Contents

List of Figures

Chapter 1

Introduction

Path finding is the method of estimating an optimal path from given source and target locations without any collisions with the environment. The movement space for the agents may be multi-dimensional (2D or 3D).

Multi-Agent path finding is an extensively researched topic because of the challenges involved in scaling the navigation algorithm to a huge number of agents. The main objective is to find optimal paths for multiple agents without any collisions with the environment as well as among the agents themselves, in a known environment given the source and target locations for all the agents.

We suggest an improvement to a Multi-Agent Path finding algorithm called Conflict Based Search (CBS) in complex environments. We used a 3D Engine called Unity for our experiments.

1.1 Organization of The Report

The project report is organized in a convenient way to provide a general introduction as well as showcase the results of experiments performed in the project.

First, we provide an introduction to the concept of path planning and its applications in robotics. Then we move on to current algorithms in the field of single and multi agent path finding. The Conflict Based Search (CBS) algorithm is explained in detail, after which we discuss the theory behind our proposed improvement for a Conflict Avoidant CBS algorithm (CA-CBS) and the changes required in the original algorithm for the improvement. We then show the results of experiments performed using our new Conflict Avoidant CBS algorithm.

1.2 What is Path Planning?

Path planning means to find an optimal or sub-optimal path from a source location to a target destination. Optimization may be in terms of distance travelled or forces applied to the object in motion. The main steps of Path Planning are exploration, path finding and motion execution.

- **Exploration** is key to path planning because often we encounter environments which are completely unknown instead of a pre-mapped environment. Also a good exploration algorithm is robust to the changes in dynamic environments.
- **Path finding** Once the exploration step is done, the map generated for the environment can be used for finding an optimal route from the source to the destination.
- **Motion execution** is crucial for a finer optimization to the object in motion.

Path planning is difficult because of the challenges in developing algorithms that take care of dynamic changes in the environment for not just one but multiple agents working in coordination. Current research in this field is tackling this specific problem of modelling the real world as close as possible.

1.3 Applications of Path Planning

Path planning is useful in almost all automated robotic systems including self driving cars, space rovers, delivery systems, video games, architectural design etc. With the advent of autonomous vehicles and automated delivery systems the need for navigation systems are realized in many fields. A few of them include -

- **Self Driving Cars** - In the current age of automation, self driving cars are becoming a reality, and with it, the need for coordinated autonomous navigation algorithms with little or no human intervention. Self Driving Vehicles are capable of sensing their environment and navigating through obstacles and traffic to reach their destination.
- **Drones / Unmanned Aerial Vehicles (UAVS)** - Drones are extremely useful in many areas and have been quite promising in recent years. They have a variety of applications in mapping, police investigation, search pursuits etc. To leverage the capabilities of drones, economical strategies of distributed sensing and cooperative path planning is necessary.
- **Smart Bins** - Smart bins are robots that can identify garbage objects and their task is to clean up an area on it's own. Navigation systems are required for Smart bins for movement in the area without any collision with other objects or people.

1.4 Unity 3D

We decided to use Unity for our purposes as it would act as a rapid prototyping tool to simulate and test our algorithms.

Fig. 1.1 Unity Engine

Unity is a 3D Game/App Development and Simulation engine. The Unity editor is popular among Indie game developers and designers for creating rich experiences with the huge variety of features that the engine provides. It boasts an extensive documentation of the platform and a large community of developers eager to help you out with problems.

Chapter 2

Prior Work

There has been extensive research on both single and multiple agent path planning for static as well as dynamic environments. All the algorithms have a common objective of providing a collision-free path for all the agents in the environment. Specific algorithms perform better in specific conditions and it's upto the developer to decide which algorithm to use for their purpose. A few algorithms like Wave propagation through Potential Fields, Flood Search, probabilistic road maps and rapidly exploring random trees have been discussed below.

2.1 Single Agent

2.1.1 Potential Fields

Potential fields [HA16] work on the concept of wave propagation or a field of gradients in a grid environment. The whole environment is considered as an artificial potential field.

The goal attracts the agent and obstacles repel and correspondingly gradients are calculated for each of the nodes. The gradients point towards the best possible move for any agent in any given location in the environment.

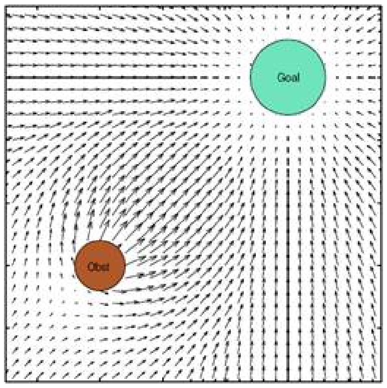

Fig. 2.1 Gradients

The most important benefit of using potential fields is the
fact that the fields are a function of the goal location and the obstacles in the environment only. They do not depend on the initial location of the agent. Hence, once the fields/gradients are calculated we do not need to recalculate them unless the environment or the goal location changes.

A disadvantage of Potential Fields is that it suffers from being trapped in a local minima before ever reaching the destination.

2.1.2 Flood Search using Multi-Bug Path Planning

For environments with a lot of obstacles a flood search [VB16] may be necessary for finding the most optimal path to the goal. A bug is a virtual object that is assumed to be moving in a particular direction towards the goal and keeps on moving till it reaches the target location.

When the bug meets an obstacle it splits into multiple bugs for the different path choices. The bugs move along the walls until specific conditions are met. This process continues till the target location is reached by all the live bugs. The best possible path to the target location can be found in this manner.

2.1.3 Randomized sampling Techniques

Various randomized sampling techniques have been devised for the purpose of gaining speed of arriving at the solution at the cost of optimality. Most of the algorithms work by greatly reducing the number of states to be explored using a variety of heuristics so as to find a bounded optimal route faster.

One such application of randomized sampling for motion planning is using probabilistic road maps (PRMs).

- **Probabilistic Road Maps** - The main steps of the algorithm are to take random samples from the environment, check whether they are in free space, and connect nodes that have a obstacle free path between them. The source and target nodes are then included. This way a smaller number of nodes are generated than naively dividing the whole environment into uniform grids.

- **Rapidly Exploring Random Trees** - This algorithm was designed to solve path planning in non-convex or higher dimension space by randomly building a space-filling tree. The tree is generated from random samples drawn from the search space.

  This algorithm is inherently biased to grow towards large unexplored areas of the problem. They can handle problems with a large number of obstacles and constraints.

2.2 Multi-Agent

The problem of multi agent path planning is NP-Complete and grows exponentially with the number of agents. The most commonly used objective to optimize is the Sum of Individual Costs (SIC).

2.2.1 A* Variants

Many different variants of A* [MG16] have been implemented for multi-agent path finding. The state space of a standard A* based search consists of all possible permutations of the agents on all the vertices. To prevent searching all the possible permutations various heuristic based methods are used to prune the search space.

- **Pattern Databases** The main idea behind it is to first abstract the state space by only considering a subset of variables and constraints. After that a complete BFS is performed on the abstract state space from the abstract goal. Distances to all abstract spaces are calculated and stored in a lookup table, which are then used throughout the searching as heuristics for the original state space.

- **Independence Detection (ID)** As the size of the state space is exponential with the number of agents, the algorithm [Sta16] tries to identify independent groups of agents for reducing the complexity of the state space. The inter-group path finding is done using various heuristic mechanisms to try and resolve conflicts among those groups. If any groups still have conflicts, they are united to form a single large group. Intra-group path finding is then done using standard A* search.

  This way the time complexity of the overall algorithm depends on the A* search algorithm of the largest group.

- **Operator Decomposition (OD)** The branching factor or the number of possible next moves is also exponential with the number of agents. OD is just another heuristic used to solve the problem of surplus nodes in the standard A* branching node expansion.

2.2.2 CBS

Another powerful algorithm is Conflict Based Search (CBS) Algorithm for Multi Agent Path finding. Our work is primarily involved with CBS and it's improvement. Hence, the algorithm has been discussed in greater detail in the next chapter.

Chapter 3

Conflict Based Search (CBS)

We used CBS [GSS12] for our multi-agent path finding problem. CBS works by decomposing the problem into a large number of single agent path finding problems, although there might be an exponential number of such single agent problems. The algorithm is divided into two levels - a High Level (Searches a Constraint Tree) and a Low Level (Constrained Single Agent path finding).

3.1 High Level Algorithm

The high level algorithm updates and searches a constraint tree (CT). Each node in the CT consists of three different things.

- **Constraints** - A set of constraints for each of the agents
- **Solution** - The list of paths from source to destination for each agent
- **Cost** - The cost of the solution (e.g. SIC or Sum of individual costs)

The algorithm is as follows:

```
def GetSolution(sources, targets, constraints):
    for all agents i:
        solution += LowLevel(source[i], target[i],
                             constraints.forAgent(i))
    return solution


def CBS(sources, targets):
    # initial constraints empty
    node.constraints <- $\phi$
    node.solution <- GetSolution(sources, targets, $\phi$)
    node.cost <- getCost(new_node.solution)

    OPEN.add(node)

    while not OPEN.empty():
        N <- OPEN.bestNode()
        C <- getFirstConflict(N.solution) # (ai, aj, v, t)
        if C == None:
            # no conflicts => solution found
            return N.solution
        for each agent ai in C:
            n = new_node
            n.constraints <- N.constraints + (ai, v, t)
            n.solution <- GetSolution(sources, targets, n.constraints)
            n.cost <- getCost(n.solution)
            OPEN.add(n)
```

At each iteration the node with the minimum cost (and the minimum number of constraints) is traversed first. At each node we check for conflicts in the solution. In our implementation we have considered two types of conflicts - *vertex conflicts* and *edge conflicts*. An example of a vertex conflict is when agent 1 moves from X to Y and agent 2 moves from Z to Y at the same time step. Here, both the agents occupy the same vertex at the same time. An example of an edge conflict is when agent 1 tries to move from X to Y and agent 2 moves from Y to X at the same time. These kind of movements are not allowed and are therefore considered as conflicts.

Once we get a conflict, we add nodes to the tree with new constraints that prevent that conflict from occurring. An example of a constraint would be preventing agent 1 from moving from X to Y or agent 2 from Z to Y at the exact time of conflict in the above example.

This process is repeated till we reach a node in the Constraint Tree without any conflicts. The solution of that node is the most optimal solution as proved in [GSS12].

Proof - Consider any node N with constraints C and cost C(N). Let the cost of the global optimal solution C(g).

$$\mathrm{C(g)} \geq \mathrm{C(N)}, \textit{ for any node N}$$

C(N) acts as a lower-bound to the solutions of all children nodes upon expansion of node N. This is because, upon expansion, constraints only get added and not removed, meaning the solutions of the nodes in the sub-tree of N cannot perform better than C(N).

$$\mathrm{C(N')} \geq \mathrm{C(N)} \textit{ for all N' nodes in the sub-tree of N}$$

Also, g will eventually be reached by continuous expansion by atleast one of the leaf nodes in CT at any time step.

$$\mathrm{C(g)} \geq \mathrm{C(N)} \textit{ for some N in the tree}$$

Since CBS explores the tree in a best first manner,

$$\mathrm{C(g)} \geq \mathrm{C(N')} \geq \mathrm{C(N)} \geq \mathrm{C(g)}, \textit{ for all subnodes N' of N}$$

3.2 Low Level Algorithm (A* Search)

The low level algorithm for finding constrained paths from the source to the destination for each of the agents may be any generic single agent path finding algorithm. In our experiments the target locations for each agent keeps re-spawning to a new location each time the target is reached. Hence for our purpose we decided to use the A* algorithm with path constraints.

The constraint obedient A* algorithm is described below.

```
# def LowLevel(source_node, target_node, constraints):
def AStar(source_node, target_node, constraints):
    open_list.add(source_node)

    while(!open_list.empty()):
        cur_node <- min_f_cost_node(open_list)
        if(cur_node == target_node) break; # target found

        open_list.remove(cur_node)
        closed_list.add(cur_node)

        for all neighbours:
            if neighbour not in closed_list:
                if (neighbour, cur_node.time_step + 1) not in
                    constraints:
                     if new_g_cost is smaller or neighbour not in
                         open_list:
                          neighbour.gcost <- new_g_cost
                          neighbour.hCost <- get_hCost(neighbour,
                              target_node)
                          neighbour.best_move <- cur_node
                          neighbour.time_step <- cur_node.time_step + 1
                          open_list.add(neighbour)
```

We can then finally get the optimal path by simply following the *best-move* variable from the target node to the source node.

Here, H-Cost is the heuristic cost used to optimize the search, G-Cost of a node is the actual cost of travelling from source node to that node and the F-Cost is the sum of H-Cost and G-Cost. The use of a heuristic cost (H-Cost) gives A* the name - informed search or best-first search. The H-Cost provides an estimate on which nodes to travel first so that the target node is reached without unnecessarily expanding some nodes that would never result in a better route to the goal. This prevents the algorithm from traversing all possible nodes in the map before it reaches the destination node, in turn finding the optimal path faster.

A* is problem specific and returns the most optimal path only if the heuristic function never overestimates the actual cost to reach the goal node.

Constrained A* - For a constrained path search, we pass an object called the constraints object to the function. It is a list of pairs of nodes and time steps. Each of the pairs in the constraints list identify a particular node and a time step at which the agent cannot be present. Hence, a simple check that the pair (neighbour, curNode.timeStep + 1) is not in the constraints will ensure that the agent will not consider the possibility of moving to that neighbour at the next time step. This in turn makes sure that the agent searches only those paths that follow the constraints.

Chapter 4

Proposed Improvement (CA-CBS)

The original Conflict Based Search algorithm depends only on conflict detection and removal for arriving at the solution. We propose a method called **Conflict-Avoidant CBS (CA-CBS)** that achieves better performance on an average as compared to the original CBS algorithm.

At every iteration of the CT Node search, CBS tries to find constrained paths arbitrarily and independently for each of the agents. A conflict avoidance scheme can be enforced into these individual path searches so as to ensure that the agents choose optimal paths that have lesser chances of causing conflicts, instead of naively generating paths based solely on the constraints given by the High Level algorithm. This is achieved using a Conflict Avoidance Table (CA Table).

4.1 Theory

CBS requires that each of the individual paths returned at each node's solution, under the given constraints, be the optimal path for each of the agents under those constraints. But usually there are multiple optimal paths for the same constraints. Using H-Cost for breaking ties among those optimal paths is not as beneficial as giving priority to paths which do not have conflicts. This is because paths with conflicts cause the Constraint Tree

(CT) to be larger.

We therefore include a different Heuristic called v-cost for each of the nodes. Calculating and accessing the v-cost for a node is of constant time per iteration of the A* algorithm, therefore they do not lead to increase of the algorithm's time complexity.

- **V-cost** - v-cost of a node is the count of total conflicts in the path to that particular node from the source node. It is calculated by using a simple method. Lets say there was a transition from node p to node q in the A* grid at time t. Then, v-cost of q, say v(q), is v(p) plus the number of conflicts in node q at time t according to the CA Table.

  In this manner, for each of the agents we find paths that have the least conflicts with the CA Table by giving priority to those optimal paths which have lesser v-cost.

- **CA Table** - For each node in the CT, we use the parent node's solution as it's initial CA Table. This would imply that the agents are more likely to choose the same paths as in the previous solution. This is true for most of the agents because the new constraint added to the node will affect only a few agents (It will definitely change the path of the agent which is assigned that constraint). Therefore, when finding a new path for the agent with the newly added constraint, if that agent is conflict-avoidant with the previously assigned paths of the other agents it will try to find paths that do not disturb the previous solution. In most cases, this would lead to finding optimal solutions faster because of the stabilization of the solution.

  We start by first finding an optimal path for the agent which has the newest constraint. When the path is returned, we update the path of this agent in the CA Table. We start with the newest constraint agent because at each node of the CT only one of the agents receive a new constraint, and this agent will definitely have to find a new path because of this new constraint. Therefore, keeping the original path from the parent node for this agent in the conflict table does not make sense, so we find a new path for this agent first.

After finding the path for that agent, this is repeated similarly to find conflict-avoidant optimal paths for the rest of the agents, each time updating the CA Table with the new path of the agent.

4.2 Algorithm

The **High Level CBS** algorithm needs to change only the *GetSolution()* function as described below in pseudocode.

```
def GetSolution(constraints, CATable):
# Initial CA Table is the solution of
# the parent node in the Constraint Tree
    A <- constraints.newest_agent()
    p <- LowLevel(A.source, A.target, constraints, CATable)
    CATable.update(A, p)

    for all other agents:
        p <- LowLevel(agent.source, agent.target, constraints, CATable
            )
        CATable.update(agent, p)
```

We would also need to modify the **Low-Level CBS** to give priority to paths with lower v-cost. This needs to be done while maintaining the optimality of the path. Hence, we use v-cost only as a tie breaker between nodes with the same f-costs.

```
def min_f_cost_node(list):
    for node in list:
        if min_node.fcost < node.fcost:
            min_node = node
        # tie breaker by v cost
        if min_node.fcost == node.fcost:
            if min_node.vcost < node.vcost:
                min_node = node
    return min_node
```

The main loop in A* now just needs to calculate the v-costs for each neighbour node while calculating the g-costs and the h-costs. Rest of the code remains the same.

```
# def LowLevel(source_node, target_node, constraints):
def AStar(source_node, target_node, constraints, CATable):
    ...
        neighbour.time_step <- cur_node.time_step + 1
        neighbour.vcost <- cur_node.vcost + CATable.getConflictsCount(
            neighbour, neighbour.time_step) # calculating v-cost
        ...
```

4.3 Inference

In CA-CBS, the agents try to find paths that avoid conflicts with -

1. Previous path of agents whose paths haven't been re-calculated yet
2. New paths of agents whose paths have been re-calculated in *GetSolution()*

This prevents the algorithm from naively returning paths that has obvious conflicts as observed in CBS, ultimately causing CA-CBS to perform better than the original CBS in many cases, as observed in the experimental results shown in the next chapter.

Chapter 5

Implementation

We implemented the CBS algorithm along with the corresponding single agent A* path finding algorithm in Unity. C# was chosen as the language for programming. We tackled only the 2D path finding task in our experiments mainly because of two reasons. First, the algorithm can easily be extended to 3D as neither CBS in general nor A* is dimension specific. Second, the 3D path finding algorithm will be harder to debug because of the issue in completely visualizing multiple 3D paths, observing actual collisions and verifying if the solutions are optimal or not.

5.1 The Environment

The environment, enclosed in a rectangular area, consists of irregularly shaped 3D buildings, agents (drones) and targets (red cubes). The area is then divided into uniform grids which are marked walkable or unwalkable with the help of Unity Layers by tagging the buildings as an unwalkable layer. Each of the agents has a single target location inside the environment.

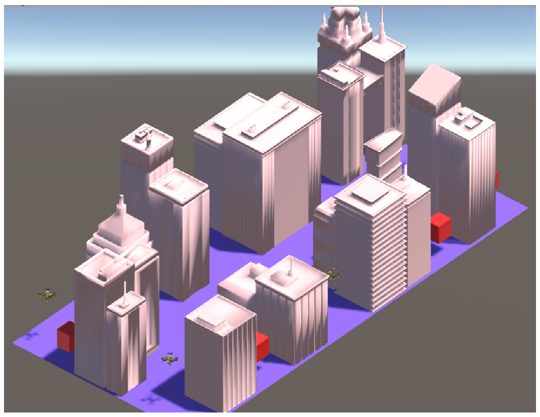
Fig. 5.1 Environment

Source and target locations are preset. An agent can move either up, down, left or right. Diagonal movements are not allowed. Each time an agent reaches it's goal, the goal is then re-spawned at another location. It simulates the actual real world problem of dynamic allocation and reallocation of goal locations. *The task is to find collision free paths for each of the drones from their source to their targets.*

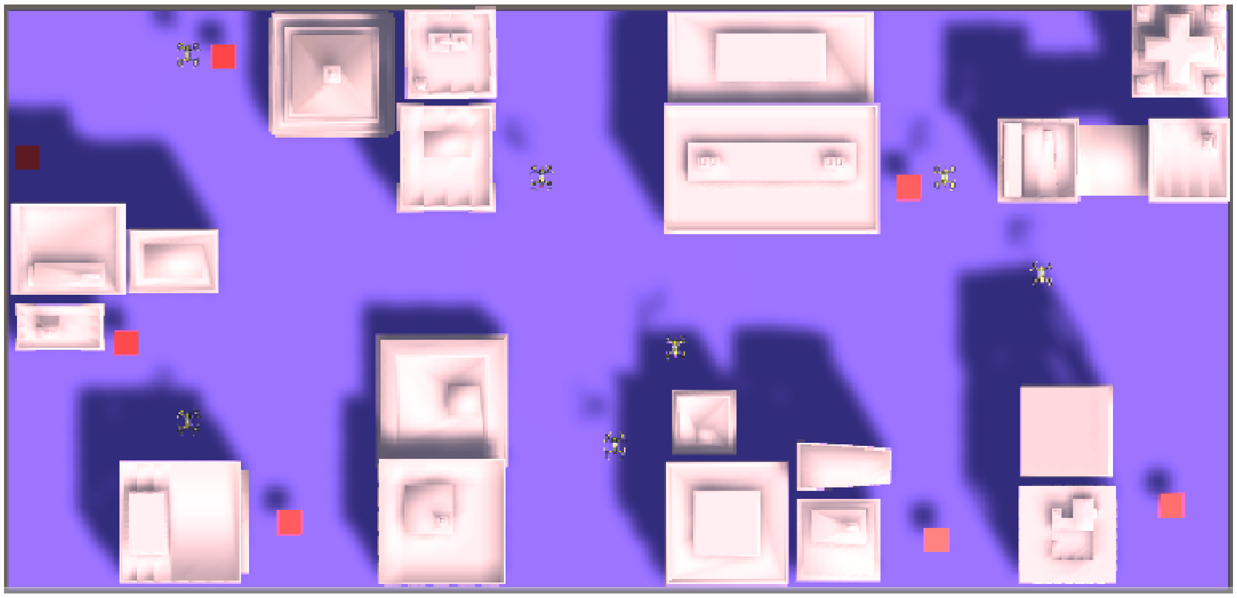

Fig. 5.2 Agents and Targets

We checked the algorithm's time required for different number of agents. For agents less than 10 both the algorithms find the solutions considerably fast. The number of nodes in the CT do not increase much for 10 agents. Therefore we increased the number of agents to 15. For some source and target locations, 15 agents take more than 200 nodes in the CT, although in some cases, both reach the optimal solution fairly fast (within 10 node searches in the CT).

Both the Original CBS and CA-CBS are run to find optimal paths for 15 different agents and goals. Every time a goal location is changed, the process is repeated. We then compare the number of nodes in the Constraint Trees of both the algorithms.

The grids in the environment are highlighted with Unity Gizmos. Grids with obstacles are given a lighter white color. Paths of different agents are given different colors at random.

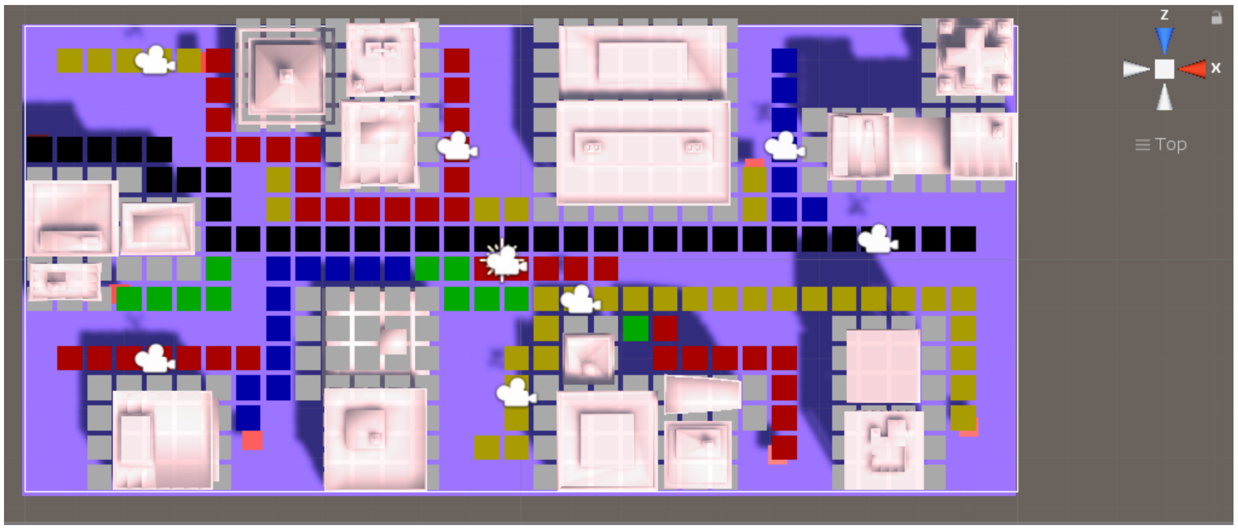

Fig. 5.3 Optimal Paths

5.2 Results and Comparison

Node counts of the CTs were logged to corresponding files for both CBS and CA-CBS for each of the source and target location test cases.

We plotted ratio of the average node count in the constraint tree in CA-CBS to that of normal CBS, in Figure 5.4. The X-Axis marks the number of nodes expanded in the CT for normal CBS. Y-Axis shows the ratio of number of nodes expanded for CA-CBS to Normal CBS.

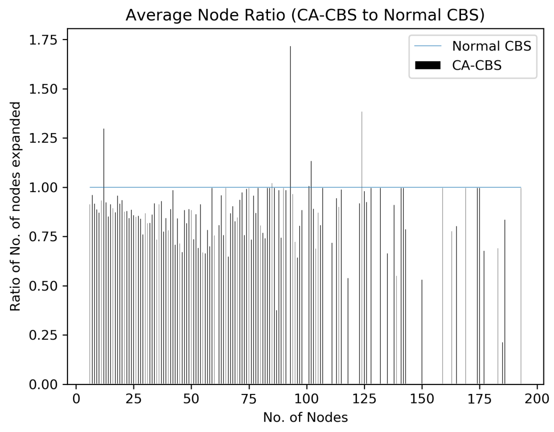

Fig. 5.4 Average Node Ratio

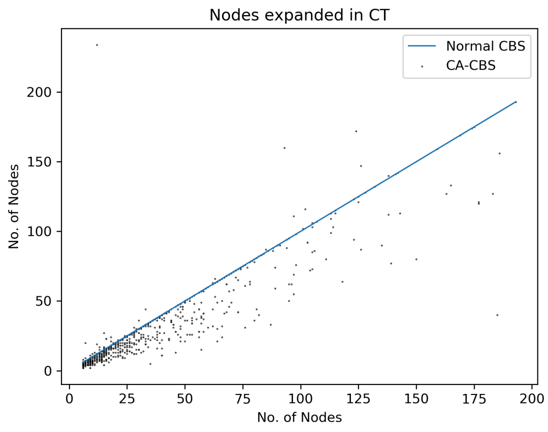

Fig. 5.5 Node Count Comparison

We can clearly see that the ratio of nodes expanded is close to 0.8 to 0.9 for many of the path-finding problems, especially for node counts smaller than 70.

Figure 5.5 shows a scatter plot for the differences in node count in the CT for the two algorithms. Again, the X-Axis shows the node count in the CT for normal CBS. The Y-axis depicts the node count of the CT for CA-CBS. We can see that many of them have lesser value than that of the original CBS.

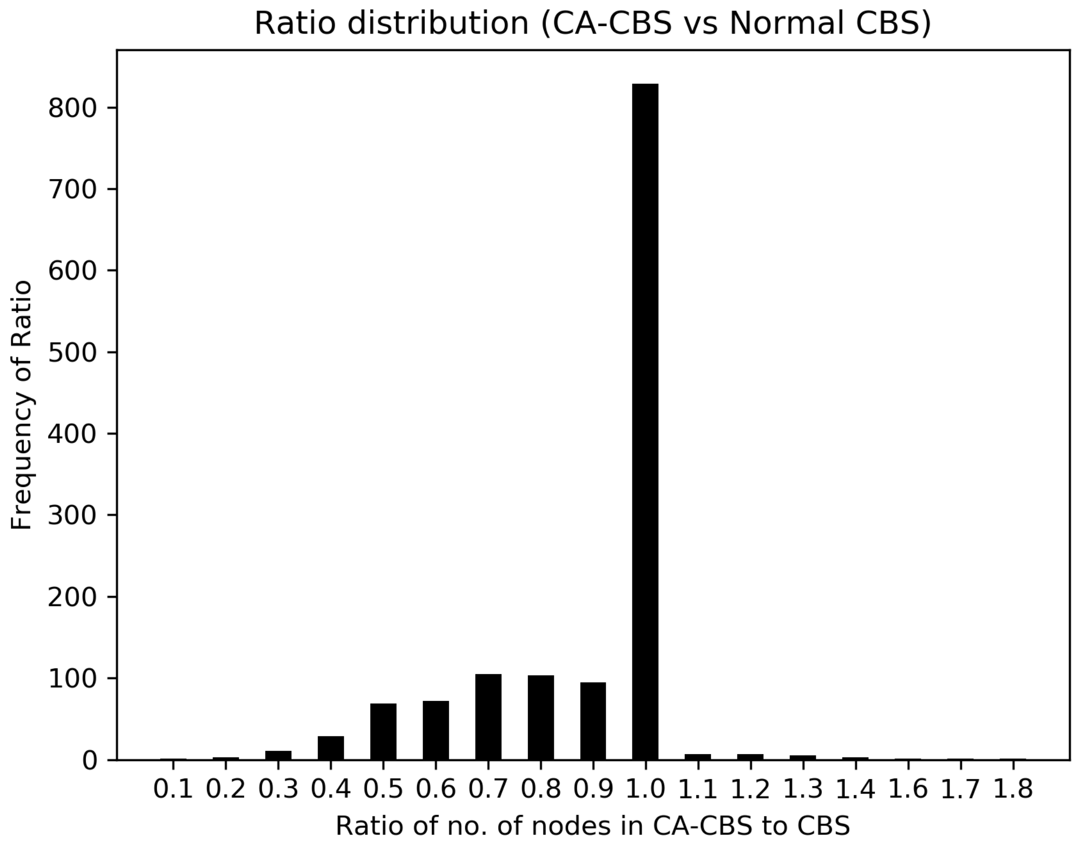

Fig. 5.6 Distribution of ratio of Nodes expanded

The distribution of Ratio of Nodes expanded in the CT for the two algorithms, gives us a clearer picture of the benefit of using CA-CBS over normal CBS. Tests were run for almost 1400 different test cases. As shown in Fig. 5.6, in 800 of them CBS and CA-CBS performed equally and in less than 20 of the test cases, normal CBS performed better. But there were almost 100 test cases for each of the CA-CBS to normal CBS improvement ratios between 0.5 to 0.9. That's a total of 500 test cases where CA-CBS performed considerable better than Normal CBS, which is approximately 35% of the total number of test cases. Also for around 60 test cases (4%), CBS took more than twice the number of nodes in the CT of CA-CBS to reach the optimal solution.

5.3 Observation

CA-CBS has been observed to perform better than CBS in almost 40% of cases. The number of nodes in the CT for CA-CBS were found to be considerably lower than CBS for a huge number of test cases and in some the number for CA-CBS was less than half that of normal CBS. Since, the algorithm takes only an added constant time, O(1), calculation as compared to the previous A* and CBS, this immediately leads to a speedup of 1.1 to 2 times the original CBS algorithm for many problem sets.

Chapter 6

Further Work

There are many challenges in path finding, especially in more sophisticated models of the real world. One of the most difficult problems among them is that of dynamic or moving objects. CBS may be used even for dynamic obstacles if the path of the object can be predicted for a considerable amount of time into the future. If the path is fixed and known prior to running the CBS algorithm, the problem is very easy. We would simply need to add the necessary constraints to avoid the path of the dynamic objects completely.

But there are two problems with this approach. First, if we cannot predict the path of the dynamic object properly for a reasonable duration into the future, we would need to run the CBS algorithm very often which is not feasible for a large number of agents. Second, the path of a dynamic object cannot be predicted with a fixed certainty, especially when those objects are animals and birds, instead of stones or balls in the air whose trajectories can be easily calculated. Objects like aeroplanes and scheduled aircrafts may change courses within the journey itself.

We would need both a good model that can predict their path atleast upto a few minutes or seconds, as well as, an algorithm that can recalculate paths to avoid collision with those moving obstacles. Even then, there might be other agents that might be following their own collision avoidance algorithms which might make it harder to predict their path as well.

This is understood by a very common real world problem of two cyclists moving towards each other. Often it's more likely both will get confused if they think too much about deciding which path to take. If both of them try to avoid paths by looking at what the other cyclist's move currently is, both tend to make hesitated changes in either direction making each other confused. Here, the best solution is to have a mutually accepted coordination rule, like every cyclist or car driver should always move to it's own left. This way the problem is avoided with ease as long as both the agents follow the rule.

Coordination is key in solving a multi agent problem. Therefore the problem of finding the most optimal path for multiple agents is hard, especially when changes in the environment are so quick and frequent. Trading off finding the optimal solution for the speed of arriving at the solution might be necessary in this regard.

Vehicles and Robots may face rapid changes in the environment at any instant, and we need to incorporate more robust models instead of naively assuming static environments which are hardly similar to the real world scenarios.

References

[GSS12] Ariel Felner Guni Sharon, Roni Stern and Nathan Sturtevant. *Conflict-Based Search For Optimal Multi-Agent Path Finding.* 2012.

[HA16] Y.K. Hwang and N. Ahuja. *A potential field approach to path planning.* 2016.

[MG16] Roni Stern Guni Sharon Jonathan Schaeffer Meir Goldenberg, Ariel Felner. *A* Variants for Optimal Multi-Agent Pathfinding.* 2016.

[Sta16] Trevor Standley. *Independence Detection for Multi-Agent Pathfinding Problems.* 2016.

[VB16] Asokan T Bhanu Chander V and Ravindran B. *A new multi-bug path planning algorithm for robot navigation in known environments.* 2016.