioural layer in a subsumption-based implementation. It is additionally trivial to tweak the performance of the BDI implementation (at the cost of quality), allowing for the possibility of users on less capable machines experiencing an acceptable simulation speed. The BDI implementation also maintained a far more stable execution time for each agent, which translates to a smoother gameplay experience.

It may also be assumed that the subsumption approach would be the easiest to implement, but that turns out to only be true for the underlying architecture. The internals of a decently implemented BDI framework offsets a significant portion of the programming that would otherwise be involved in the implementation of each atomic behaviour, leading to both simpler code and a more efficient solution.

Under subjective analysis the BDI implementation is again more successful for this problem domain, granting the actions of the agents a more natural appearance rather than the occasionally indecisive anomalous behaviour of the agents being controlled by a subsumption stack.

It is therefore fair to conclude that the BDI model implementation for this project is a better fit for the original problem scenario than the subsumption-based approach, and so will be used as the basis for the AI during the continued development of this game project.

There are a few possible avenues for extension of the BDI implementation that may be of value. One is the implementation of shared plans for tasks like barricade construction or the exploration of territory, where one agent coordinates a group of subservient workers. This allows for more coordinated actions while also reducing the time spent deliberating on plans by offsetting the planning of many agents to just one. A further extension would be a more complex barricade design system; currently barricades are only constructed across the entrances of buildings, but this can be improved by having agents decide to block off entire streets of the city to enclose a larger area. This would fit well with the proposed plan sharing system, and would lead to more interesting and possibly more successful agents.

References

Bonura, S., Morreale, V., Francaviglia, G., Marguglio, A., Cammarata, G. & Puccio, M. (2009), 'Intentions in bdi agents: From theory to implementation', *7th International Conference on Practical Applications of Agents and Multi-Agent Systems* **55**, 227–236.

Brooks, R. (1990), 'Elephants don't play chess', *Robotics and Autonomous Systems* **6**, 3–15.

Butler, G., Gantchev, A. & Grogono, P. (2001), 'Object-oriented design of the subsumption architecture', *Software–Practice and experience* **31**, 911–923.

Campbell, A. & Wu, A. (2011), 'Multi-agent role allocation: Issues, approaches, and multiple perspectives', *Autonomous Agents and Multi-Agent Systems* **22**, 317–355.

Hart, P., Nilsson, N. & Raphael, B. (1968), 'A formal basis for the heuristic determination of minimum cost paths', *IEEE Transactions on Systems Science and Cybernetics* **4**, 100–107.

Jaklin, N., Cook, A. I. & Geraerts, R. (2013), 'Real-time path planning in heterogeneous environments', *Computer Animation and Virtual Worlds* **24**, 285–295.

Permadi, F. (1996), 'Ray-casting tutorial for game development and other purposes', http://www.permadi.com/tutorial/raycast/rayc7.html. Accessed Jan 21st 2014.

Rao, A. & Georgeff, M. (1995), 'BDI agents: From theory to practice', *Proceedings of the First International Conference on Multi-Agent Systems* pp. 312–319.

Sturtevant, N. & Buro, M. (2005), 'Partial pathfinding using map abstraction and refinement', *AAAI'05 Proceedings of the 20th national conference on Artificial intelligence* **3**, 1392–1397.