Collective Pathfinding in Dynamic Environments

Carlos Astengo-Noguez and José Ramón Calzada Gómez

ITESM Campus Monterrey
castengo@itesm.mx, jrcalzada@itesm.mx

Abstract. Pathfinding is a critical element of AI in many modern applications like multiple mobile robots, game industry and flock traffic navigation based on negotiation (FTN). Classical algorithms assume a unique mobile agent with a complete and accurate model of its environment. New applications demand algorithms that consider multiple agents moving in partially known and changing environments. This paper introduces a new algorithm capable of planning paths for multiple agents and takes the FTN scenario as the basis of an example.

1 Introduction

Navigation is the problem of finding a collision-free motion for an agent-system from one configuration (or state) to another [3]. The agent could be a videogame avatar, a mobile robot, or something else. Localization is the problem of using a map to interpret sensor data to determine the configuration of the agent. Mapping is the problem of exploring and sensing an unknown environment to construct a representation that is useful for navigation or localization. Localization and mapping can be combined.

Motion planning has received considerable attention in the research literature (for a good survey see [3] and [8]). There are a number of interesting motion planning tasks such as navigation among moving obstacles, manipulation and grasp planning, assembly planning, and coordination of multiple agents. Most of the work assumes the agent has a complete and accurate model of its environment before it begins to follow its own planned path; less attention has been paid to the problem of multiple (competitive or cooperative) agents moving in such well known environments and even less attention to those who have partially-known environments.

Once the task and the agent system are defined, an algorithm can be chosen based on how it solves the problem. Due to the nature of the problem, it is necessary to find an adequate balance between optimality, completeness, and computational complexity. We must be willing to accept increased computational complexity if we demand optimal motion plans or completeness from our planner.

Based on how the plan is constructed, planners can be divided in two categories. Offline planners construct the plan in advance based on a known model of the environment and pass it on to an executor afterwards. Online planners build the plan incrementally during execution. A similar issue arises in control theory when attempting to distinguish between feed-forward control (commands based on a reference trajectory and dynamic model) and feedback control (commands based on error from the desired trajectory) [3].

Agents are assumed to operate in a planar (R^2) or three dimensional (R^3) environment or (vectorial) space, called workspace W. This workspace will often contain obstacles; let WOi be the i-th obstacle. The free workspace is the set of points

$$W_{free} = W - \bigcup_i WO_i$$

Motion planning, however, does not usually occur in the workspace. Instead, it occurs in the configuration space Q (also called C-space), the set of all agent configurations. Every dot in the configuration space represents a possible arrangement of the agent in the problem space. A contiguous line in configuration space represents a plan, that is, a sequence of planned locations, starting from a starting point (S) and leading to a target (T).

There is a distinction between path planning and motion planning. A path is a continuous curve on the configuration space. It is represented by a continuous function that maps some path parameter, usually taken to be in the unit interval [0, 1], to a curve in Qfree . The choice of unit interval is arbitrary; any parameterization would suffice. The solution to the path planning problem is a continuous function c $\in$ C0 such that

c : [0, 1] ? Q where c(0) =qstart, c(1) =qtarget and c(s) $\in$ Qfree $\forall$s $\in$ [0, 1].

When the path is parameterized by time t, then c(t) is a trajectory, and velocities and accelerations can be computed by taking the first and second derivatives with respect to time. This means that c should be at least twice-differentiable, i.e., in the class C2. Finding a feasible trajectory is called trajectory planning or motion planning.

There are two main approaches on multiple cooperative pathfinding: Decoupled planning, which plans for each robot independently and coordinates them later; and Centralized planning, which plans the motion of the robots in their "composite" configuration space.

The game industry is demanding algorithms that allow multiple agents to plan for non-colliding routes on congested-dynamical environments [9], such problems also appear in flock traffic navigation (FTN) based on negotiation[2].

FTN uses A* for path planning and, only at intersections, flocks use the Dresner and Stone reservation algorithm [5]. We intend to improve this algorithm using cooperative strategies.

2 Classical (Individual) Pathfinding

The problem in path planning is to find a path for an agent from some location in its environment to another location called target or goal such that obstacles are avoided and a positive cost metric is minimzed. The problem space can be formulated as a set of states connected by directional arcs, each of which has an associated cost. Cost can be defined to be the distance traveled, energy expended, time exposed, etc.

2.1 Pathfinding with A* and D*

A* is a best-first search algorithm that has been widely used in the game industry. It evaluates nodes by combining g(n), the cost to reach the node, and h(n), the estimated cost to get from the node to the goal.

$F(n) = g(n) + h(n)$

A* is complete and optimal if h(n) is an admissible heuristic [10]. D* is an algorithm capable of planning paths in unknown or partially-known dynamic environments in an efficient, optimal and complete manner [12]. The algorithm is formulated as an optimal find-path problem within a directed graph, where the arcs are labeled with cost values that can range over a continuum.

2.2 Cooperative Pathfiding

In cooperative pathfinding each agent is assumed to have full knowledge of all other agents and their planned routes [11]. The complementary problem is called "non-cooperative pathfinding", where the agents have no knowledge of each other's plans and must predict their future movements. There is also another approach called "antagonist pathfinding" where agents try to reach their own goals while preventing other agents from reaching theirs.

There are two possible approaches for multi-agent pathfinding [8],

1. Centralized approach: It takes all agents into account at once and finds all possible solutions. This problem is Pspace-hard.
2. Decoupled or distributed approach: It decomposes the task into independent or weakly-dependent problems for each agent. Each agent can search greedily for a path according to its destination, given the current state of all other agents.

FTN based on negotiation uses a decoupled approach called local repair A*: each agent searches for a route to the destination using A*, ignoring all other agents except for its current neighbors. It is in this neighborhood that negotiation takes place and the flock is created. The agents then follow their route (according to the bone-structure) until a collision is imminent.

It is clear that collisions will happen at the intersections, so there are, with this approach, two possible solutions:

1. Using the Dresner and Stone reservation method [4][5][6][7].
2. Using the Zelinsky [14] brute force algorithm: whenever an agent or a flock is about to move into an occupied position it instead recalculates the remainder of its route.

The implementation of each method depends on the information and the time that the agent or flock has at that particular moment.

The Zelinsky algorithm usually suffers from cycles and other severe breakdowns in scenarios where bottlenecks are present [9] [14]. The Dresner and Stone reservation model was developed for individual agents that can accelerate or decelerate according to the reservation agenda. In simulations performed it was shown that, at the beginning, it works properly, but as time moves on the agents are eventually stopped.

2.3 Cooperative Pathfinding with A*

The task is decoupled into a series of single agent searches. The individual searches are performed in three-dimensional space-time and takes the planned routes of other agents into account. A wait move is included in the agent's action set to enable it to remain stationary. After each agent's route is calculated the states along the route are marked into a reservation table. Entries in the reservation table are considered impassable and are avoided during searches by subsequent agents.

The reservation table represents the agents' shared knowledge about each other's planned routes. The reservation table is a three-dimensional grid: two spatial dimensions and one time dimension. Each Cell that is intersected by the agent's planned route is marked as impassable for precisely the duration of the intersection, thus preventing any other agent from planning a colliding route. Only a small proportion of grid locations will be touched, and so the grid can be efficiently implemented as a hash table [11].

1. New Cooperative Pathfinding Algorithm
Now we can propose a new algorithm that combines D* and Cooperative A*. The algorithms assumes a pre-processing step that calculates preliminary paths for all agents using A*. The A* algorithm must be able to find paths for agents starting from a certain point, moving at a certain speed, and possessing a defined visibility index. The visibility index indicates how far in time can an agent see other agents' paths.

Once the pre-processing step has run, the algorithm loops over the elements of a priority queue. At first agents are added randomly to the queue.

For each agent its path starting from the position occupied at the current time up to the position it will occupy at the current time plus the visibility index is verified against the reservation table. If no collisions are found the agent's delay is set to 0, each step in the path reserved in the table, and its position updated. If a collision was found previous reservation in this step are undone and an altered speed that allows the agent to arrive to the collision point later, in an attempt to avoid it, is calculated.

If the new speed is lower than a pre-determined speed threshold then the agent calculates a new path, in an attempt to minimize its delay. The new path is calculated using A* from the goal to the current position. The algorithm requires the current time, the visibility index and the agent's speed to query the reservation table in order to produce a collision-free path. If the altered speed is not lower than the speed threshold, then the agent's path is adjusted to match the new speed. In either of these scenarios, the delay caused by the collision is calculated.

Finally, if the agent hasn't yet reached its goal, it is added back into the queue, using its delay as the priority, making sure that the most delayed agents have a chance to make reservations in the table before other agents.

```
while !queueIsEmpty() do
  i = dequeue()
  for t in currentTime...currentTime + visibilityIndex
    if reservationTable[path[i][t], t] = -1 then
      reservationTable[path[i][t], t] <- i
    else
      collision <- true
      undoReservations(path[i], currentTime, t)
      newSpeed <- decreaseSpeed(i, reservationTable[path[i][t], t)
      if newSpeed < speedThreshold then
        path[i] <- astar(goal[i], path[i][currentTime], reservationTable, currentTime, visibilityIndex, speed)
      else
        delays[i] <- delays[i] + calculateDelay(path[i], agentSpeeds[i], newSpeed)
        agentSpeeds[i] <- newSpeed
        adjustPathToSpeed(path[i], agentSpeeds[i])
      end
    end
  end
  if collision = false then
    agentPositions[i] <- path[i][currentTime]
  end
  if agentPositions[i] != goal[i] then
    enqueue(i, delays[i])
  end
end
```

2.4 New Cooperative Pathfinding Algorithm Applied to Flock Traffic Navigation

We will explain the classic FTN based on Negotiation algorithm and then propose an improvement to avoid collisions at intersections.

2.5 Flock Traffic Navigation

Flock Traffic Navigation (FTN) based on negotiation is a new approach for solving traffic congestion problems in big cities [1]. In FTN, vehicles can navigate automatically in groups called flocks allowing the coordination of intersections at the flock level, instead of at the individual vehicle level, making it simpler and far safer. To handle flock formation, coordination mechanisms are issued from multi-agent systems.

The mechanism to negotiate [2] starts with an agent who wants to reach its destination. The agent knows an a priori estimation of the travel time that takes to reach its goal from its actual position if he travels alone. In order to win a speed bonus (social bonus) he must agree with other agents (neighbors) to travel together at least for a while. The point in which the two agents agree to get together is called the meeting point and the point where the two agents will separate is called the splitting point. Together they form the so-called "bone" structure diagram.

Individual reasoning plays the main role in this approach. Each agent must compare its a priori travel time estimation versus the new travel time estimation based on the bone-diagram and the social bonus and then make a rational decision. Decision will be made according to whether they are in Nash equilibrium (There is no incentive for either of them to choose another neighbor agent over the agreed one) or if they are in a Pareto Set [13].

If both agents are in Nash equilibrium, they can travel together as partners and can be benefited with the social bonus. In this moment a new virtual agent is created in order to negotiate with future candidates. Agents in a Pareto Set can be added to this "bone" diagram if their addition benefits them without affecting the original partners negatively. Simulations indicate that flock navigation of autonomous vehicles could substantially save time to users and let traffic flow faster [2].

Until now, Agents make their own path planning according to A* and only at intersections use the Dresner-Stone Algorithm.

2.6 A Collision Detection Flock Traffic Navigation Algorithm

FTN based on Negotiation now are decoupled in two main parts:

```
OFFLINE Algorithm
For each Agent do
plan using A* and find
an optimal path traveling and
an arrival time estimation.

REAL Time Algorithm
For each spirit flock do
A reservation $\delta$-time units forward according to
its path
If reservation = TRUE
Follow previous calculated A*-path
else
Conflict Module
```

The critical issue is the conflict Module that can be changed according to the social rules. Here we present a conflict module according to the rules in [2].

```
Conflict Module
Rules
Priority 1: Larger Flock goes first
Then tile is marked as obstacle.
Priority 2: If Size of Flocks are equal
compare delay-table
delayed Flock goes first
Then tile is marked as obstacle
Priority 3: If none previous priority is accomplished
Use Stone-Dresner Algorithm
```

3 Application to Continuous Domains

When applied to computer games, path finding usually takes place on top of one of two representations: A grid structure that wholly describes the traversable game world or a waypoint graph that samples the continuous space over which game agents can move.

The first variant can usually be seen in Real-Time Strategy Games (RTS) and bi-dimensional role-playing games (RPG). The cooperative path finding algorithm in dynamic environments is a perfect fit for these representations and allows game agents to react realistically to the presence of other agents and unforeseen obstacles. Other genres, however, require the simulation of a continuous space updated in fixed time-steps.

Applying a grid-like structure to such spaces can be prohibitively expensive. The algorithm can be modified to work in continuous domains by replacing the reservation table with an analysis of world geometry.

The algorithm consists of an update function, responsible for advancing the state of each agent by a single time-step. The agents are stored in a priority queue that uses each agent's total delays as its key. The function calculates the time elapsed between the last and the current call and uses this value to update the agents. The agents are updated by first performing a collision-detection test between the Minkowski sum of the agent's path and an assigned bounding volume and each of the Minkowski sums of the other agents' predicted paths and their bounding volumes. If a collision is detected, the agent will try to adjust its speed to avoid the intersection point at the intersection time. If this value falls under a specified threshold, the agent will, instead, attempt to calculate a different path.

The predicted paths are calculated using only the other agent's current position, orientation, and speed. These predictions are also limited by the forecasting index, which indicates how far in time are agents willing to predict.

This approach allows the path-planning operation to be distributed across frame updates and the individual update steps for each agent cause no side-effects, making them trivially parallelizable. The algorithm can be further optimized by pruning the collision-detection search by using spatial partitioning schemes such as kd-Trees and by limiting the path used to calculate the Minkowski sums with the forecasting index [15].

The update function is presented below:

```
INPUTS: agents[0...N], forecastIdx[0...N], agentPositions[0...N], agentSpeeds[0...N], speedBound,
agentOrientations[0...N], agentDelays[0...N], agentPaths[0...N], boundingVolumes[0...N]
WHILE agents NOT EMPTY
    a <- agents[0]
    removeFromQueue(a)
    currentPath <- minkowski(agentPaths[a], boundingVolumes[a], forecastIdx[a])
    FOR o IN 0...N
        otherPath <- minkowski(predictPath(agentPositions[o], agentSpeeds[o],
            agentOrientations[o], forecastIdx[a]), boundingVolumes[o], forecastIdx[a])
        IF intersects(currentPath, otherPath) THEN
            slowdown <- calculateSlowdown(agentPaths[a], agentSpeeds[a], intersectionPoint,
                intersectionTime)
            IF slowdown < speedBound THEN
                IF isOtherPathAvailable(a) THEN
                    previousPath <- agentPaths[a]
                    agentPaths[a] <- calculateNewPath(a)
                    delay <- calculatePathDelay(agentPaths[a], previousPath)
                    agentDelays[a] <- agentDelays[a] + delay
                    addToQueue(a, agentDelays[a])
                ELSE
                    delay <- calculateSpeedDelay(agentSpeeds[a], slowdown)
                    agentSpeeds[a] <- slowdown
                    agentDelays[a] <- agentDelays[a] + delay
                    addToQueue(a, agentDelays[a])
                END
            ELSE
                agentPositions[a] <- updatePosition(agentPaths[a], agentSpeeds[a])
            END
        END
    END
    agents <- buildPriorityQueue(agentDelays)
```

4 Concluding Remarks and Future Work

athfinding is a critical element of AI in many modern applications like multiple mobile robots, game industry and flock traffic navigation based on negotiation (FTN).

We develop a new algorithm capable of planning paths for multiple agents on partially known and changing environments inspired by cooperative A* and D*.

From a distributed approach (Decoupled) our collective pathfinding in dynamic environments algorithm decomposes the task of individual plan into weakly-dependent problems for each agent. Each agent can search greedily for a path according to its destination, given the current state of all other agents. Then based on a space-time search space each agent attempt to make a reservation on (x,y,t,δ) where x,y are in the Euclidean space, t is a time measure and δ is a forward planning-vision measure (forecasting index).

Because these kinds of algorithms are problem dependent we developed a modification of our collective pathfinding in dynamic environments algorithm in the FTN context.

Taking care that in FTN the main issue is that it is based on negotiation a conflict-solver module that has the social rules within.

Evidently in FTN we will not in general obtain a globally optimal path from the individual agent perspective but it is a at least a better plan compared with traveling alone (the worst scenario is if an agent can't find Nash or Pareto partners in the whole path so, it becomes a 1-individual flock).

It was shown that the algorithm can, with relatively few modifications, work on continuous domains updated in fixed time-steps, such as those used by most 3D computer games. The shift from using a reservation table to analyzing world geometry allows the work to be cleanly distributed amongst the agents. This creates a clear separation of concerns and the lack of side-effects makes it trivially parallelizable.

We develop a collective pathfinding in dynamic environments in the two representations: A grid structure that wholly describes the traversable game world like the FTN scenario and also into a graph that samples the continuous space over which game agents can move. Future work will be focused on this last issue.

References

1. Astengo-Noguez, C., Sánchez-Ante, G.: Collective Methods on Flock Traffic Navigation Based on Negotiation. In: 6th Mexican International Conference on Artificial Intelligence (MICAI 2007). Springer, Aguascalientes (2007)
2. Astengo-Noguez, C., Brena, R.: Flock Traffic Navigation Based on Negotiation. In: Proceedings of 3rd International Conference on Autonomous Robots and Agents (ICARA 2006), Palmerston North, New Zealand, pp. 381–384 (2006)
3. Choset, H., Lynch, K.M., Hutchinson, S., Kantor, G., Burgard, W., Kavraki, L.E., Thrun, S.: Principles of Robot Motion: Theory, Algorithms, and Implementations. MIT Press, Boston (2005)
4. Dresner, K., Stone, P.: Multiagent Traffic Management: A Reservation-Based Intersection Control Mechanism. In: The Third International Joint Conference On Autonomous Agents and Multiagent Systems (AAMAS 2004), New York, July 2004, pp. 530–537 (2004), http://citeseer.ist.psu.edu/630629.html
5. Dresner, K., Stone, P.: Multiagent Traffic Management: An Improved Intersection Control Mechanism. In: The Fourth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2005), Utrecht, The Netherlands, July 2005, pp. 471–477 (2005), http://www.cs.utexas.edu/users/kdresner/pubs/files/2005aamas-dresner.pdf

6. Dresner, K., Stone, P.: Human-Usable and Emergency Vehicle–Aware Control Policies for Autonomous Intersection. In: The Fourth Workshop on Agents in Traffic and Transportation (ATT 2006)., Hakodate, Japan, May 2006, pp. 17–25 (2006), http://www.cs.utexas.edu/users/kdresner/pubs/files/2006att-dresner.pdf
7. Dresner, K., Stone, P.: Sharing the Road: Autonomous Vehicles Meet Human Drivers. Sharing the Road: Autonomous Vehicles Meet Human Drivers. In: The Twentieth International Joint Conference on Artificial Intelligence (IJCAI 2007), Hyderabad, India, January 2007, pp. 1263–1268 (2007), http://www.cs.utexas.edu/users/kdresner/pubs/files/2007ijcai-dresner.pdf
8. Latombe, J.C.: Robot Motion Planning, 1st edn., 672 pages. Kluwer Academic Publishers, Springer (1991)
9. Pottinger, D.C.: Coordinated Unit Movement. Game Developer Magazine 3(3) (January 1999), http://www.gamasutra.com/features/game_design/19990122/movement_01.htm
10. Russell, S., Norvig, P.: Artificial Intelligence A Modern Approach, 2nd edn. Pearson Education Inc., New Jersey (2003); Personal Library, ISBN 0-13-790395-2
11. Silver, D.: Cooperative Pathfinding. American Association for Artificial Intelligence (2005)
12. Stentz, A.: Optimal and Efficient Path Planning for Partially-Known Environments, pp. 3310–3315. IEEE, Pittsburgh (1994)
13. Wooldridge, M.: Introduction to Multiagent Systems. John Wiley and Sons, Chichester (2002)
14. Zelinsky, A.A.: Mobile Robot Exploration Algorithm. IEEE Transactions on Robotics and Automation 8 (December 1992)