Just-in-time collaborative path-finding for AI agents

Carlos Alfredo Tirado Cortes

School of Computer Science, Newcastle University, UK

c.a.tirado@newcastle.ac.uk

**ABSTRACT**

To this moment, there is a variety of collaborative path-finding algorithms, each of them with their own strengths and weaknesses. Even though current algorithms solve the collaborative path-finding problem easily, most of them have important weaknesses that can’t be ignored (from unsolvable situations, to a large CPU resource consumption). For that reason, it was decided to improve an algorithm such that it’s low cost could be harnessed to solve more complex problems. The aim of this project is to enhance a just-in-time collaborative pathfinding algorithm in order to have an efficient algorithm that can be used in the game industry. To implement a relevant improvement, an algorithm that is capable of making 2 collaborative pathfinding algorithms collaborate with each other will be implemented. Through the development of this project, the development of this algorithm has been the biggest and the most important challenge so far.

1. Start by calculating the F, G, and H values of the current node (this is, the node where the agent is at the beginning of the pathfinding).
2. Add the current node to the open list.
3. Let P be an empty node.
4. Enter the cycle:
   1. If the open list is empty, quit, a path couldn’t be found.
   2. Try to get the value with the lowest value of F from the open list and assign it to P.
   3. If P is the destination, exit the cycle.
   4. Otherwise, add the value of P to the closed list.
   5. Retrieve the adjacent elements of P, calculate their G, H, and F values.
   6. Set the value of P as the father of the retrieved nodes.
   7. Add the retrieved nodes to the open list.
   8. If the element is already in the open list, compare the G values of both and keep on the list the element with the lowest value.
   9. Repeat.
5. Once the goal node has been found, retrace to all the parents of the node to get the path to the node.

**General Terms**

Algorithms, Performance, Design, Experimentation.

**Keywords**

Artificial Intelligence, A\*, Collaborative Pathfinding, Games, C++, Just-in-time Pathfinding.

# **1. INTRODUCTION**

Pathfinding can be defined as calculating the best route from a starting point, to a goal; this calculation usually involves evading obstacles through the map [1] and is usually made by computer software. Once a route has been calculated, the applications can vary from transit planning, telephone traffic routing, [2] and, on the game industry, this could be used in a rage of ways that vary from helping the player to reach a certain point, to having an artificial intelligence (from now on Artificial Intelligence will be referred as AI) use this route to reach its target point.

The A\* algorithm (pronounced Astar) is the most commonly used and one of the most efficient algorithms used for pathfinding [1] [3]. It is important that when using the A\* algorithm, the space where the path-finding is going to take place (from now on this space will be referred to as map) is divided into nodes. Each node represents a space where the path-finding agent can possibly travel. There are some cases when these spaces are of different sizes, but for this project, we will consider that all the nodes are of the same size and that one node can only be occupied by one agent (agent will be, from now on, the name given to the element, or the AI, that is path-finding through the map) at any given time. It is also important to say, that each map has a set of nodes where the agent can’t move, this nodes will be called obstacles.

Each available node, contains 3 values: F, G, and H. The value of G represents the total cost to move from a certain node, to that node. The value of H is the Heuristic value, this value represents an approximation cost from the node being calculated, and the target of the path-finding (without taking into account any obstacles) [4]. Normally, a Manhattan Heuristic (the sums of the x and y distances to the destination) will be chosen for a heuristic [5], never less, there are different flavors of Heuristic that can be used, leading to different types of A\*. Finally, the value of F is the sum of the G value and the H value of a node; the newly calculated F value is normally used to compare nodes against each other when pathfinding. When pathfinding with A\*, there are also two important terms that need to be discussed: open list and closed list. The open list is a list of nodes that are going to be processed, and the closed list is a list of the elements that where already processed. To further understand the terms that where just explained, please refer to the A\* algorithm that is presented on Algorithm 1.

When talking about multi-agent pathfinding, we can describe it as finding a solution for a map with 2 or more agents, each agent having a starting position and a goal position (as in normal pathfinding). A valid solution will be a set of paths where each of the agents reach their destinations without having any type of conflict with any other agent (a conflict can be described as two agents trying to occupy a node at the same time) [6]. For this type of scenarios, A\* presents serious problems, since it ignores the other agents, or it even treats them as obstacles [5]. The solution for this, is to use a cooperative pathfinding algorithm. A cooperative pathfinding algorithm can be described as an algorithm where every agent “knows” of the other agents “intentions”. The algorithms still use A\*, but they use it on a way where each agent takes into account other agents path [5].

Algorithm 1: The A\* algorithm

In this paper, some of the research done for the creation of the collaborative pathfinding algorithm is shown, comparing the algorithms that where found through the investigation for this project A justification for the project is explained, and a project plan for the remaining of the project is presented.

# **2. RELATED WORK**

When speaking about related work, it is inevitable to speak about the Silver’s Algorithm for collaborative pathfinding found in [5]. This algorithm will be explained first, and every other related algorithm will be compared to at least this algorithm. The reason why it is given much relevance to this algorithm and why it is the base of this project is because of the just-in-time capabilities that the algorithm presents, which will fully described up next.

## 2.1 Silver´s Algorithm for Collaborative Pathfinding

It was explained on the introduction that normal A\* presents various problems when trying to apply it as-is to collaborative pathfinding. One of the main problem it presents is that elements end up hindering each other without letting anyone proceed [5].

Silver proposes some enhancements that need to be applied to the A\* algorithm in order to allow just-in-time collaborative pathfinding. These modifications will be explained up next.

*2.1.1 The concept*

The concept of this algorithm is simple. If you have n number of agents, you start calculating agent 1’s path. Since it is the first agent, you calculate its path as if it was the only agent on the map. After the path has been calculated, the second agent’s path is calculated around the first agent’s path. When the third agent’s path is calculated, it is calculated around the first and the second’s agent’s path, and so on. This is just an overview of what the algorithm does, more detailed information about this algorithm is explained next.

**Figure 1: Example 1 representation**

*2.1.2 A third dimension: time*

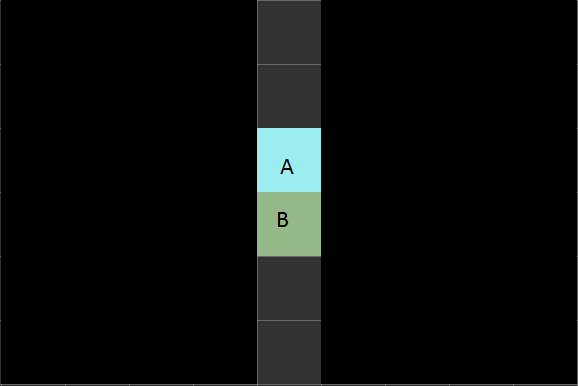
Firstly, Silver tells us in [5] that is essential to add a third dimension to the pathfinding problem: time. The map that is used for normal A\* calculations only had x and y elements to represent their position on the map (this map is defined by [5] as a spatial map), now, the map will have a third dimension t that represents time (this map is defined by [5] as the time-space map). To give an example of how this works, let’s say that an agent a needs to move by one space in the y axis. On a spatial map, the agent will be currently at position (x, y) while the new position will be (x, y + 1); however, on a time-space map, the current position of the agent will be (x, y, t) and the new resulting position of this agent will be (x, y + 1, t + 1) [5] . More about this will be explained in section 2.1.4.

*2.1.3 The pause movement*

Another important change that Silver suggests is that an agent must have another possible movement beside up, down, left and right (or diagonal movements in more complex A\*): pause. The pause movement will have the same cost as the other movements (in the case of complex A\* where diagonal movements are permitted, the pause movement must have the same cost as the up, down, left and right movements). This movement is needed in case the agents needs to maintain stationary until a node is released. The pause movement can be described as starting at the position (x, y, t) and moving to the position (x, y, t + 1) [5].

*2.1.4 The reservation table*

The third element that Silver suggests adding is a reservation table. The reservation table helps to understand why the t element is added into the map, since the reservation table is in charge of saving whether a node is reserved by an agent at a given time t. When a node has been reserved by an agent at a time t, any other agent that needs to use that node at time t will not be able to occupy it. Using a reservation table on this cooperative pathfinding algorithms is essential since it is the way how one agent can communicate to the others what nodes it is using.

Whenever an agent is about to reserve a node in the reservation table, the node must be reserved not only at time t, but also at time t + 1. An example will be used to explain this situation. The test case that will be used for this example is depicted on Figure 1.

In figure 1 we can see that in color black are all the obstacle nodes, in color gray we can see normal nodes, in color blue we can see agent A, and in color green we can see agent B. The current locations of agents A and B are at time t. Now suppose that at time t + 1, agent A needs to occupy the node occupied by node B, and node B needs to occupy the node occupied by node A. If at the reservation table each of the agents only reserved their node at time t, the movement previously described will be possible. As we can perceive from figure 1, the movement is physically impossible given that an agent fully occupies one node. For this reason, a node must be reserved for time t and for time t + 1, so that we can avoid any kind of odd movements like the example just described.

*2.1.5 Heuristics*

The performance of A\* greatly depends on the chosen heuristics [5]. It is stated in [5] that a Manhattan heuristic (or any similar technique) may result in a deficient way of handling heuristic values. This occurs because Manhattan heuristic doesn’t takes into account if there are any obstacles when doing the calculations; this has as a result a scenario where there will be nodes where their F value is lower than the true distance to the destination [5]. That is why Silver states in [5] that a true distance heuristic method should be used. The method to calculate the real heuristic is described in the next section: Backward Search.

*2.1.6 Backward Search*

Silvers proposes in [5] that a backward search should be implemented in order to calculate the real heuristic. A normal A\* (from now on, normal A\* will be used to define the A\* algorithm described at the introduction of this paper, using the Manhattan heuristic as the preferred heuristic) will be executed using as the start point the goal of the agent, and as the end point the node where the agent starts the search (that’s why it is called backward search). Once the F, G and H values for the nodes have been calculated, we look in the closed list for the node we are calculating the heuristic for, once found, we set the value of H to the value of G obtained in the backward search. If the intended node isn’t located at the closed list, normal A\* can continue executing the cycle until the node is on the closed list.

*2.1.7 Other Elements*

There are some other elements that Silver talks about in [5]. Silver explains that to make the pathfinding real-time, a depth must be set. The agent must plan up to d steps in advance; Silver explains that the depth d must be selected according to the size of the map, and the amount of agents traversing the map. This will not only help reduce the consumption of time and memory, but only will help at the calculation of the next d steps, since an element will have to calculate the next d steps after steps have been reached, (of course, only if the agent hasn’t reached its destination).

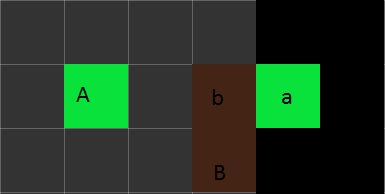
**Figure 2: Collaborative Pathfinding Problem**

Silver explains that this method of pathfinding is another reason why the backwards search was chosen for heuristic; since most of the times we will be traversing partial paths, the use of the real heuristic will help so that the partial paths are leaned towards the correct direction.

Finally, Silver presents a term called the terminal node. In the case when there are two partial paths with the same cost, the terminal node will help decide which partial path to take. The terminal node is the node at d + 1 of the partial path. From that node, a normal A\* will be run to the destination of the agent. The total cost of the result obtained from the A\* will be assigned to the terminal node (the cost is calculated by adding the F values of all the nodes in the path), so that the node that produces the lower cost will be the node of the partial path that will be selected to follow.

*2.1.8 Real Time*

One of the most important features about Silver’s algorithm is that it works on real time. As we saw previously, the agent updates its route after a certain amount of steps. By limiting the amount of steps calculated, the agent gives the illusion that it can “see” a up to an amount of steps, and after those steps are progressed, the agent is able to “see” further away, helping it to plan a better path [5]. This real-time behavior of the algorithm allows us to save time and space in calculating long routes that will probably not be used. Planning far ahead (especially in big maps with large amount of units) has the disadvantage that requires large amounts of memory and resources; this situation becomes worse if ultimately the calculated path is not needed because some changes occurred on the configuration of the map or if some event triggered a change of goal, which ultimately leads to the agent recalculating the path. For these reasons, an algorithm that possess real-time capabilities is the most viable option for a multi-agent pathfinding scenario. This real-time characteristic is one of the main reason why Silver’s algorithm was chosen to be the base algorithm for this project.



*2.1.9 The weakness*

Over all of the advantages that the Silver’s algorithms present, there is one disadvantage that Silver’s present. In Figure 2, a probable scenario encountered in multi-agent pathfinding is depicted. In this scenario, 2 different agents are represented by two different colours: green and brown. As in the previous example, grey squares represent available nodes while black squares represent obstacle nodes. The starting nodes are represented by a capital letter, while the goal nodes are represented by a lower case letter. The problem arises when agent B reaches its goal, because agent B has finished pathfinding, it is blocking agent A from reaching its destination. Several tests where run using this algorithm with similar test cases (where an agent blocks another agent of reaching its destination) and in all of them the algorithm failed to find a valid solution.

**2.2 Conflict Based Search for Multi-Agent Pathfinding**

As stated on the introduction, multi-agent pathfinding consists of two or more agents with different starting positions and different goal positions. The goal of the Conflict Base Search for Multi-Agent Pathfinding algorithm (for now on referred as CBS) is to return a set of movements for each agent, these movements must help the agents reach their destination without conflicting each other [6]. The CBS algorithm is divided into two phases: the high and the low level algorithm. Each algorithm will be explained, but first, some essential terminology will be presented in order to fully understand the algorithms.

*2.2.1 The Concept*

The general concept of this algorithm is simple: the paths of each individual agent are calculated as if they were alone in the map, once the paths are correctly calculated, they are checked one against the other to see if any 2 or more agents are trying to occupy the same node at a specific time, and if so, paths are modified in order to satisfy the conflict encountered. To better understand this lets have an example where we have agent 1 and agent 2 pathfinding through a map. Suddenly, at time t, both agent 1 and agent 2 try to occupy the same node n. From this we can have to scenarios: whether agent 1 stays at time t in the node it was occupying at time t -1 while agent 2 moves through its route normally, or agent 2 is the one that has to wait. After analyzing both scenarios, the cheaper scenario is selected, and the validation of path is continued until both agents reach their destinations without interfering with the other agent’s path.

*2.2.2 Terminology*

In order to fully understand the algorithm, some new terminology is introduced to us in [6] by its authors:

* Path: The term path is used to describe the set of movements one agent must follow in order to reach its destination.
* Solution: k paths for the k agents in the problem.
* Constraint (a, n, t): A constraint is described as an event where an agent a, occupies a node n at a time t.
* Consistent Path: A path that satisfies all constraints.
* Consistent Solution: When all the paths of a solution are consistent.
* Conflict (ai, aj, n, t): When agents ai and aj try to occupy node n at the same time t.

Algorithm 2: The High Level Algorithm

* Valid Solution: When there are no conflicts on the solution.

*2.2.3 Constraint Tree and Constraint Node*

The essence of the algorithms lies on the constraint tree and the constraint node. A constraint tree is a binary tree made up of constraint nodes. This tree must contain a goal node; a goal node is a node where there are no conflicts on its solution (if the tree only has a root node, then the root node becomes the solution). If there are two or more nodes that can be considered goal nodes, then the node with the lowest cost (more about calculating the cost will be explained later) is considered the goal node.

A constraint node is made up of the agents that participate on the problem, their respective paths, a cost for this node (the sum of the costs of the paths) a set of constraints (if the node is the root node, then the set of constraints is empty), the constraints of its parent node (if any) and, if applicable, a conflict.

As it was explained before, there are two parts of the CBS algorithm, the high level algorithm and the low level algorithm. The high level algorithm, is the algorithm in charge of all the logic behind the management of the constraint tree, and the management and creation of the constraint nodes; while the low level algorithm is in charge of the pathfinding elements.

*2.2.4 The High Level Algorithm*

In Algorithm 2, we can see the foundations of the high level algorithm, it is important to have a look at the algorithm before going any further on explaining it.

1. Create an empty node, this node will be the root node of our constraint tree.
   1. This node must have an empty set of constraints.
2. Use the lower lever algorithm to calculate the individual routes of each of the agents in the node.
3. Calculate the cost of the node.
4. Insert the node to the Constraint Tree.
5. Enter the cycle:
   1. If the tree is empty, exit, no solution was found.
   2. Let P be the best node in the tree.
   3. Validate the paths of P.
   4. If the solution has no conflicts, then we have our goal node.
   5. Otherwise, let C be the first conflict of the node.
   6. For each agent in the conflict:
      1. Create a node A.
      2. Add the constraints of P to A.
      3. Add a constraint using the current agent and the position and time of C.
      4. Add the solution of P to A.
      5. Update the solution using the low-level algorithm.
      6. Calculate the cost of the node.
      7. Insert the node to the tree.

Basically, a node is created and the agents are added to the node. Each of the agents paths are calculated as if they were the only agent on the map. Once calculated, the paths are validated against each other. If no conflict is found at each time step, constraints are created. If at the end of the validation there was no conflict, this node is considered the goal node, and the solution to the problem. But, if a conflict was detected at some point, the validation must be completely halted, a conflict must be created containing all the agents that are involved in the conflict, and the location and time when the conflict occurs. If a conflict has been detected, the node must be “expanded” in order to find a solution.

The process of expanding the node works as follows: for every agent in the conflict, a child node will be created. Each newly created node will inherit the parent’s constraints, and will add a new constraint, consisting on the location, the time of the conflict being solved, and the agent of this node. Afterwards, the paths are revised against the new constraint, are modified if needed, and the revision of the paths continue as previously explained. This process is repeated until the goal node is found.

*2.2.5 The Low-Level Algorithm*

The low level algorithm is simply search for the route of the agents of a node in a decoupled manner [6] using normal A\*. If there are any constraints that must be taken into account (when the current node is a child node of a node being expanded), the pathfinding uses them so the resulting path doesn´t enters in conflict with the other’s path.

*2.2.6 Comparison with Silver’s Algorithm*

Compared to Silver’s, CBS is a more complex and more computationally expensive algorithm. During some tests made by the author of this paper, CBS proved to be a more reliable algorithm when delivering a solution for a problem, never less, the solution delivered took more computational resources and more time than the silvers algorithm.

Both algorithms where given different complex test cases to solve, both proved to solve almost all of them correctly (even though Silver’s solved faster) except but one. If we recall Example 1 from Figure 1, we can see a “Head to Head collision in a Narrow Path”. Silver’s successfully solved the problem, but CBS, even though it found an answer, it is solved incorrectly. The reason behind it (as described on Silver’s section) is the way the reservation table is formed. As described in Example 1, the reservation table allows us to avoid this kind of conflicts by reserving a spot for time t, and for time t + 1. Never less, when creating constraints in the CBS algorithm, the constraint is created only for the current time t. As a result, when a problem like this is solved by CBS, the algorithm just “swaps” the positions of the elements resulting on an invalid movement.

*2.2.7 Real Time*

Having a real time algorithm is essential for this project, that is why special attention is put on whether an algorithm is or isn’t real time. As was discussed earlier, Silver’s algorithm contains the real time capability by calculating only a certain amount of steps creating a partial path; after a certain amount of steps are progressed, a new set of steps is calculated, This process repeats itself until the agent reaches the goal node. This is not the case of CBS; firstly, all the routes are pre-calculated, which means that if any change happens, the algorithm lacks the flexibility to modify the route (or even recalculate a whole new route). Secondly, once a route has been calculated and modified according to the constraints and conflicts, the route remains unchanged for the rest of the problem. This is the main reason why, even though CBS remains an important algorithm, it is not the base of the project the way Silver’s algorithm is.

**2.3 MAPP: A Scalable Multi-Agent Path Planning Algorithm**

Multi-Agent Path Planning (or MAPP as it will be referred from now on) is a pathfinding algorithm that identifies all the agents that participate on the map (or at least, all the agents with a solvable solution) and calculates their individual paths without the need of replanning [7].

As it will be explained later, the MAPP algorithm is divided into 2 sections: the path computation section, where all the pathfinding is done, and the alternate paths are created (more on that later); and the movement algorithm, where the movement of the agents in the map is administered.

*2.3.1 The concept*

The concept of this algorithm is based on the idea that there will be no need for replanning a route at any given point during the runtime of the algorithm (unlike Silver’s where replanning is done after a certain amount of steps, or CBS where replanning is done if any conflict is found in the map). On simple words, the algorithms gets all the agents on the map and starts calculating their paths, if during the movement phase, there is an agent that needs to occupy a node that has already been occupied by another agent (or that suddenly changed from being an empty node to an obstacle node), the agent uses a pre-calculated alternative path to get to the node where it was currently heading, that way, a possible collision is avoided.

*2.3.2 Slidable*

In order to understand the two parts that conform the MAPP algorithm, it is important to explain what the term Slidable means. Wang & Boeta, 2001 define a Slidable agent (on their paper it is described as unit, but to maintain a consistency it will still be referred as agent) as an agent whose path meets 3 important characteristics: Contain an alternate connectivity, has an initial blank and has target isolation [7]. The principle of alternate connectivity is described by Wang & Boeta, 2011 as for every 3 consecutive nodes in the path of an agent (except the last 3 nodes, the ones that lead to the destination) there must be a an alternative node that leads from the first node to the third node; this is, if we have three nodes (n1, n2, n3) there must always be an alternative path from n1 to n3 [7]. On Figure 3 (taken directly from [7]) we can see much clearer what the alternate connectivity means.

The second characteristic of a slidable agent is initial blank. This characteristic simply states that the first node of the path of an agent must be an empty node.

The third characteristic is target isolation. By this, the authors of [7] mean that the goal nodes (referred in [7] as targets) of other agents must not be part of any path except their own.

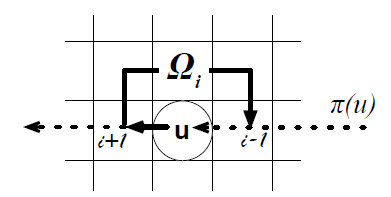


Figure 3: Example of Alternate Connectivity. Ω represents the alternate path. Π Represents the normal path. U is the agent.

*2.3.3 Path computation*

The algorithm for the path computation is quite simple. It calculates each agent’s path individually as if they were alone in the map. The A\* algorithm used is practically the same, except for one difference: the alternate connectivity must be calculated. For this reason, A\* is modified at the time when an adjacent node is being added to the open list. Let’s say there is a node n’, this node has as a parent node n, and at the moment of expanding the node (looking for adjacent nodes) we encounter the node n’’. In order to add n’’ to the open list, there must be a path that goes from n to n’’; if such path exists, the node is added to the open list, otherwise, the node is discarded. As a result, not only the path of the agent is being calculated, but also all the alternative paths are created on this step. These alternative paths are not only for the use of the agent that calculated them, since this alternative paths are between nodes itself, they can be cached so that any other agent can later use them for their own benefit.

*2.3.4 Agent Progression*

The way the agents in a map move depends on their priority. The priority of the agents is set by the order in which the agents move. The first agent to move is the one with the highest priority, the second element is the one with the second highest priority and so on. The priority can’t be set only by the order of the elements, according to [7] the priority can also be set heuristically, this means, since the order of the movement of the agents affects how long will it take for the whole problem to get solved, setting the elements that are nearer to their destination a higher priority will help the rest of the agents to reach their destination faster. Usually, [7] suggests that the priority of an agent is given randomly, and every progression step (more on this below) the priorities should be reassigned.

Another important concept that must be mentioned is the progression step. A progression step is when all the agents involved on the problem have progressed by one step in their paths.

The last concept that will be explained before presenting the algorithm is the private zone concept. According to [7], the private zone is created so that lower priority agents don’t interfere with the progress of higher priority elements. The private zone of an element consists on the current node of the agent (if the current node is part of the main path of the agent) and the past node of the agent (if the past node is part of the main path of the agent, and if the current node is not the starting position of the agent).

Now that we have described some of the important elements, in Algorithm 3 the agent progression algorithm is presented.

1. For each agent that can progress:
   1. If the current node is not part of the path, do nothing, this agent was moved by a higher priority agent.
   2. Else if the current node is a node that was already visited, do nothing, this agent has been repositioned.
   3. Else if the next node is part of the private zone of a higher priority agent, do nothing, wait until that node gets released.
   4. Else if the next node is empty, move to the next node.
   5. Else if the next node is occupied by a lower priority agent:
      1. Move the lower priority agent to an alternate route.
      2. Move to the next node.
   6. Else do nothing.

Algorithm3: The MAPP progression algorithm.

As it can be perceived from the algorithm, an agent can move to their next node only if the next node is empty or if it is being occupied by a lower priority element (if so, the lower priority agent is moved to an alternate path, so that the higher priority agent can occupy this node). It can also be appreciated that if the next node of an agent is part of the private zone of a higher priority agent, then the agent must wait at its current node until the next node is unoccupied. It can also be appreciated that an agent can be moved back to a previous node (on step 1.b), this is called repositioning, and it will be covered in the later section.

*2.3.5 Repositioning*

By the end of a progression step, there may be still active agents that have their advancing condition broken (advancing condition broken means that an agent couldn´t progress to the next node in its path in one progression step). Examples of a broken advanced condition are when the actual position of an agent is not part of their path (hence, it is part of a partial path, hence this agent has been moved from its original path by a higher priority agent), or if the actual position of an agent is part of the path, but the next position is occupied by another agent (or is part of some other private zone). In this cases, repositioning is used to leave the agents on a well-positioned place before the next progression step [7]. A very inexpensive solution proposed by [7] to solve this issue is to use reverse repositioning. By this, the agent retraces the steps it took, and goes back a certain amount of steps until it reaches a well-positioned state. For example if the agent is on a node that is not part of its path, using this method it can go back to a node that is part of its original path, and be ready to move to the next node.

*2.3.6 Comparison with Silver’s algorithm*

The main difference that can be detected between MAPP and Silver’s algorithm is how the paths are being calculated. Silver’s algorithm calculates paths on real-time, while MAPP paths and alternatives are pre-calculated. On silvers, a number n of steps are calculated, and when the agent has traversed steps, some more n steps are calculated. On MAPP, the total route is calculated and all the possible alternate routes available through the path are pre-computed before any of the agents starts to advance. This gives us the conclusion that the MAPP algorithm not a real-time algorithm.

Some other differences between these two algorithms are the use of time, since MAPP doesn’t uses the time element at all; the A\* algorithm used in MAPP is different than the one used by Silver’s algorithm, and the fact that on silvers, even though elements have arrived to their destinations, they continue pathfinding to their goal node (which means they continue executing the pause move on their current node).

The main similarity that both these algorithms contain is the priority. While in Silver’s algorithm, the first agent to calculate its route is the one that gets the preference of what path to take without having to reroute, on MAPP the highest priority element gets to move to whatever node it chooses, even if that node is occupied, the other agent must move and set free that node for the higher priority agent.

*2.3.7 Comparison with CBS algorithm*

The main difference between the CBS and the MAPP algorithm is (like Silver’s) the path calculation. On CBS, paths are pre-calculated individually, and modified according the constraints and conflicts detected. On MAPP, the paths pre-calculated once and never modified.

Another big difference encountered between these two algorithms is the use of priority. CBS doesn’t use the priority element, when two elements try to occupy the same node at the same time, two Constraint Nodes are created, each one containing the two different scenarios where either one agent or the other agent occupies the node first. Once these nodes has been calculated, the cheapest option is selected.

The main similarity that these two algorithms contain is that both are not real-time algorithms since both pre-calculate the paths before execution.

# **3. PROJECT APROACH**

The aim of this project is to enhance the already existing just-in-time collaborative pathfinding algorithms, in order to have an algorithm that is usable in the game industry.

As it can be read from [8], there are many multi-agent pathfinding algorithms that can be used in the game industry. This situation presented itself as an important obstacle for the development of this project. Never less, [8] makes a simple comparison between those algorithms, and even though each individual algorithm present powerful strengths, they also present several weaknesses (for example, an algorithm called TASS is introduced, and it is explained that this algorithm only works on maps with at least four free nodes, for more on this, please refer to [8]) or as it was presented previously, algorithms present several situations where they will not work.

Another important issue with multi-agent pathfinding (and in pathfinding in general) is the amount of computer resources it consumes. Strictly speaking about games, a computer should be capable of calculating efficiently a path for one (or various) agents, while it must fulfill any other demands a CPU can have (such as physics, graphics, or any other AI) [9]. Having a non-real-time algorithm to calculate paths could be harmful for the resources of the computer, because of the need of extra memory to save the individual paths for all the agents (and in cases of algorithms like MAPP, save all the side routes as well). For this reason, a just-in-time algorithm could be beneficial for the CPU resources, since it allows the programmer to choose how many steps ahead are to be calculated (thus, selecting how many steps are to be saved in memory) before recalculating the next amount of steps.

Based on the previous reasons, it was decided that this project will aim to improve Silver’s algorithm. Silver’s algorithm was selected because it is a cheap algorithm in terms of computer resources, and because of its just-in-time capabilities. The area of improvement of the algorithm will be by identifying the situations where the algorithm fails to find a solution, and solve them. In order to achieve this, a detection algorithm will be created to detect which test cases are not solvable by Silver’s; this algorithm must detect in what part of the map the event happens, in what time frames the event occurs, and who are the agents involved in the event. Once this information is available, CBS algorithm will be applied to these elements in order to find a solution to the test case. Afterwards, the previously calculated route will be updated with the solution found with CBS.

The desired outcome of the project will be a just-in-time algorithm for multi-agent pathfinding, which uses a small amount of resources, and is able to solve any kind of complex situation. On table 1, a simple comparison between this new algorithm that is currently under development, and the algorithms explained in the previous section is presented.

Table1. Comparison between the desired algorithm, and the previously described algorithms.

|  |  |  |
| --- | --- | --- |
| Algorithm | Just-in-time capabilities | Resource consumption |
| This projects outcome | Calculates a certain amount of steps indicated by the programmer. | Keeps it at minimal, only a certain amount of steps are calculated. |
| Silver’s | Calculates a certain amount of steps indicated by the programmer. | Keeps it at minimal, only a certain amount of steps are calculated. |
| CBS | None. All paths are pre-calculated. | Some resources are needed to save the routes of all the agents involved in the problem. |
| MAPP | None. All paths (and alternative paths) are pre-calculated. | Resources are needed to save the paths of the agents involved and to cache the alternative paths in case they’re needed. |

# **4. PROJECT PLAN**

This project will be using the programming language C/C++ for the development of the algorithms. The Development Environment used so far is Visual Studio 2013, integrated with the versioning control software GitHub to keep track of the versions of the project.

The working methodology for this project is setting goals that need to be fulfilled in a weekly basis. The first weeks, background reading was done in order to fully understand the topic ahead. Some test code was written to see the various forms of simple A\* working.

The next block of 4 weeks where used to find a direction for the project, as well as to code and test Silver’s algorithm. The results of these weeks of work was having a code base to start working on the project.

The next week a CBS code base was created. The CBS algorithm was coded and tested, having as a result the second part of the code base of this project.

The next two weeks where based on joining the code from CBS and Silver’s algorithm into one single codebase. Some coding was needed in order to make both codebases fit each other. As a result of this week’s work, a codebase that allows the user to choose between the two algorithms was created.

The next week of work was dedicated to create various complex test-cases to test Silver’s algorithm and see if it was able to solve them. As a result, some “unsolvable” test cases where detected, as well as some test cases where Silver’s algorithm proved to work correctly.

The next two weeks of work where dedicated on the creation of an algorithm to detect test cases that are not solvable by Silver’s. To achieve this, some common characteristics from the “unsolvable” test cases where highlighted and used in the detection algorithm. As the result of these weeks of work, the detection algorithm was working.

The next weeks of work, where based on working on making Silver’s algorithm and CBS collaborate in order to solve the detected problems. The code base still contains a relevant amount of bugs, since CBS and Silver’s aren’t fully able to “collaborate” and provide a working solution.

Throughout all the steps of development that where described previously, many small areas of improvement were detected in code of both algorithms; this lead to various corrections done to the code in order for the code to run faster and more efficiently.

For future working, it is planned to eliminate the bugs in the code in order to have a fully working algorithm. Since all the testing has been done on maps that are not considered to be large (maps from sizes that vary from 7 by 7, to maps of sizes 15 by 15), the next step test the algorithm in bigger maps and do the necessary changes if needed. The tests done so far have been done with a maximum of 4 agents per map, like the previous step, the algorithm will be tested with large amounts of agents pathfinding in small and big maps, this with the purpose to detect any possible issue and solve it. Finally, some potential uses for the algorithm will be investigated and tested.

The final objective is to have a just-in-time algorithm that works efficiently, no matter the size of map or the number of agents pathfinding simultaneously.

# **5. References**

|  |  |  |
| --- | --- | --- |
| [1] |  | A. Patel, «Pathfinding,» 1997. Available: http://theory.stanford.edu/~amitp/GameProgramming/. |
| [2] |  | X. Cui y H. Shi, «A\*-based Pathfinding in Modern Computer Games,» *International Journal of Computer Science and Network Security,* vol. XI, nº 1, pp. 125 - 131, January 2011. |
| [3] |  | B. Stout, «Smart Move: Itelligent Path-Finding,» Gamasutra, 12 February 1999. Available: http://www.gamasutra.com/view/feature/131724/smart\_move\_intelligent\_.php?page=4. |
| [4] |  | D. Silver, «Cooperative PathFinding,» University of Alberta, 2005. |
| [5] |  | A. Boeta, R. Herbrich y T. Graepel, «Video Games and Artificial Intelligence,» Microsoft Research, Cambridge. |
| [6] |  | S. Guni, R. Stern, A. Felner y N. Sturtevant, *Conflict-Based Search For Optimal Multi-Agent Path Finding,* 2009, pp. 564 - 569. |
| [7] |  | C. Wang y A. Boeta, «MAPP: a Scalable Multi-Agent Path Planning Algorithm with Tractability and Completeness Guarantees,» *Journal of Artificial Intelligence Research,* nº 42, pp. 55 - 90, September 2011. |
| [8] |  | A. Boeta, B. Bouzy, M. Buro, C. Bauckhage y D. Nau, «Pathfinding in Games,» de *Artificial and Computational Intelligence in Games*, Dagstuhl Publishing, 1998, pp. 21 - 31. |
| [9] |  | N. Sturtevant, «Memory-Efficient Abstractions for Pathfinding,» American Associaton for Artificial Intelligence, Edmonton, 2007. |