Cooperative Pathfinding

David Silver—University of Alberta
silver@cs.ualberta.ca

When a single unit pathfinds through a map, basic A* search is perfectly adequate. But when multiple units are moving at the same time, this approach can break down, often with frustrating consequences for the player. Perhaps you have watched in despair as your twenty prized samurai all try at once to cross a narrow bridge, and then run in circles until they are annihilated. Or maybe your captain has wandered off behind enemy lines when two scouts have crossed his intended route. With cooperative pathfinding, these problems can be consigned to history. Units will move efficiently through the map, avoiding each other's paths and getting out of the way when necessary.

The Problem with A*

Imagine what would happen if you asked everyone in your office to choose a new desk. After they've had a good look around, you decide to blindfold them. Next, you tell them to go to their new desks, in the shortest possible time. Sure, they might already know a good route to the new desk, and how to avoid obstacles. But they can still bump into each other. Each time they collide, they might replan and select a new route, but this won't prevent further collisions. If your office is spacious and sparsely populated, then everyone will quickly find new seats. But with enough people in the office, or narrow gaps between the desks, the whole process can degenerate into a never-ending cycle of collisions and replanning.

The approach described above is used in most current games. A* search is used to find the shortest path to the destination for each unit [Stout00]. This search ignores the presence of other units, or perhaps treats them as stationary obstacles. If a collision is imminent, the units involved will re-search and select a new path.

In many cases this solution is perfectly adequate. Often, constraints are imposed on the level designs to try and avoid the worst-case situation described above. Sometimes, however, problems do occur, and the resulting pathfinding behavior can appear odd, stupid, or just plain broken.

Cooperative pathfinding attempts to remove the blindfolds, and allows units to know each other's intentions. A* search is still used, but in a way that takes account of other units' movements. At first this will make the search much slower, but later in the article we will see ways to regain much of the lost speed.

The only requirement is that units can communicate their planned paths. This is generally most appropriate for units on the same side; enemy units will not usually be so

cooperative. Non-cooperative pathfinding raises the tricky issue of path prediction, and isn't discussed here.

This article assumes familiarity with the basic ideas and terminology of A*. You might want to take a look at an introductory A* article [Stout00, Matthews02] to refamiliarize yourself before proceeding.

The Third Dimension

To tackle the cooperative pathfinding problem, the search algorithm needs to have full knowledge of both obstacles and units. However, when units move around there is no satisfactory way to represent their routes on a stationary map. To overcome this problem, we extend the map to include a third dimension: time. We will call the original map the *space map* and the new, extended map the *space-time map*. Luckily, you don't have to be Einstein to understand how to use it. The space map consists of a two-dimensional grid of *locations:* `Location(x, y)`. The space-time map consists of a three-dimensional grid of *cells:* `Cell(x, y, t)`.

We will be using a simple 4-connected grid to illustrate the basic ideas, using the actions *North*, *East*, *South*, and *West*. Units also need to have one additional action, to *Pause*. In crowded situations, the best option can sometimes be to remain stationary until a bottleneck clears.

Executing *North* corresponds to moving a unit through the space-time map from `Cell(x, y, t)` to `Cell(x, y + 1, t + 1)`. Similarly, executing *Pause* moves a unit from `Cell(x, y, t)` to `Cell(x, y, t + 1)`. All five actions have a cost of one, corresponding to their duration. An action is legal if there is no obstacle at the target location, and no unit using the target cell. The latter is determined by consulting the reservation table, explained below.

A* search can now be used on the space-time map. The goal of the unit is to reach the destination at any time. A* will find the route that achieves this goal with the lowest cost; this is the quickest path to the destination. We will call this procedure *space-time A**, to distinguish it from the usual application of A*, which we call *spatial A**.

In space-time A*, the cost of a path measures the number of actions required to reach the destination. The length of the path can be greater than its distance, because of the *Pause* action. Of course, searching an extra dimension increases the amount of work that A* will have to do, but we will address this issue later.

Reservation Table

Once a unit has chosen a path, it needs to make sure that other units know to avoid the cells along its path. This is achieved by marking each cell into a *reservation table*. This is a straightforward data structure containing an *entry* for every cell of the space-time map. Each entry specifies whether the corresponding cell is available or reserved. Once an entry is reserved, it is illegal for any other unit to move into that cell. The reservation acts like a transient obstacle, blocking off a location for a single time-step in the future.

Using a reservation table and a space-time map, we are able to solve the cooperative pathfinding problem. Each unit pathfinds to its destination using space-time A*, and then marks the path into the reservation table (Figure 1). Subsequent units will avoid any reserved cells, giving exactly the coordinated behavior that we desire.

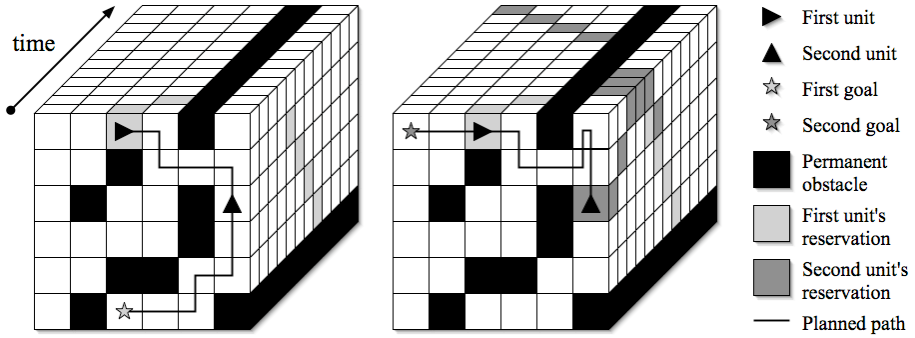

Figure 1 Two units pathfinding cooperatively. (A) The first unit searches for a path and marks it into the reservation table. (B) The second unit searches for a path, taking account of existing reservations, and also marks it into the reservation table.

Unfortunately, this way of using the reservation table doesn't prevent two units crossing through each other, head to head. If one unit has reserved *(x, y, t)* and *(x + 1, y, t + 1)*, there is nothing to stop a second unit from reserving *(x + 1, y, t)* and *(x, y, t + 1)*. This problem can be avoided by making two reservations for each location involved in the action, one at time *t* and one at time *t* + 1. Alternatively, head to head collisions can be explicitly identified and marked as illegal actions. The figures in this article assume this second approach, for clarity.

Space maps are often large (perhaps 256 x 256), and space-time maps will be larger still (e.g. 256 x 256 x 256). Fortunately, we know that they are sparse. The number of reservations at any time-step should correspond roughly to the number of units in the map. This can be implemented efficiently with a hash table, using the space-time index *(x, y, t)* as the key. The `std::hash_map` class, supplied with certain versions of the Standard Template Library, will do just fine.

We now have a complete algorithm for cooperative pathfinding. However, we still have plenty of work to do. Naïve application of this algorithm can be very inefficient. Also, successful cooperation can depend on the order in which units pathfind, whereas we

would prefer a more robust algorithm. Finally, units don't disappear upon reaching their destination, so we require some ongoing processing.

Choosing a Heuristic

The performance of A* depends upon the choice of heuristic. With the search space extended by an extra dimension, the choice of heuristic becomes even more important. First, we consider a simple heuristic, and take a look at the problems it causes. We then consider a more sophisticated heuristic, before moving on to discuss its implementation.

Manhattan Distance Heuristic

For grid-based maps, the *Manhattan distance* is often used as a heuristic. It is simply the sum of the *x* and *y* distances to the destination. It provides a good estimate of the time to reach the destination on an open map. However, if the shortest path to the destination is circuitous, then the Manhattan distance becomes a poor estimate. To understand this, we need to delve into the innards of A*.

During A* search, new locations are kept on the *open list* and explored locations are kept on the *closed list*. At each step, the most promising location is selected from the open list according to its *f* value. The *f* value estimates the total distance to the destination, passing through that location. It is the sum of *g*, the distance traveled by A* to reach the location, and *h*, the heuristic distance from the location to the destination.

Using the Manhattan distance heuristic, many locations on the map can have an *f* value that is less than the true distance to the destination. For example, the map in Figure 2A shows the *f* values for all locations visited by spatial A*. Almost the entire map has been explored before the destination is found, a phenomenon known as *flooding*.

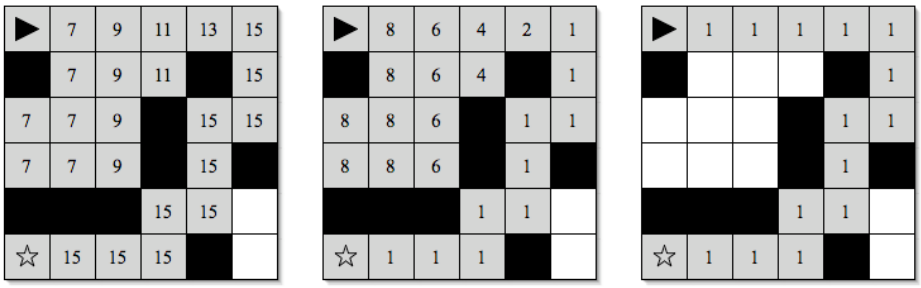

Figure 2 The Manhattan distance heuristic can be inefficient. (A) Flooding occurs in spatial A* when the *f* values (shown) at many locations are lower than at the destination. (B) The number of visits to each location (shown) can be high during space-time A*. (C) Using the true distance heuristic, the number of visits to each location (shown) is optimal.

Now consider what happens if we use the Manhattan distance in a space-time map. Again, it will work well on an open map. But take a look at the equivalent map in Figure 2B, which shows the number of times each location is explored by space-time A*. Not only will space-time A* explore many more locations at the time of each deviation, it will explore those locations at later times too. This is because pausing, or returning to a previous location, appears more promising than moving away from the destination. In other words, the problem has been magnified many times.

True Distance Heuristic

So, it's clear that we need a more accurate heuristic for searching space-time. What about the shortest distance to the destination, taking account of obstacles, but ignoring units? We call this the *true distance* heuristic, and it is exactly the length of the path given by spatial A*. In other words, the true distance heuristic measures the path that a 'normal' pathfinding algorithm would find to the destination.

If we had access to the true distance heuristic, this would be a great improvement. In fact, if no other units get in the way, then space-time A* will proceed directly along the shortest path to the destination (Figure 2C). If other units do get in the way, then it will explore additional cells. With a crowded map we should expect more deviations from the shortest path, and more cells will be explored. The time taken by the search will reflect the level of cooperation required.

The heuristic is required at every location explored by space-time A*. If we naïvely execute spatial A* from each location, this will be even slower than using a poor heuristic. Fortunately, with a little thought we can generate all the heuristics we need from a single spatial A* search. We will investigate this idea further after a small detour.

Consistency

An *admissible* heuristic never overestimates the distance to the goal. A* search with an admissible heuristic is guaranteed to find the shortest path. However, there is a stronger property, known as *consistency* [Russell03]. A *consistent* heuristic maintains the property $h(A) \leq cost(A, B) + h(B)$ between all locations A and B. In other words, the estimated distance doesn't jump around between the locations along a path.

The Manhattan distance is a consistent heuristic. Taking an action will move the unit to an adjacent location, reducing its estimated distance by at most one. The true distance heuristic is also consistent.

A* has a very useful property when a consistent heuristic is used. As soon as a node is put into the closed list, the shortest distance to that node is known. This distance is g, the cost from the start to the node. We will make use of this property in the next section.

Backwards Search

To efficiently calculate the true distance heuristic, we must make a reverse flip and imagine what happens if spatial A* is run backwards. Normally the search begins at the unit's current location, and continues until it reaches the destination. Instead, lets try starting at the destination, and searching backwards.

Take a look at Figure 3A, showing the results of running spatial A* backwards, using the Manhattan distance heuristic. The search has been completed, and all of the colored locations are on the closed list. From the consistency property, we know the shortest distance from the 'start' to each marked location. But because the search is going backwards, we actually know the shortest distance from the destination to the location. In other words, the *g* value for each node contains precisely the quantity we are looking for: the true distance to the destination.

Unfortunately, when the backwards search completes, we might not have the true distances for all required locations. But this can easily be fixed, by continuing the backwards search. A* normally completes when it reaches the goal, but this isn't a requirement. For example, in Figure 3B the shortest path to the unit is of length $f = 8$. But A* can be continued, to find locations on paths which have an estimated length of $f = 10$, $f = 12$, and so on. In fact, we can stop it whenever we like. If we discover that we still need more locations, we just resume the search until we have them.

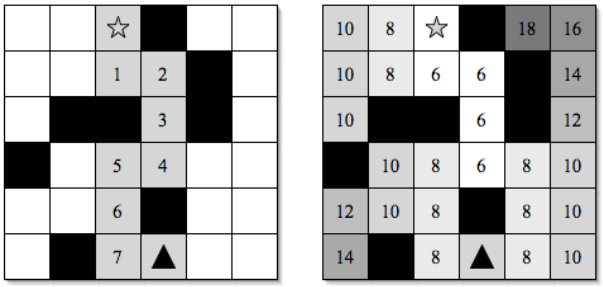

Figure 3 Backwards A* searches from the goal towards the unit's current location. (A) When the search reaches the unit, the *g* values on the closed list (shown) give the true distance to the destination. (B) The *f* values at which a location will be explored (shown) form a series of contours.

To use this idea in practice, we run two searches side by side. Pathfinding is performed by the main search, using space-time A*. At each location explored by the main search, the true distance heuristic is requested. This is where the auxiliary search comes in, using backwards spatial A*. If the location is already on the closed list, its *g* value is returned immediately. Otherwise, the backwards search is resumed until the requested location is put onto the closed list, at which point its *g* value is returned.

So how much work does the backwards search do? Is it actually saving time, or just adding unnecessary computation? It turns out that the backwards search can save a great deal of time [Silver05]. Looking again at Figure 3B, we can see a series of contours expanded by the backwards search. The innermost contours, $f \leq 8$, will contain most of the true distances required. If other units interfere with the path, the main search might need to look further afield. For example, if a unit needs to wait for three time-steps until a choke point clears, then contours up to $f \leq 11$ may need to be searched. This will provide the true distances for all locations that deviate from the shortest path by up to three steps. The amount of work done by the backwards search depends upon the level of cooperation required.

Limiting the Search Depth

A natural question to ask about cooperative pathfinding is how far ahead in time to look? For perfect cooperation long into the future, we would need to look as far ahead as the longest path. But this might be hundreds of timesteps, and who knows what could happen in that length of time? Most likely, the players will have issued new commands by then and all of our hard work will be wasted. There is no point planning for contingencies that will never come to pass.

Furthermore, looking far ahead in time has a significant cost, in both time and memory. Space-time A* would need to search a larger map, due to the longer time dimension. The reservation table would need to store paths for all units, hundreds of steps ahead. Ideally, we should have some upper limit on how much cooperation is required – and only search ahead that far into the future. This estimate should take account of the nature of the map, and the number of units in that locality. A unit crossing a crowded, narrow bridge is likely to be delayed considerably longer than a unit passing through a deserted plain.

We will call this limit on cooperation the *search depth*, denoted by d. For the purpose of illustration, we will assume that d is constant for all units, but this need not be the case. The depth of space-time A* is limited to d steps. If the destination has not yet been reached, then a partial path is returned. The unit begins following the partial path until it is necessary to compute a new one.

The idea of a partial path might be ringing alarm bells in your head. Often, following an incomplete path can be disastrous – after all, it could turn out to be a dead-end! However, space-time A* is using the true distance heuristic to determine its direction; it already has perfect knowledge of the shortest path. The purpose of the search is purely for cooperation, to determine how to avoid other units' planned paths. A partial path in this context will not normally lead to disaster. Units have a tendency to move around, and so most blockages caused by units will be temporary in nature.

Perhaps more importantly, this behavior will still look intelligent to the player – after all, this is closer to how humans behave. Remember the thought experiment with the blindfolds? Cooperative pathfinding removed the blindfolds. Limiting the search depth

specifies how far they can see. No one in the office has a detailed description of each other's plans. But by looking around, each person can get a good idea of their nearby neighbors' short term intentions.

Terminal Node

Limiting the search depth to *d* gives rise to a problem. If the unit doesn't reach its destination, then all partial paths will have a cost of *d*, so how do we choose between them? The answer is to use the true distance heuristic. Consider the partial paths in Figure 4A. Each partial path has a cost of $g = d = 8$, because the destination hasn't been reached. However, the true distance from the end of the partial path can be used to estimate the total cost of each path. We would like the unit to select the partial path with the smallest true distance at the end.

To implement this idea using space-time A*, we introduce a *terminal node* at the end of the search. This node, at depth $d + 1$, represents the destination. The extra cost to reach this node is equal to the true distance from the location at depth *d*. The terminal node effectively summarizes the rest of the search beyond depth *d* (Figure 4B). The overall result of this search is identical to running space-time A* for *d* steps, and then spatial A* until the destination.

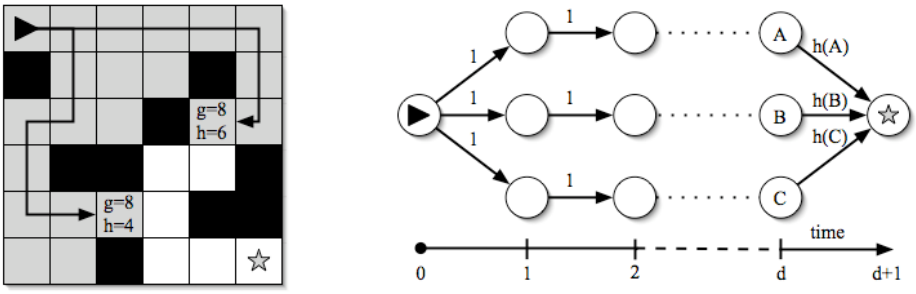

Figure 4 When the pathfinding depth is limited, the true distance heuristic can discriminate between partial paths. (A) All partial paths complete after $d = 8$ steps. (B) A terminal node can represent the remainder of the search in a single step.

The Game Must Go On

Games aren't puzzles. Pathfinding isn't a single problem that we solve, and then watch the solution play out. Units are asked to pathfind intermittently throughout the game. They don't magically disappear at their destination, and other units might still need to pass them, for example at a choke point. We need a system that can deal with ongoing requests for units to pathfind to a destination, and will continue to cooperate once they're there.

Fixed Depth Search

We already have most of the pieces required for this system. However, we need to make some final adjustments to space-time A*. First, it should search to precisely depth d, whether or not the destination is reached. Next, we need to modify the cost of actions. Normal actions, such as *North*, should still incur a cost of 1. However, if a unit *Pauses* on its destination, we assign this action a cost of 0.

Using this system, units never stop pathfinding. Each unit plans d time-steps ahead, even if they are sitting on the destination. This prevents stationary units from becoming permanent obstacles, and means they will try to get out of the way when necessary. If no one needs to pass, then the search cost will be minimal – the unit will just select the zero cost *Pause* action at every step.

Interleaving Searches

Each unit searches to depth d, finds a partial path, and starts following it. At some point it needs to search again, but when would be best? As it comes to the end of the partial path, it becomes less and less cooperative. Eventually it will only have a plan for one step ahead – which is no better than the basic A* approach that we are trying to improve upon! On the other hand, replanning every time-step would be prohibitively expensive. A good compromise is to replan halfway through the partial path. To ensure this will plan sufficiently far ahead, the search depth can be set to twice the required level. If we have estimated that 8 steps of cooperation are required, then we set $d = 16$.

Now that units pathfind continually, there will be n units following paths at any given time. For performance reasons, we would like an equal number of searches to be performed at every time-step. This can be achieved by interleaving the searches, so that roughly $2n / d$ units replan at the same time. If the search depth is constant for all units, this is particularly straightforward to achieve. Just stagger each unit's initial search to begin on a different time-step, and they will maintain an even interleave for the whole game.

Interleaving searches can also make cooperative pathfinding more robust. Consider the paths reserved by two successive units. The first unit selects and reserves a partial path from time t to $t + d$. At the next time-step, the second unit will plan its partial path from time $t + 1$ to $t + d + 1$. The last step of this path doesn't overlap with the first unit; it is chosen independently. In other words, every unit gets an opportunity to be the top dog, making the first reservation at a new time-step. The pecking order rotates around, preventing any single unit from hogging the reservation table. In addition, deadlocks can often be resolved when the order rotates. We have killed two birds with one stone, improving both the performance and the robustness!

Resuming Backwards Search

In our quest to discover true distances, we devised a backwards A* search that can be resumed on demand. But now units are following partial paths, and replanning at regular intervals. The question is, can the backwards search be reused when units replan? Happily, the answer is "yes" – if you have plenty of memory. The drawback is that the backwards A* data (open and closed lists) must be stored separately for every unit.

As you will remember, backwards A* begins at the destination and searches towards the unit's current location. Because the Manhattan distance is consistent, the *g* values on the closed list are the true distances to the destination. But by the time the unit replans, it will have moved location. If we try to resume the search using the old data, the estimated distances could become *inconsistent*.

To avoid this problem, the unit pretends that it stayed put. It searches backwards towards its *original* location, resuming the search as required. The Manhattan distance remains consistent, and the true distances are generated to the destination. The only question is – will it find the locations that it needs?

Take another look at the contours in Figure 3B. If the unit is close to the original shortest path, then it will quickly generate all required true distances. But if it has been pushed far off-course, it could take many contours before the required locations are found – possibly most of the map. Test how much work the backwards search is doing. If it grows too large, then it is probably worth restarting it from scratch.

Unit Priorities

Up until now we have assumed that all units are created equal. However, in many games this is not the case. The player's character, heroes, or arch-enemies could all be considered higher priority by the pathfinding system. Ideally, lesser units should be giving way to more important ones.

One way to achieve this effect is to sort the pathfinding order according to unit priority. But if partial paths and interleaving are used then all units get an approximately equal opportunity to reserve their paths. A better approach is to store priorities in the reservation table. When a unit of priority p finds a path, it makes its reservations at priority p. When another unit with priority $q > p$ plans its path, it will ignore those reservations. If it chooses a path that overrides existing reservations, the overridden units are flagged. Flagged units must then find an alternative path, taking account of the new reservations at priority q.

A simple, binary priority system can also give units a second chance to find a path. Despite our best efforts, it is possible that a unit could have no legal path available. In this desperate situation the unit can be temporarily promoted to high priority. It can then find a way out of trouble, leaving the low priority units to deal with the consequences. If any of these units become stuck, they will be promoted too. They will also search for new paths, only taking account of the other high priority units. There is no guarantee that this

will succeed; there are always situations in which no solution exists. But on reasonable maps this technique provides additional security, at the cost of some additional searches.

Reservation Table Revisited

So far, we have assumed that paths fit neatly into a space-time grid. There was a one-to-one correspondence between the cells used by space-time A* and the entries of the reservation table. However, there is no reason this has to be true. Each unit can structure space-time in the most appropriate way for its own search; the cells of space-time can be of any size. But the reservation table has a fixed structure; its entries are of fixed size and are shared between all units.

For example, two different units might move at different speeds. A slow unit will take longer to move to a new location, requiring a space-time grid with long time-steps (Figure 5A). In contrast, a fast unit would have much shorter time-steps (Figure 5B). Space-time A* pathfinds through the appropriate grid for that unit's speed. The reservation table has its own structure, and is used to communicate availability between the different units (Figure 5C).

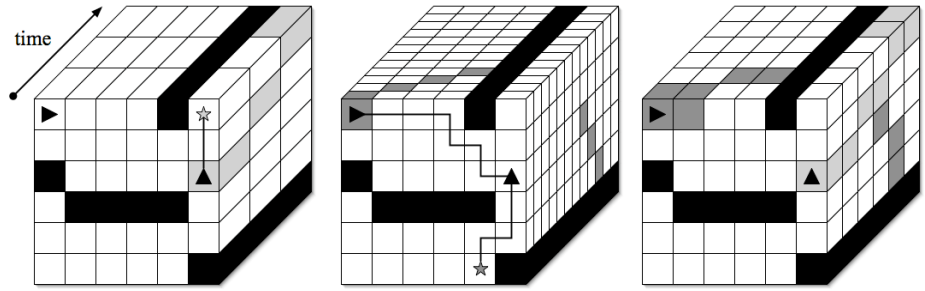

Figure 5 The space-time map can be different from the reservation table. (A) A slow unit has deep cells in its space-time map. (B) A fast unit has shallow cells in its space-time map. (C) The reservation table is shared between both units, despite their differences.

The *footprint* of an action determines the set of entries in the reservation table that correspond to a single move in space-time. For example, the footprint of a diagonal move might include a cube of eight cells (Figure 6A). A large unit's actions would have a wider footprint at each time-step (Figure 6B). Slow actions would have a footprint lasting for more time-steps (Figure 5A). Curved actions would include all entries intersected by the motion through space-time (Figure 6C).

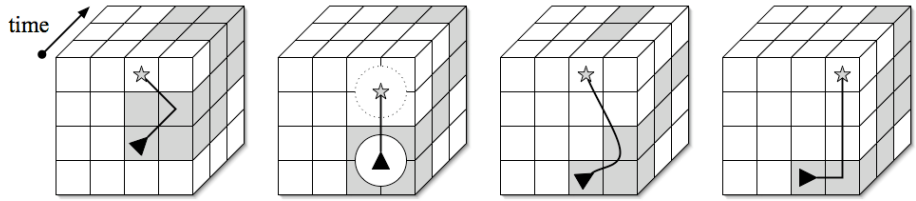

Figure 6 Action footprints for: (A) diagonal movement, (B) a large unit, (C) curved movement, d) varying speeds.

At each step of space-time A*, all actions are tested for legality. To be considered legal, every entry in the footprint of the action must be available in the reservation table. At the end of the search, the resulting path is reserved. All entries in the footprints of the actions along the path must be marked into the reservation table.

If footprints become large, then many entries will need to be checked. However, the granularity of the reservation table can be initialized to any value, depending on the pathfinding requirements. A coarse reservation table will simplify the process of cooperation, but prevent coordination at close quarters. A fine-grained reservation table will require more lookups, but enable tighter maneuvers.

Conclusion

Cooperative pathfinding is a general technique for coordinating the paths of many units. It is appropriate whenever there are many units on the same side who are able to communicate their paths. By planning ahead in time as well as space, units can get out of each other's way and avoid any conflicting routes.

Implementing a space-time search can be tricky to get right. The techniques presented here will get you started, and no doubt you will come up with a few more of your own. Many of the usual enhancements to spatial A* can also be applied to space-time A*. Moreover, the time dimension gives a whole new set of opportunities for pathfinding algorithms to explore.

David Silver 9/4/05 6:47 PM
Comment: Insert reference here?

Try out the ideas in this article. Take the blindfolds off your units. You might be surprised at how intelligent they look, and with any luck, so will the player!

References

[Matthews02] Matthews, James, "Basic A* Pathfinding Made Simple," AI Game Programming Wisdom, pp. 105-103. Charles River Media, 2002.

[Russell03] Russell, Stuart and Norvig, Peter, *Artificial Intelligence: A Modern Approach*, Prentice Hall, 2003.

[Silver05] Silver, David, “Cooperative Pathfinding,” Proceedings of the First Conference on Artificial Intelligence and Interactive Digital Entertainment: pp. 117-122.

[Stout00] Stout, Bryan, “The Basics of A* for Path Planning,” Game Programming Gems, pp. 254-263. Charles River Media, 2000.

Bio

David Silver
silver@cs.ualberta.ca

Shortly after graduating from Cambridge University, David Silver co-founded Elixir Studios, where he was CTO and lead programmer. After five long years developing *Republic: the Revolution*, he realized that game AI wasn't as easy as he had once thought. He left the games industry to return to his academic roots and follow his true passion – research. He is currently studying for a PhD on Reinforcement Learning and Computer Go at the University of Alberta, where he is often found distracted by interesting pathfinding problems.

AIWisdom.com Abstract

Cooperative pathfinding is a general technique for coordinating the movements of multiple units. Units communicate their planned paths, enabling other units to avoid their intended routes. This article explains how to implement cooperative pathfinding using a space-time A* search. Moreover, it provides a number of improvements and optimizations, which allow cooperative pathfinding to be implemented both efficiently and robustly.