1-22-2020

Improving Lookahead search for grid-based pathfinding

Shrijan Karmacharya
University of Windsor

Improving Lookahead search for grid-based pathfinding

By

Shrijan Karmacharya

A Thesis
Submitted to the Faculty of Graduate Studies
through the School of Computer Science
in Partial Fulfillment of the Requirements for the
Degree of Master of Science
at the University of Windsor

Windsor, Ontario, Canada

2020

Improving Lookahead search for grid-based pathfinding

by

Shrijan Karmacharya

APPROVED BY:

_____

M. Hlynka
Department of Mathematics and Statistics

_____

S. Saad Ahmed
School of Computer Science

_____

S. Goodwin, Advisor
School of Computer Science

January 22, 2020

DECLARATION OF ORIGINALITY

I hereby certify that I am the sole author of this thesis and that no part of this thesis has been published or submitted for publication.

I certify that, to the best of my knowledge, my thesis does not infringe upon anyone's copyright nor violate any proprietary rights and that any ideas, techniques, quotations, or any other material from the work of other people included in my thesis, published or otherwise, are fully acknowledged in accordance with the standard referencing practices. Furthermore, to the extent that I have included copyrighted material that surpasses the bounds of fair dealing within the meaning of the Canada Copyright Act, I certify that I have obtained a written permission from the copyright owner(s) to include such material(s) in my thesis and have included copies of such copyright clearances to my appendix.

I declare that this is a true copy of my thesis, including any final revisions, as approved by my thesis committee and the Graduate Studies office, and that this thesis has not been submitted for a higher degree to any other University or Institution.

ABSTRACT

Pathfinding is an essential part of navigation systems, often used in video games, route planning and robotic navigation. A* search has been one of the most well-known and frequently used algorithms for pathfinding. A* uses an $open\ list$ and a $closed\ list$ to keep track of all nodes generated and expanded. The size and performance of these data structures are major drawbacks of A*. Lookahead is used to investigate future outcomes and improve the quality of available choices. Lookaheads are done on a DFS manner from the frontier of A* search. This combination of A* and DFS lookahead has been shown to save space when working with puzzles. We leverage this concept with grid-based pathfinding in video games to save the amount of space consumed. However, because grids contain redundant paths, the DFS lookaheads end up being an overhead as they do not maintain a list of nodes visited or expanded. By using a domain-specific pruning technique, we significantly improve the time taken by the algorithm and further improve upon the space consumed. A combination of lookahead and A* search with this pruning technique is, therefore, able to achieve improvement in both space-consumed and time-taken over the standard A* search algorithm for grid-based pathfinding.

DEDICATION

To my beloved family:

Parents: Sushil Karmacharya & Jasoda Karmacharya

Siblings: Shrijeet Karmacharya

ACKNOWLEDGEMENT

I would like to express my sincere gratitude to my advisor Dr. Scott Goodwin for his advice, support and encouragement throughout my graduate studies. I am grateful for his immense knowledge of the domain, time and patience, without which my master's degree would be incomplete.

I would also like to thank my committee members Dr. Sherif Saad and Dr. Myron Hlynka for their guidance and support to improve this thesis.

I would like to offer a sincere thanks to all faculties and staff at the School of Computer Science for their assistance and support.

Finally, I would like to thank my family and friends for supporting me over the course of my academic career.

TABLE OF CONTENTS

TABLE OF FIGURES

LIST OF TABLES

LIST OF EQUATIONS

CHAPTER 1:

Introduction

1.1 Thesis Claim

A* with lookahead is a variant of A* search that performs limited DFS lookaheads from the frontiers of A*. This algorithm saves space by using DFS lookaheads which is linear compared to the exponential nature of A*. We claim that this scheme works well in a grid-based domain for saving space. However, as paths in grids are highly redundant, DFS lookaheads tend to expand an exponentially large number of nodes at each iteration slowing the speed of the algorithm considerably.

We then, propose the use of a domain-specific search space reduction technique, which prunes the number of children generated at each level based on the direction of the search. Using this pruning technique, we achieve speeds comparable to or better than the standard A* search. A combination of these two techniques provides improvements in both space-consumed and time taken over the standard A* algorithm in a grid-based path-planning domain.

1.2 Pathfinding

Pathfinding plays a significant role in graph search problems wherein a path is found based on certain criteria between nodes in the graph. This criterion often corresponds to a positive result of some kind (cheapest, fastest, best) in the problem domain from which the graph was derived from. The pathfinding/path-planning problem can be used to model problems in different domains like solving puzzles, optimizing task scheduling, operations research,

and routing in computer networks and computer games. (Norvig, 2010)Therefore, pathfinding remains an active area of research in the Artificial Intelligence domain.

Pathfinding holds a special place in Video game AI. Real-Time Strategy Games (RTS), Role Playing Games (RPG) and Multiplayer Online Battle Arena (MOBA) heavily depend on pathfinding either as a component of a Non-Player Character (NPC) or as a component of the Player. This entity that benefits from the results of pathfinding is known as an Agent. Depending on the number of agents, the pathfinding problem can be divided into Single-Agent Pathfinding or Multi-Agent Pathfinding.

A generic pathfinding problem formulation for a video game is as follows:

a. The game environment is the state space,
b. The start and goal node are locations in the game environment,
c. The unit utilizing the path generated is the Agent

1.2.1 Graph Representation

Game environments or maps are represented as a graph in one of three ways: Grids, Navigation Meshes or Waypoints. Each of these is a simplified representation of the search space.

1.2.1.1 Grids

Grids are the most frequently used representation of game environments. Grids are a uniform subdivision of the state space into tiles. Each tile in the grid can either be traversable or untraversable. Furthermore, traversable grids can have different costs

associated with it depending on the type of terrain on the map. Subdivisions for grids are divided based on tiles.

The most common grid types are square, triangle and hexagonal (Patel, 2010). In a grid-based map representation, each tile represents a node. For each neighbor of a tile, there implicitly exists an undirected edge from that tile. The number of outgoing edges a node has depends on the number of neighboring tiles it contains, which depends on the movement allowed on the grid. For example, for a square grid with 4 adjacent tiles, if only straight movements are allowed (NWSE directions) then it has 4 neighbors. If diagonal movements are allowed on top of a straight movement, then the tile has 8 neighbors. This map representation is used for all experiments done for our thesis.

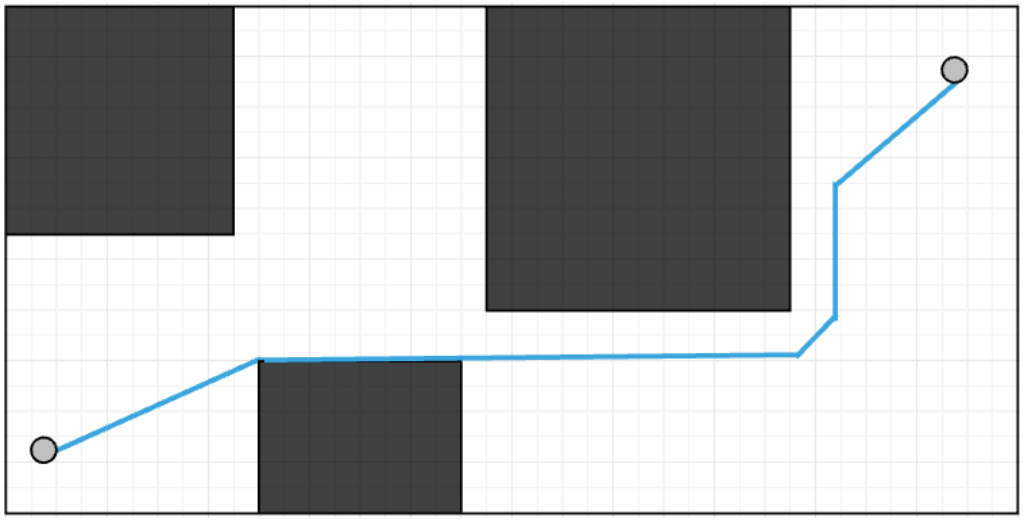


Figure 1: Grid Representation

1.2.1.2 Navigation Mesh

Navigation Mesh or Navmesh, are graphs where the traversable areas are represented as interconnected polygons. Obstacles are not part of the state space in a navigation mesh. Each polygon in Navigation Mesh can have different weights associated with them. Agents

in Navigation Mesh can travel within the polygon without having to worry about obstacles usually trivially as a straight line (Patel, 2010) (P.Mehta, 2015). Adjacent polygons of a Navigation Mesh are connected to each other as a graph.

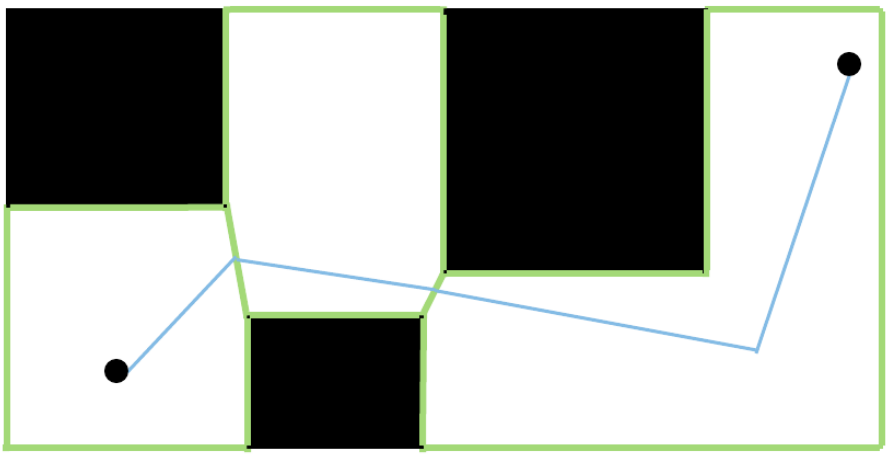


Figure 2: Navigation mesh representation

1.2.1.3 Waypoints

Waypoints are the final method of representation of game maps. They consist of nodes that are placed at a location in the graph (P.Mehta, 2015) (Patel, 2010). Waypoints can be set by the player or by game designers. Waypoints added by game designers are often seen as landmarks on the map (Patel, 2010). Waypoints set by players are more common in RTS and MOBA games. Waypoint set by game designers is common in Role Playing Games or games that trigger in-game events (Nareyek, 2004).

The waypoints generated by the player and the waypoints set by programmers are usually not along an optimal path therefore the path generated using waypoints can be sub-optimal too. Similarly, the same waypoints cannot be used across different start and goal nodes.

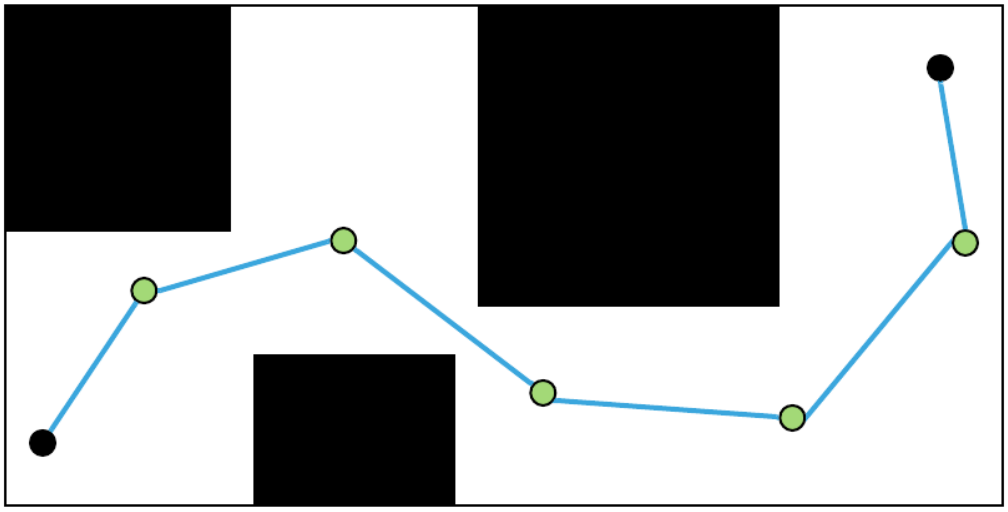


Figure 3: Waypoint Representation

1.2.2 The general graph search structure

For our thesis, we establish that our state space is represented as a grid map. Each node is a tile and each tile has octile movement (Straight + Diagonal). Each node in the grid has 8 neighbors connected by a bidirectional edge or an undirected edge where movement between the edges in either direction is allowed. These edges are the actions in our state space. We shall go into this in detail later when we describe optimizations and rules applicable to a grid.

The key infrastructure for all graph-based search algorithms are,

1. Nodes:

Nodes are data structures in search algorithms that hold the state, its parent, and various other details pertinent to the search algorithm.

2. $open\ list$/ frontier/ generated nodes

The $open\ list$ is a list-like data structure that holds multiple nodes. Minimally, this data structure allows nodes to be added and removed from it. The nodes held here are nodes that have the potential to be in the solution path found by the pathfinding algorithm. The implementation of an $open\ list$ differs between the type of search algorithm being used.

3. $closed\ list$/ explored set/ expanded nodes.

The $closed\ list$ is a data structure similar to the $open\ list$. The nodes in the $closed\ list$ are nodes that have already been visited by the search algorithm. The nodes in the closed list are part of the path found. The $closed\ list$ needs to be designed so that data in a $closed\ list$ can be read without overhead. The $closed\ list$ is usually implemented using hash tables because its lookup has $O(1)$ time complexity.

A general graph-based search algorithm has the following scheme,

```
Input: Graph(V, E) with start node start and goal node goal
Output: path from start to goal
function graph − search:
Initialize open with start
Initialize closed with empty
while(open is not empty)
        use a search strategy to select node from open
        add node to closed
        expand chosen node, adding its children to open, if not
        in open or closed
```

The general graph-based search algorithm scheme starts with initializing the *open list* with start node for pathfinding. The start node consists of the start location as its state. It empties the *closed list*. It then loops through the *open list* selecting one node to expand. All children of the nodes expanded are then added to the *open list*. All famous graph-based search algorithms differ from each other only in its Search Strategy. A search strategy is a process selecting a node from the frontier or *open list* to expand and then moved to the *closed list*. The data structure used for the *open list*, therefore, depends on the algorithm and how it helps optimize the search strategy this algorithm uses. (Norvig, 2010)

1.2.3 Search Algorithms

The solution to a pathfinding problem is usually found using a search algorithm. A general search algorithm consists of the node, its children, a list of children that were previously generated/frontier of the search/ *open list* and a list of nodes that were previously expanded/visited nodes/*closed list*. (Norvig, 2010)

There exist different search algorithms, however, they can be classified into two major categories: Informed Search and Uninformed Search. Uninformed search algorithms are those that do not integrate domain knowledge into the search strategy. Informed search, on the other hand, makes use of domain-specific knowledge, and integrates it into the search strategy. (Norvig, 2010)

Breadth-First Search, Depth First Search, and Uniform Cost Search are well-known Search Algorithms. Breadth-First Search uses the shallowest node first search strategy for node selection. Depth-First Search uses the deepest node first strategy for node selection. Uniform Cost search uses a node with the lowest path cost first $g(n)$ as its search strategy. If the path cost for between each node and its child is constant, then Uniform Cost Search is the same as Breadth-First Search. In literature, Uniform Cost Search is also known as Dijkstra's for a single goal node (Holte, 2010).

Best First Search and A* Search are the most well-known informed search algorithms for Pathfinding. Both algorithms make use of a heuristic function $h(n)$, which uses domain-specific knowledge to drive the search strategy. Best First Search solely relies on the heuristic function as its search strategy (Norvig, 2010) whereas, A* search uses a combination of path cost and heuristic function as its search strategy (Hart, Nilsson, & Raphael, 1968).

1.2.4 Performance Measures for pathfinding

Like with problem-solving, there are various measures to evaluate the algorithms in pathfinding.

- Completeness: Completeness of an algorithm evaluates if the algorithm is guaranteed to find a path if there is a path to the goal.
- Optimality: Optimality checks if the solution found by the algorithm is optimal. For pathfinding, it checks if the path generated or found by the algorithm is the shortest path.
- Time Complexity: Like all algorithms in the field of Computer Science, the performance of the algorithm is evaluated in terms of its time complexity or time taken. As pathfinding exists as a subfield of AI, the time complexity of the pathfinding algorithms is measured in terms of the effective branching factor b_e and the shallowest depth of the solution, d.
- Space Complexity: Essentially means how much space is consumed by the algorithm while it finds the solution. Like the time complexity of the algorithms, the space complexity is measured in terms of the effective branching factor b_e and the shallowest depth d of the solution. It is usually computed as the nodes stored in memory i.e. the node generated. Space complexity is also a common measure of performance in Computer Science.

1.3 Thesis Contribution

The space complexity of A* search is exponential $O(b^d)$ in nature. The Depth-First Search algorithm on the other hand, has a linear space complexity of $O(d)$ for its tree search variant. A* search uses a combination of heuristics and the cost of the path to create an efficient search strategy. The path found by A* search is optimal whereas, DFS may or may not find any path to the goal. Combining the two schemes, we can leverage the space

complexity of DFS to improve A* search's space complexity. On top of this, returning the cost of the frontier to A* helps improve the performance of the heuristic search. The combination of this scheme called A*L or A* with Lookahead has been shown to be efficient in the puzzle domain (Roni Stern, 2010).

Using a combination of this scheme for grid-based pathfinding is slightly more difficult. As grids are notorious for having highly redundant paths (Daniel Damir Harabor, 2011), a tree search based DFS lookahead will cause significant overhead. This overhead will overshadow any space-based improvements that the A*L can provide. We propose using a neighbor pruning algorithm specific to the grid domain. This algorithm reduces redundant and cyclic paths when used in DFS and symmetric paths when used in A* search. Using this pruning technique, A*L becomes viable as an option in the grid domain, showing improvements in both time and space compared to the standard A* search.

1.4 Thesis Organization

This thesis is divided into five major chapters. The first chapter introduces the basic concepts that will be used throughout the rest of the chapters. Alongside introductory concepts, it also provides key underlying concepts for our work. The second chapter goes into detail about the major algorithms and concepts, as well as details into key literature that motivated research into this topic. The third chapter covers the proposed methodology and algorithm for this thesis. It goes over techniques that lead to an improvement in the algorithm. The fourth chapter describes the experimental setup, results of the experiments and analysis of the results. The fifth chapter offers a conclusion and key findings alongside future research into the topic. Appendices consist of tables with data from our experiments.

CHAPTER 2:

Background and Literature Review

In this section, we start by introducing the A* search algorithm. We explore the concepts that are relevant to the A* search algorithm. Then, we define different types of heuristics and how they relate to the grid domain. After that, we look at recent literature relevant A*L algorithm. We explain the algorithm and the key concepts behind it.

2.1 The A Search*

The A* search algorithm is the most popular algorithm for pathfinding problems. Because A* uses heuristics to guide the search, it is an informed search algorithm. For a certain Graph G with a Start node and a Goal node, A* search finds an optimal path from the $start$ node to the $goal$ node. The problem solved by A* is a minimum cost problem, therefore, returning the shortest path from the $start$ node to the $goal$ node.

A* builds a search tree from the state space by expanding nodes. A* begins by adding the $start$ node into the list of frontiers ($open\ list$). The algorithm keeps looping through the $open\ list$, until either the $open\ list$ is empty or if the node selected for expansion is the $goal$ node (Norvig, 2010).

As mentioned before, the process of selecting a node to expand from the $open\ list$ is known as the search strategy. There are three major parts in A* search's strategy.

1. $g - cost$ or $g(n)$ is the actual cost of the path from $start$ node to node n. The $g(start)$ is equal to 0 and the $g(goal)$ is equal to the length of the path.

2. $h - cost$ or $h(n)$ is the heuristic estimate of cost from node n to the $goal$ node. The $h(goal)$ is equal to 0.
3. $f - cost$ or $f(n)$ is an evaluation function and is the summation of $g - cost$ and $h - cost$ the node. It is represented by the formula,

$$f(n) = g(n) + h(n) \quad (1)$$

Equation 1: A evaluation function f*

$A*$ selects the node with the lowest $f - value$ from among all the nodes in the $open\ list$ as the next node to be expanded. Before expansion, A* places the node selected into the list of nodes that have already been visited or $closed\ list$. It proceeds to expand the node by generating all the node's neighbors. All the generated neighbors are then evaluated and placed into the $open\ list$.

In the algorithm, $A*$ selects nodes from the $open\ list$ to expand. It calls the node that is to be expanded as $current$. The node $current$ is placed into the $closed\ list$. If the current node is the goal node, $A*$ returns the goal node. The optimal path can be built by recursively generating the parents, from $goal$ node to the $start$ node.

All neighbors from the current node are then expanded in the algorithm. Each neighbor in the for loop is then designated as the $neighbor$ node. $newNeighborCost$ is a temporary variable that stores the cost of path taken from $current$ node to the $neighbor$ node. A* then checks if the neighbor already exists in the $closed\ list$. If the neighbor already exists in the $closed\ list$ and the cost of path taken from $current$ node to this $neighbor$ is less than that compared to the path it took when it was visited, it removes $neighbor$ node from the $closed\ list$. If the current path is longer or of same length as the previous path, then it ignores the $neighbor$ node.

```
Input: Graph G, start node start and end node goal
Output: Least Cost Path from start to goal
function Astar()
        open.add(start))
        closed.clear()
        g(start) = 0
        h(start) = heuristic_function(start, goal)
        f(start) = g(start) + h(start)
        while open is not empty:
                current = open.pop() //Node at top of OPEN with lowest F value
                if current = goal:
                        return goal
                open.remove(current)
                closed.add(current)
                for each neighbor of current:
                        newNeighborCost = g(current) + distance(current, neighbor)
                        if neighbor in closed:
                                if g(neighbor) ≤ newNeighborCost:
                                        continue
                                else:
                                        closed.remove(neighbor)
                        if neighbor in open:
                                if g(neighbor) ≤ newNeighborCost:
                                        continue
                         open.add(neighbor)
                         g(neighbor) = newNeighborCost
                         h(neighbor) = heuristic_function(neighbor, goal)
                         f(neighbor) = g(neighbor) + h(neighbor)
                         neighbor.parent = current
        return null
end
```

Figure 4: A Search*

A* search checks if the $neighbor$ exists in $open\ list$. Like with $closed\ list$, if the $neighbor$ already exists in $open\ list$, and the cost of path taken from current node to this $neighbor$ is less than that compared to the path it took when it was generated previously, then it removes this node from the $open\ list$. If the current path is longer or of same length as the previous path, then it ignores the neighbor node.

The neighbor is then evaluated, wherein, it's $g-cost$, $h-cost$ and $f-costs$ are computed. The neighbor is then added into the $open\ list$ as a possible candidate to be

expanded next. When a neighbor in A* is removed from $closed\ list$, for it to be re-evaluated and added to the $open\ list$ again, the node is said to be re-expanded.

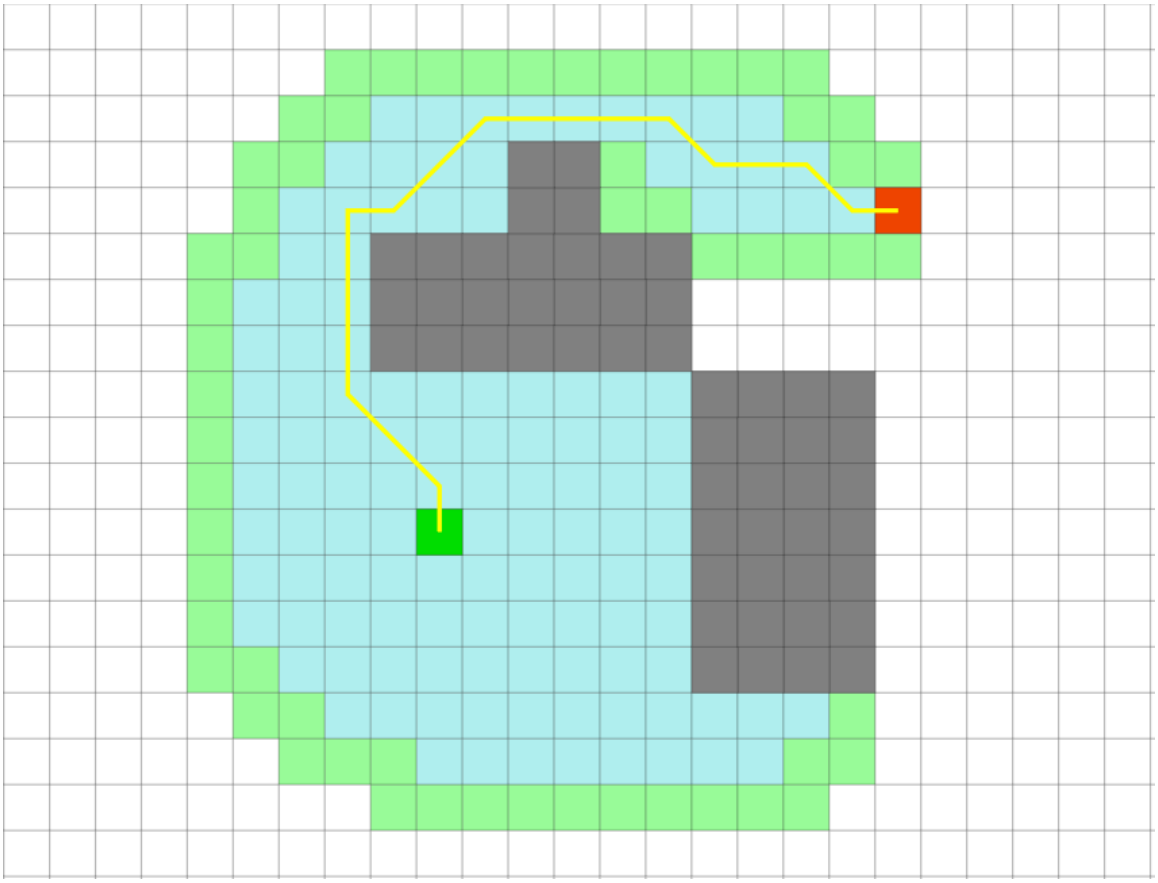


*Figure 5: Pathfinding using A**

2.1.1 Constraints on Heuristics

The A* search as an optimal path algorithm, works, only when certain conditions are met. These conditions are constraints on top of heuristics. ***Admissibility*** of a heuristic function guarantees that the algorithm finds an optimal path if there exists one. A heuristic function is called admissible if it never overestimates the cost to reach goal (Hart, Nilsson, & Raphael, 1968). In A*, the heuristic $h(n)$ is said to be admissible, if it would never exceed

the actual cost to reach the goal node from any node n. If we assume, $h(n)*$ as the actual optimal cost to reach the goal from node n,

$$h(n)\ is\ admissible\ if,$$

$$\forall n\ h(n) \leq h^*(n) \tag{2}$$

Equation 2: Admissibility of heuristic

Consistency is the next constraint on the heuristic function. A heuristic function $h(n)$, is said to be consistent for node n if the estimate for the node is less than or equal to the sum of the cost of the path from the node n to its children and heuristic estimate of its children. This condition is also called monotonicity. Consistency is a stricter condition than Admissibility (Norvig, 2010). A consistent heuristic is also admissible, therefore, any heuristic that is consistent guarantees that the path found by A* search is optimal.

Consistency has another consequence in A* search. If we look at the algorithm provided in this thesis, there are conditions for when a neighbor needs to be checked if it already exists in $open\ list$ or $closed\ list$. When the heuristic is consistent, then it guarantees that every node chosen for expansion will never be re-expanded or updated in the $open\ list$. Formally, consistency is defined as,

$$\forall n, n'\ h(n) \leq cost(n, n') + h(n') \tag{3}$$

Equation 3: Consistency of heuristic

Where, n' is a child of n and $h(goal) = 0$.

2.2 Heuristics

A heuristic function is used to incorporate domain knowledge into search algorithms. The heuristic function can be either used in conjunction with state-space knowledge or on its own to derive a novel search strategy. The use of a heuristic function is what separates Informed search algorithms from Un-Informed search algorithms. Usually, a heuristic function is denoted as h, and for any node n, the heuristic value is an estimate from the node n to the goal, denoted as $h(n)$. For A* search, if the heuristic value $h(n)$ is set to 0, it turns into Uniform Cost Search, if *OPEN* does not have any other path that is less than the current path cost (Holte, 2010). A good heuristic function usually helps improve search by reducing the number of nodes expanded (Norvig, 2010) (Korf, 2000).

Pathfinding problem in a grid-based environment means finding the shortest path from one point in the grid to another. For square grid-based maps, there are two well-known heuristic functions, the Manhattan distance and the Euclidean distance.

2.2.1 Manhattan Distance

Manhattan Distance, or city block distance is the distance between two points in which the movement is only either vertical or horizontal. For a coordinate system (x, y), the Manhattan distance between two points $A\ (x_1, y_1)$ and $B\ (x_2, y_2)$ is calculated as sum of the absolute differences in the $x - coordinate$ and the $y - coordinate$. It is given by the formula,

$$h(A, B)_{Manhattan} = |x_1 - x_2| + |y_1 - y_2| \quad (4)$$

Equation 4: Manhattan Distance

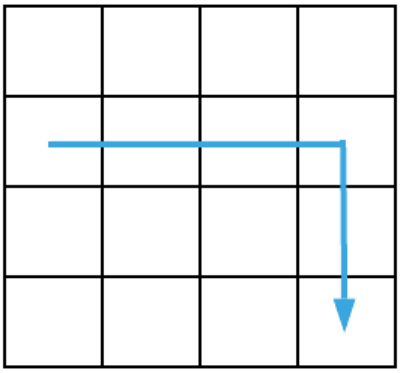


Figure 6: Manhattan Distance

On grid-based maps, Manhattan distance is often considered a standard heuristic. When used in grids where only straight movements are allowed, A* search with Manhattan Distance can find the optimal path. However, when used in grid maps where diagonal movements are also allowed, the Manhattan Distance can result in sub optimal solutions. This is because Manhattan Distance will overestimate the cost of path for diagonal movement, making the heuristic inadmissible.

2.2.2 Euclidean Distance

Euclidean distance is the straight-line distance or the airline distance between two points. For a coordinate system (x, y), the Manhattan distance between two points $A\ (x_1, y_1)$ and $B\ (x_2, y_2)$ is calculated as the root of the squared difference between respective $x - coordinates$ and $y - coordinates$. It is given by the formula,

$$h(A, B)_{Euclidean} = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \quad (5)$$

Equation 5: Euclidean Distance

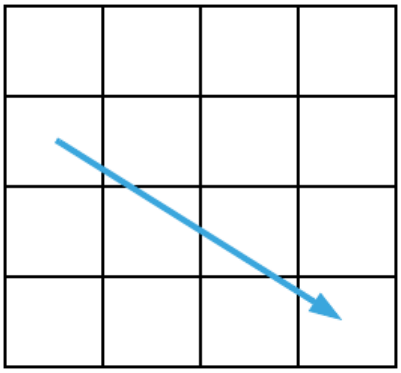


Figure 7:Euclidean Distance

Euclidean distance is more expensive to compute than Manhattan distance. However, regardless of the movement allowed, i.e. Straight only, straight with diagonal or any directional movement, when Euclidean Distance is used as the heuristic function, A* will be able to find an optimal path. For diagonals, the Euclidean distance will still underestimate or be equal to the cost of path, guaranteeing admissibility.

2.3 A with Lookahead (A*L)*

The major issue with A* search is the memory requirements of A*. A* is more likely to run out of memory far before the time taken becomes an issue (Norvig, 2010). A* needs to store all nodes in an $open\ list$ and $closed\ list$. The space complexity of A* is exponential in nature.

$Depth\ first\ Search$, on the other hand, has memory requirements that are mostly linear as it needs to store the branch it currently is working with. The version of DFS we are talking about here is the Tree-Search version of DFS. The flaw of this version of DFS is that it is not complete if redundant paths exist, i.e. DFS fails to find a solution even when

there exists a solution. It is very less likely that DFS will ever find an optimal path when using the Tree-Search Version of DFS, except in a very rare circumstance (first node to be generated is always along the optimal path).

Combining DFS and A* search would allow both algorithms to benefit. While there exist algorithms, IDA*, which combine these two schemes, they often have problems with needing to explore the same nodes repeatedly. The memory complexity of IDA* is linear similar to DFS. In conjunction with its low memory constraints, the path resulting from IDA* is optimal in nature, however, because IDA* expands the nodes at certain depth repeatedly, for graphs with multiple redundant paths, IDA*'s time taken to find an optimal path is very large.

Lookaheads with $A*$ or $A*L$ is an algorithm proposed by Stern et al. which combines the scheme $A*$ search and doing depth first search lookaheads from the nodes being generated. The lookaheads are bound either by depth or by cost. For the experiments in the paper, either one would work as they work with a puzzle domain where the cost of each action is

The key variables of the algorithm are,

UB – is the upper bound on cost of children to be expanded by $A*L$. UB is equal to the cost of the best solution found so far. Unlike BRFSL, $A*$ needs to expand all children and perform lookahead for all nodes for which f-value is less than the current best solution (UB). Once UB is set i.e. it is not infinity, any child with cost greater than or equal to UB can be pruned. The children are goal tested and if the child is a goal node UB is updated.

LHB – is the lookahead bound which helps set bound on DFS lookahead. It is the lowest value among the current UB, $f-score$ of expanded node $f(current)+k$.

Lookahead cost k – is the value which is used to limit the lookahead either through cost or through depth

h_u – is the updated heuristic value after lookahead is done

$f_{updated}$ – is the updated f value where,

$$f_{updated} = h_{updated}(neighbor) + g(neighbor)$$

$bestpath$ - is the stack that holds the best available path from the frontier node.

```
Input: node current to be expanded, UB upper bound variable
if f(current) ≥ UB:
        break;
for each neighbor of current:
        newNeighborCost = g(current) + distance(current, neighbor)
        if neighbor == goal:
                UB = Min(UB, newNeighborCost)
        if newNeighborCost > g(neighbor) or f(neighbor) ≥ UB:
                continue
        if f(current) == f(neighbor):
                f_updated(neighbor) = f_updated(current)
                continue
        LHB = Min(UB, f(current) + k)
        if f(neighbor) ≤ LHB:
                Mincost = ∞
                Lookahead(neighbor, LHB, UB, Mincost)
                h_u(neighbor) = Min(h_u(neighbor), Mincost - g(neighbor))
        else:
                h_u(neighbor) = h(neighbor)
        f_updated(neighbor) = h_u(neighbor) + newNeighborCost
        if f_updated(neighbor) ≥ UB
                continue
Add, update open as needed
```

Figure 8: A ∗ L expansion cycle

The algorithm above is a modification on top of $A*$ after a node selected is expanded. The Search Strategy for this algorithm is to select the node with lowest $f_{updated}$ value. Therefore, the $open\ list$ is sorted using the $f_{updated}$ value. Like with $A*$ search,

$newNeighborCost$ is a variable that stores the cost taken to reach this node $neighbor$ while going through $current$. If the neighbor generated is equal to the goal node, Upper Bound variable UB is set as the lesser of $newNeighborCost$ and previous UB.

The algorithm prunes neighbors whose $f\ values$ exceed that of UB or if the cost to the neighbor through this path is not the lowest cost to this path. The Lookahead Bound variable, LHB is then set as the lesser of UB or expanded node's $f - score$ + lookahead value k, $f(current) + k$.

If the $f - score$ of neighbor is less than the Lookahead bound LHB, then depth-first search lookaheads are performed from the neighbor node until the frontier that is exceeds the LHB. The $MinCost$ variable is first initialized to infinity, then used to store the minimum $f - cost$ from all the frontiers. The updated heuristic h_u is set as either the previous h_u or $MinCost - newNeighborCost$, whichever is lower. After this, the algorithm works like A* where it inserts into the $open\ list$ or updates the $open\ list$ and $closed\ list$ (re-expansion).

```
Input: node current for lookahead, LHB lookahead bound variable, UB upper bound
variable, Mincost minimum cost of lookahead to be returned
function lookahead(current, LHB, UB, Mincost):
    S.push(current)
    for each neighbor of current:
        if neighbor == goal:
            if g(neighbor) < UB :
                bestpath = S
            UB = Min(UB, g(neighbor))
            Mincost = Min(Mincost, g(neighbor))
        else:
            if f(neighbor) ≤ LHB and f(neighbor) < UB:
                Lookahead(neighbor, LHB, UB, Mincost)
            else:
                Mincost = Min(Mincost, f(neighbor))
    S.pop()
```

Figure 9: Lookahead portion of $A*L$

The lookahead part of the algorithm is a recursive Depth-First Search that is bounded by the lookahead value LHB. Current node to be expanded is stored in the stack S and popped after all nodes have been generated. $neighbor$ nodes are generated from the $current$ node and evaluated. If a goal is found during the lookahead stage, the UB variable is updated based on neighbor's $g\ cost$, $g(neighbor)$. The stack S is saved as the best cost path to the goal if the current path is the best path found to the goal. This is either a min of the previous UB or neighbor's $g\ cost$ to guarantee optimality of the algorithm. $Mincost$ is also updated based on the neighbor's $g\ cost$.

When the neighbor is not the goal, lookaheads are performed recursively until the $f - score$ of current neighbor, $f(neighbor)$ is greater than the LHB or UB. When the $f - cost$ exceeds either the UB or the Lookahead bound LHB, $Mincost$ is set as minimum of current $Mincost$ and $f - score$ of the neighbor in the frontier.

CHAPTER 3:

Proposed Methodology

3.1 Motivation

The initial motivation for the work was when doing lookaheads from a node n and returning the minimum cost from frontiers, we might be able to ignore certain branches and thus save space by reducing the number of nodes generated and expanded.

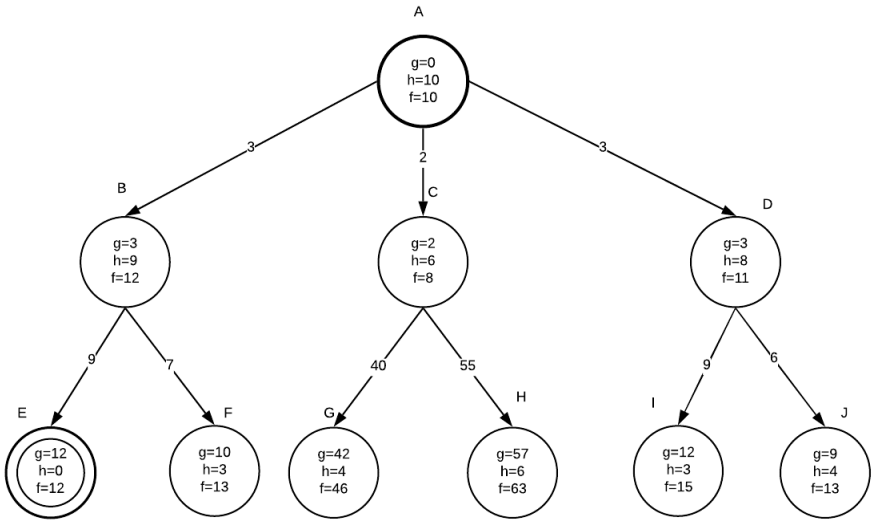


Figure 10 : A Search Tree*

Let us look at a sample search tree for $A*$ search as shown above. For the sample, let us assume that the start node is A and the goal node is E. $A*$ adds A to the $open\ list$. It selects and expands Node A generating 3 children, B, C, and D respectively. The $f-score$ for all the children is computed and added to the $open\ list$, while node A is added

to the list of expanded nodes or $closed\ list$. The node with the lowest $f - score$, node C is selected for expansion. When node C is expanded, nodes G and H are generated and evaluated as node C is added to the $closed\ list$. Node D, with $f - score$ 11 is then selected for expansion from the $open\ list$. Nodes I and J, which are the children of node D are generated and node D is added to the $closed\ list$. Node B is expanded next with children E and F, while B is added to the $closed\ list$. Node E is selected for expansion next, and since Node E is the $goal$ node, it is added to the $closed\ list$ and the search ends. We look at the nodes stored in $open\ list$ and $closed\ list$ at the end of the program.

$open\ list$ = {J, F, I, G, H}

$closed\ list$ = {A, C, D, B, E}

The total number of nodes stored in $open\ list$ and $closed\ list$ are 10.

We look at how the same search tree is evaluated with lookahead at depth of 1. For every node to be generated, a lookahead search is done up to depth 1 i.e. its children. Among the frontier of the lookahead nodes, the node with lowest $f - score$ is returned to $A*$ and the child from which lookahead was performed, will have its updated $f_{updated}$ set as the value returned from lookaheads. The start node A and the goal node E remain same as the previous example.

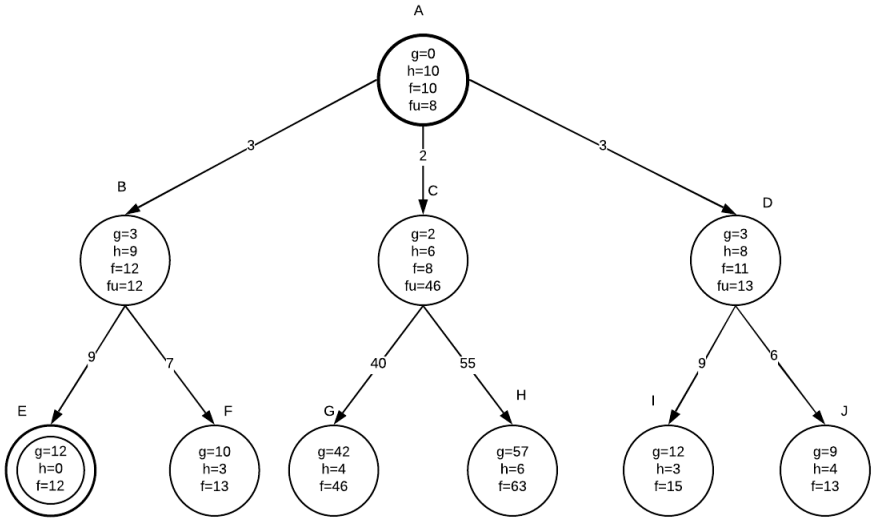


Figure 11 : A with Lookahead Search Tree*

Node A is first added to the $open\ list$. A lookahead is done for A as all its children B, C and D are generated but not stored anywhere yet. The node with the lowest $f-score$ among them is node C, which has a value of 8. This value is propagated back to A with cost 8. This step of generating the $f_{updated}$ for the root node may be skipped as the root node is added to the closed list regardless. And if the root node is the goal node, then the search does not need to take place.

After node A is selected for expansion, its children are generated. For each child generated, a lookahead search is performed to a depth of 1. Lookahead from node B generates E and F, with E having the lowest $f-score$. Node B's $f_{updated}$ is set to 12. Similarly, lookahead from node C, results in $f-score$ of 46 being propagated back to node C. Lookahead from node D results in $f-score$ of 13 being propagated back. The diagram above shows the respective $f_{updated}$ values of the nodes represented by "fu". Node B is expanded because it has the lowest $f_{updated}$ and lookahead for its children are performed.

Since E is the goal node, there is no more lookahead performed for E. Unless lookahead from F results in a $f - value$ lower than 12, node E is set for expansion. Since node E is the goal node and there are no nodes that have an $f_{updated}$ value less than E the search concludes. We look at the $open\ list$ and the $closed\ list$ for this search.

$open\ list$ = {C, D, F}

$closed\ list$ = {A, B, E}

We see that the total number of nodes stored in $open\ list$ and $closed\ list$ have now decreased to 6.

Improving the $f_{updated}$ value is not the only place where $A * L$ can save space. The $A * L$ algorithm described in the earlier section re-uses the lookaheads to prune out all nodes that have a $f - value$ greater than the upper bound variable UB. Let us look at the same example but using $A * L$ algorithm. We do a lookahead up until a depth of 1. Our objective remains the same, that is to find the path from start node A to goal node E.

Node A is expanded like before and the $f_{updated}$ is set as 8. Its three children are generated starting with B. When a lookahead is done from B, it finds the goal node E. The goal node E now sets the upper bound variable UB to 12. Since both nodes C and D have a $f_{updated}$ greater than the UB, they are both pruned and never added to the $open\ lis$t. Therefore, the open and closed lists look as follows at end of the program.

$open\ list$ = {B}

$closed\ list$ = {A}

The total number of nodes stored is 2. We should, however, note that the nodes that were generated and discarded during the lookahead stage were 9. While these do not affect the memory consumed, they do affect the time taken to run the algorithm.

3.2 The grid domain

The grid-based map representation is used very often in video games. The grid-based representation exists for different types of games, RPGs like Dragon Age and RTS like Starcraft. There exists a compilation of standard benchmark maps from these games. These benchmarks have the following characteristics. All maps are represented as 2D grids. The maps are octile in nature. Therefore, the movements allowed on these maps are straight and diagonal. All tiles, therefore, have eight neighbors except the ones on the boundary of the map. All trees, walls and unpassable terrain are considered as obstacles and are untraversable. Unpassable terrain adds another constraint to these benchmark maps. Because units occupy space, it should not be possible for them to move through an obstacle. This applies if a diagonal movement is to be made between tiles but there exists an obstacle adjacent to the parent tile in one of the straight directions. This is called corner-cutting and is disallowed in these benchmark maps (Sturtevant, 2012).

Figure 12 : Example of corner cutting in left and proper path for agent on right

Each Straight movement on the map has a cost of 1 and diagonal movement has a cost of $\sqrt{2}$. When grids have cost between neighbor tiles as defined above it is called, a uniform-cost grid.

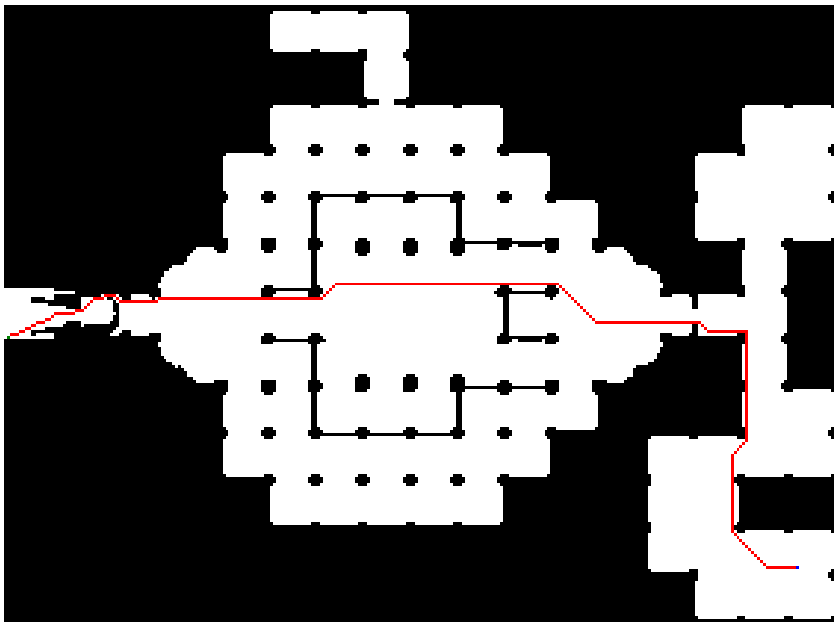


Figure 13 : A Search on a standard map*

3.2.1 Path Symmetry

Uniform cost grids are special form of graphs because they consist of many redundant paths. Along with redundant paths they also contain what is called a symmetric path (Daniel Harabor, 2011). While graph search algorithms have option of not exploring redundant paths using *open list* and *closed list*, tree-search algorithms do not have the same option. The DFS lookahead part of the algorithm A*L uses the tree-search version of DFS. If we used Graph Search version, we would not be able to save as much memory as we need to keep track of visited nodes and generated nodes (Norvig, 2010). Furthermore, regardless of the type of algorithm path symmetry cannot be avoided by standard search algorithms.

Multiple paths can be defined as symmetric, if for a pair of start and end nodes, there exist multiple paths with the same path cost (Daniel Damir Harabor, 2011). Path symmetry forces search algorithms to evaluate equivalent states (Daniel Damir Harabor, 2011). Paths are symmetric if the edges between them or direction of movement between them are a permutation of each other (Daniel Harabor, 2011).

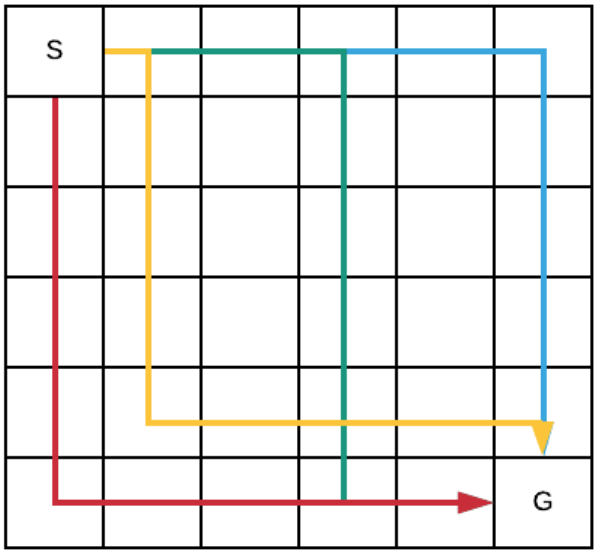


Figure 14: Path Symmetry

Path symmetries are easier to visualize when the movement of the graph is limited to straight movement. We can see that all paths are optimal with a cost of 10. Likewise, if $A*$ search is performed with Manhattan heuristic on the graph all nodes will have a $f - score$ of 10.

0 10 10 S	1 9 10	2 8 10	3 7 10	4 6 10	5 5 10
1 9 10	2 8 10	3 7 10	4 6 10	5 5 10	6 4 10
2 8 10	3 7 10	4 6 10	5 5 10	6 4 10	7 3 10
3 7 10	4 6 10	5 5 10	6 4 10	7 3 10	8 2 10
4 6 10	5 5 10	6 4 10	7 3 10	8 2 10	9 1 10
5 5 10	6 4 10	7 3 10	8 2 10	9 1 10	10 0 10 G

Figure 15 : The g, h and f costs of a grid with straight movement and Manhattan distance

The number of nodes expanded until the goal is found will solely depend on the tie-breaking strategy. In worst case scenario, every node is first expanded before the goal is reached. This scenario can be true if ties are broken based on lowest $g - cost$ or FIFO queue implementation of $open\ list$.

3.3 Search Space reduction using directional pruning

The paper (Dainel Harabor, 2012) on the JPS pathfinding system defines the Jump Point Search. Jump Point Search works on top of A* search with two sets of rules: Pruning Rules and Jumping Rules. For this thesis, we are interested in the pruning rules that drive Jump Point Search. The pruning rules on this paper are updated from the pruning rules from (Daniel Damir Harabor, 2011) to not allow corner-cutting in grids. These pruning rules are online (does not require pre-processing) and optimality preserving (Daniel Damir Harabor,

2011). The basis of pruning rule is that when expanding a node, all children which can be reached by path shorter than the current path is pruned.

3.2.2.1 Natural Neighbors

For any node x, that has its parent p(x) and node $n \in neighbors(x)$, let us assume there are two paths π and π',

$$\pi' = < p(x), y, n > where\ y \neq x \tag{7}$$

$$\pi = < p(x), x, n > \tag{8}$$

Then the pruning rules are defined as,

1. For straight moves prune all neighbor nodes where,

$$len(\pi') \leq len(\pi) \tag{9}$$

Intuitively, if a node x has been chosen for expansion then it is a node along the shortest path. Any neighbor of node x, that has a shorter and can be reached without traversing through node x, can be pruned (node above x). This is because, if for some reason, the shorter path was not expanded or was expanded but is not along the optimal path, it is pointless to expand it from x as the path to that node from x will not be the shortest path to that node, i.e. $g(n)$ from node x will not be the smallest $g(n)$. For equal paths, like the one diagonal to node x it needs to be pruned to avoid path symmetry. If we remember the definition of path symmetry, two paths are symmetric if they have the same cost, and the movements are a permutation of one another. This means that the neighbor n, that is diagonal from node x can be reached through another path.

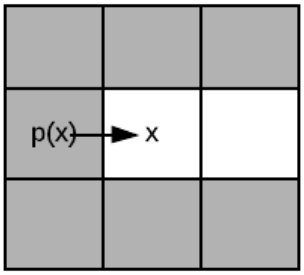


Figure 16 : Natural Neighbor for a straight move

The figure above shows node x being expanded from parent $p(x)$. The direction of movement is towards the right from parent $p(x)$. All nodes in grey are pruned out based on equation 9 defined above. The only remaining unpruned neighbor is the neighbor to the right of x.

2. For diagonal moves, prune all neighbor nodes where,

$$len(\pi') < len(\pi) \tag{10}$$

This follows the same intuition as pruning straight moves. Node x has been expanded and the children n which can have the shortest path without going through x can be pruned out because going through node x would not result in the smallest $g(n)$ anyway. However, because straight movements omitted neighbors that could be reached through paths of equal length, we need to include them for diagonal moves. We believe that these length based pruning rules can be interchanged between straight move and diagonal moves i.e. have straight moves prune $len(\pi') < len(\pi)$ and diagonal moves prune $len(\pi') \leq len(\pi)$ instead. Either way, only one path to the node is expanded and the symmetric one is omitted.

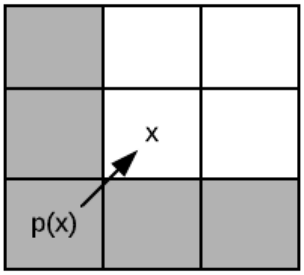


Figure 17: Natural Neighbor for a diagonal move

The figure above shows node x being expanded from parent $p(x)$ when the direction of travel is diagonal. There are three remaining unpruned neighbors of x. If one of the children of x is expanded next, it will also follow the pruning rules defined above. The children to the top and right of x will follow the straight pruning rules. The child diagonal to x will follow the diagonal pruning rule. The unpruned neighbors of x are called the $natural\ neighbors$ of x (Daniel Damir Harabor, 2011).

3.2.2.2 Forced Neighbors

There are changes to the pruning rule if a node encounters an obstacle. If there exists an obstacle adjacent to the parent node which is orthogonal to the direction of expansion, then none of the neighbors in the direction of the obstacle can be pruned. These neighbors generated because of the obstacle are called the forced neighbors.

Forced neighbors adhere to the pruning rules given above as there exists no shorter path to those nodes without going through x.

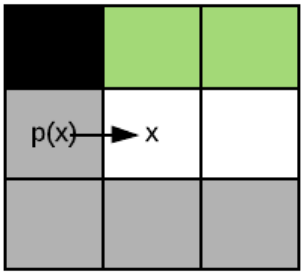


Figure 18: Forced neighbors for a straight move

The nodes in green are the $forced\ neighbors$ of x because of obstacle present adjacent to $p(x)$ and orthogonal in direction of movement (Dainel Harabor, 2012). For the green nodes shown above, the nodes must go through node x to be the shortest path to the node because of the obstacle present above $p(x)$.

The diagonal movements cannot have forced neighbors because having obstacles to the right of or above $p(x)$ would mean that the expansion of x has cut a corner which is an illegal move.

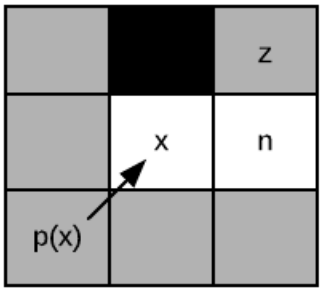


Figure 19 : Blocked neighbor in a diagonal move

We must pay special attention to corner-cutting in grids while applying these pruning rules. While expanding the node x, if there exists an obstacle on one of the straight moves like shown above, the diagonal node z is now pruned because of corner-cutting and the only

remaining natural neighbor is the node to the right of x. When the node n shown above is expanded from parent node x, z becomes part of n's forced neighbor.

3.3.1 Effect in Lookaheads

While this search space reduction technique works for $A*$ search, its effect on DFS lookaheads is significantly better. Because the DFS search implemented here is of tree-search nature, lookaheads done from any child node are exponential. Essentially, every node from the frontier generates b^m children. This makes the time complexity $O(b^m)$ where m is the maximum depth and b is the branching factor. This effect is compounded in grids if lookahead is performed in cost bounded DFS. The problem with cost bounded DFS is that there can be multiple nodes that are a fraction of their $f - values$ away from each other. As we saw in the example above for straight movement and Manhattan distance, all nodes had the same $f - value$. A similar thing is true for when we use octile movement with Euclidean distance. While most nodes won't have the same $f - value$, a difference of $f - cost$ of 1 may span anywhere from 10 – 100s of nodes.

The branching factor for DFS lookaheads in $b = 8$. As time complexity of the algorithm is $O(b^l \times b_e^{d-l})$, where b_e is the branching factor in the A* portion of the algorithm, b is the branching factor in DFS lookahead and l is the length of the DFS search. A branching factor of $b = 8$ would mean there would be a significant overhead for l. With search space reduction, the branching factor is less than $b = 3$ (2 for empty grid). There is a significant difference in expanding nodes with branching factor of the exponential of 8 and branching factor of the exponential of 2. DFS lookahead also needs to expand these nodes again while the A* version of the algorithm only expands nodes along the optimal path.

Not keeping track of visited nodes in the $closed\ list$ and expanded nodes in the $open\ list$, the path to the same node along an optimal path $g(n)$ can be generated with different costs. Because all redundant paths and symmetric paths are removed, the DFS lookahead does not expand the same node more than once per lookahead because the pruning technique is optimality preserving (Daniel Damir Harabor, 2011).

Similarly, implementing the pruning technique in the $A*$ part of the algorithm also reduces the number of nodes generated per expansion of the node thereby decreasing the number of times we would need to perform lookaheads. The effective branching factor b_e that is used in both space and time complexity is also affected by this pruning technique. Note that all nodes that need to be expanded for $A*$ will be expanded regardless of if the pruning technique is used or not.

For children with the same $f-value$ as its parent, we do not do a lookahead search. Instead, we assign the same $f_{updated}$ value to the child as we did to its parent. This was dubbed trivial lookahead by (Roni Stern, 2010), however, we add the node to $open\ list$ instead of moving it directly to $closed\ list$.

Let us assume, there is a node n, for which the $f-value$ is the same as its parent. If the lookahead search of $k=1$, does not find a path to the goal with $f(parent)+k$ then it is very likely that the path in the direction towards child node n will result in a dead-end. This causes A* to expand nodes that are not along an optimal path. When adding these nodes directly to the closed list, (Zhaoxing Bu, 2014) found that using trivial expansion increases the number of nodes expanded and generated in maps with unit cost.

CHAPTER 4:

Experimental Setup

4.1 Implementation Details

We implement the algorithms using C# as the programming language. All experiments are done on a computer with Intel Core i7-7700 CPU and 16 GB RAM. The visualizations are built as a bitmap where each tile is a pixel in the bitmap. We use C#'s HashSet for implementation of the $closed\ list$ and Priority Queue for the $open\ list$. The experiments are Single Agent pathfinding problems in a static map. The maps used are benchmark maps from various video games at Moving AI Labs website (https://movingai.com/benchmarks/formats.html, n.d.).

We use two algorithms $A*$ search and $A*$ with Lookahead. Both algorithms have been modified to not allow corner cutting through obstacles. We compare two variants of A* with Lookahead search, one with the domain-specific search space reduction techniques/pruning technique and the other without it. From the standard benchmark maps, we use scenarios that are available to get the start and end points.

We make different comparisons for A* search versus A* with Lookahead and A* with Lookahead versus A* with Lookahead using pruning. The focus of our research is showing that the lookahead based search has performance gains on memory consumed and with the pruning techniques, we can overcome limitations of the lookahead search and achieve execution time better than or similar to A* search.

We run experiments on different map sizes with varying obstacle chance for A* vs A*L. We vary the size of k in A*L from 0 to 5 and do cost-bounded lookaheads for our experiments. For unpruned A*L, we only run the experiments in a small map as the time taken would be significantly larger than the pruned version.

4.2 Experimental Setup

We've discussed grid-based maps before because it is pertinent to our thesis and it ties in with the search space reduction. For our experiments, we use movement costs of 1000 for straight moves and 1414 for diagonal moves in our grids. The final cost of the path is divided by 1000. Obstacles are evaluated at a cost of integer max value which is 2,147,483,647. They are not evaluated with a cost for $A*$ part of the search but return a cost of integer max value for the lookahead portion. This value is also used in place of ∞ to initialize the minimum cost variable ($MinCost$) and the Upper Bound variable UB too.

The standard maps are .map files that are readable like text files. The first four lines are map descriptions like map name, row and column. Remaining part of the file consists of the map details. (https://movingai.com/benchmarks/formats.html, n.d.). Each character in the file is a tile in the map environment. A space, '.', 'G' and 'S' are characters that represent traversable terrain. All other characters are considered untraversable. We add another specific character 'Z' as obstacles that were randomly generated in our experiments. The process of generating obstacles are defined in the section below. After our experiments, we save a new map with character Z for our randomly generated obstacles as a .map file. This way we can reproduce our experiments if needed.

For start and end points, we randomly select points from the scenario files among the $1/3^{rd}$ largest scenarios.

4.2.1 Search Parameters

We use specific maps of different sizes approximately 128x128, 211x215, 320x320, 385x385 and 512x512. We use actual video game maps from Baldur's Gate's unscaled maps, Dragon Age Origins, and Starcraft.

Most maps already have obstacles present in them. We randomly add extra obstacles amongst the available traversable terrain based on the obstacle chance we want. The percentages of obstacles we add are 0, 7, 15 and 30. This way we can see the effect of obstacles on the algorithms. For every traversable node, we use the random function to determine if the node is going to be an obstacle or not.

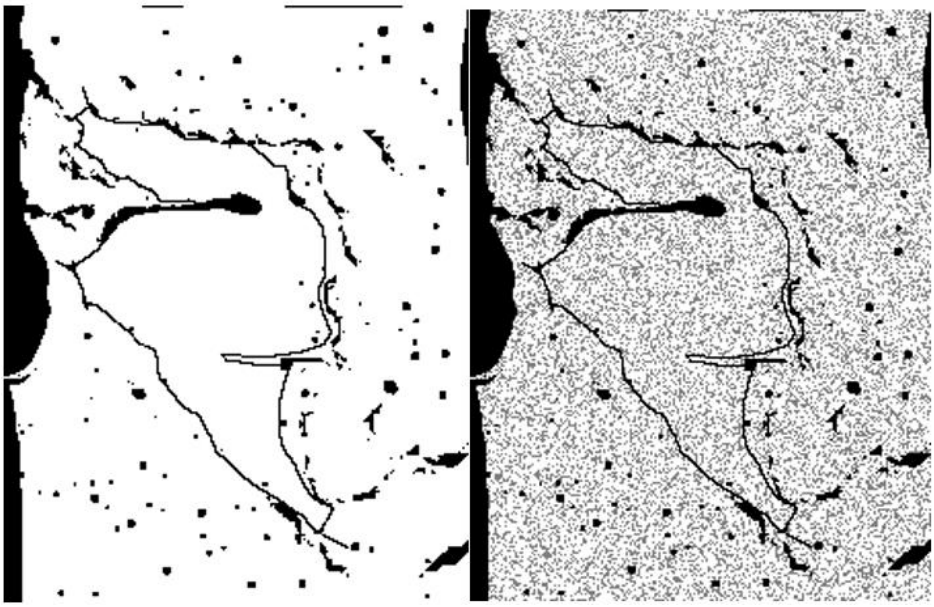


Figure 20: Map with 0% added obstacles on left, map with 30% obstacles on right

The figures are maps with 0% obstacles added and 30% obstacles added. Each pixel in the bitmap is a tile. The pixels in white are traversable tiles and the pixels in black are original obstacles in the game map. The ones in gray are obstacles that we have randomly added to the map. We only vary the obstacles to up to 30% of traversable terrain because increasing the obstacle percentage to around 40% generated maps that failed to find a path for majority of the $start$ and $goal$ nodes.

We vary the lookahead cost k between 0 to 5 from the cost of the parent. The number of frontiers generated by the lookahead process has significant overhead in terms of the algorithm's runtime. We go all the way up to a k value of 5 to see the tradeoff of memory consumed to time taken against standard the $A * search$ algorithm.

4.2.2 Performance Evaluation

We gather results from our experiments at runtime and compare A* with Lookahead for different values of k with standard A* search. We take the Time Taken, the number of nodes generated, the number of nodes expanded, path length and number of nodes expanded during lookahead as performance evaluation metrics.

4.2.2.1 Time Taken

For each algorithm, we use a stopwatch to calculate the time taken. We use C#'s stopwatch class `System.Diagnostics.Stopwatch` and measure the time elapsed. We first initialize all variables and classes. We call the start method before we start our search and call the stop method after the search finishes. The time taken is reported using the `ElapsedMilliseconds()` method.

The time taken in milliseconds is not an implementation-independent performance evaluation metric. This metric will vary depending on different implementation methods and environments. We use this metric to compare the performance of A* with lookahead against the standard A* search. So, for this thesis, time taken should be a fair performance evaluation metric.

4.2.2.2 Number of Nodes generated

The number of nodes generated is an implementation and platform independent performance evaluation metric. It corresponds to the space complexity of our algorithms. We expect the number of nodes generated to decrease when we use A* with Lookahead. Nodes generated are the total number of nodes stored in the memory. We expect the nodes generated to decrease when the pruning process is used.

4.2.2.3 Number of Nodes expanded

The number of nodes expanded is an implementation and platform independent performance evaluation metric. We expect the number of nodes expanded to decrease when we use A* with Lookahead. Nodes expanded are nodes removed from the $open\ list$ and added to the $closed\ list$.

4.2.2.4 Path Length

The path length is the lowest cost path from start node to the goal node. The final path is optimal for A* search and A* with Lookahead Search. Regardless of the algorithm, the cost of optimal path should remain same. The path length does vary for the same start and goal node with the number of obstacles present in the map. The pathlength has impact on

the number of nodes expanded and the time taken. Increase in path length means more lookaheads done on the map. The average path length is the average for experiments done on a map with certain search parameter. We track the path length across different values of k to show that regardless of the value of k our algorithm is optimal.

4.2.2.5 Average Nodes Expanded during Lookahead

The average number of nodes expanded during Lookahead is the number of nodes expanded during the DFS portion of the algorithm. While lookahead helps save memory, they are also redundant in nature. While expansion of a small number of nodes per lookahead can lead to benefit in performance, large number of nodes expanded per lookahead results in an increase in time taken for the path to be found. Because lookaheads are done only from generated nodes, the average node expanded is calculated by,

$$Average\ Lookahead\ nodes\ expanded = \frac{Total\ lookahead\ nodes\ expanded}{Nodes\ generated}$$

4.2.3 Results and analysis

We use our test framework to evaluate performance of A* with Lookahead and compare it with standard A* search. As discussed above, we use locations in map from scenarios already available. We run 50 experiments with obstacle chance of 30% as finding a path for that is the hardest. We generate 50 start and end points for the map that will be used with other obstacle chances. It is better to use start point and end points from a map with 30% obstacle chance than to generate a map with 30% obstacle chance that must find a path for existing start and end points.

The maps are selected based on the open space available. The experiments ran fine with tight spaces if there weren't corridors which had a very small width. The start and end points remain same for different obstacle chances per map. This gives us an opportunity to see the change in performance and path length with relation to the number of obstacles in the map.

Added Obstacle chance	Number of Experiments	Map Size	Algorithms
0% 7% 15% 30%	50	128 x 128 211 x 251 320 x 320 384 x 384 512 x 512	A* Search A*L with k=0 A*L with k=1 A*L with k=2 A*L with k=3 A*L with k=4 A*L with k=5

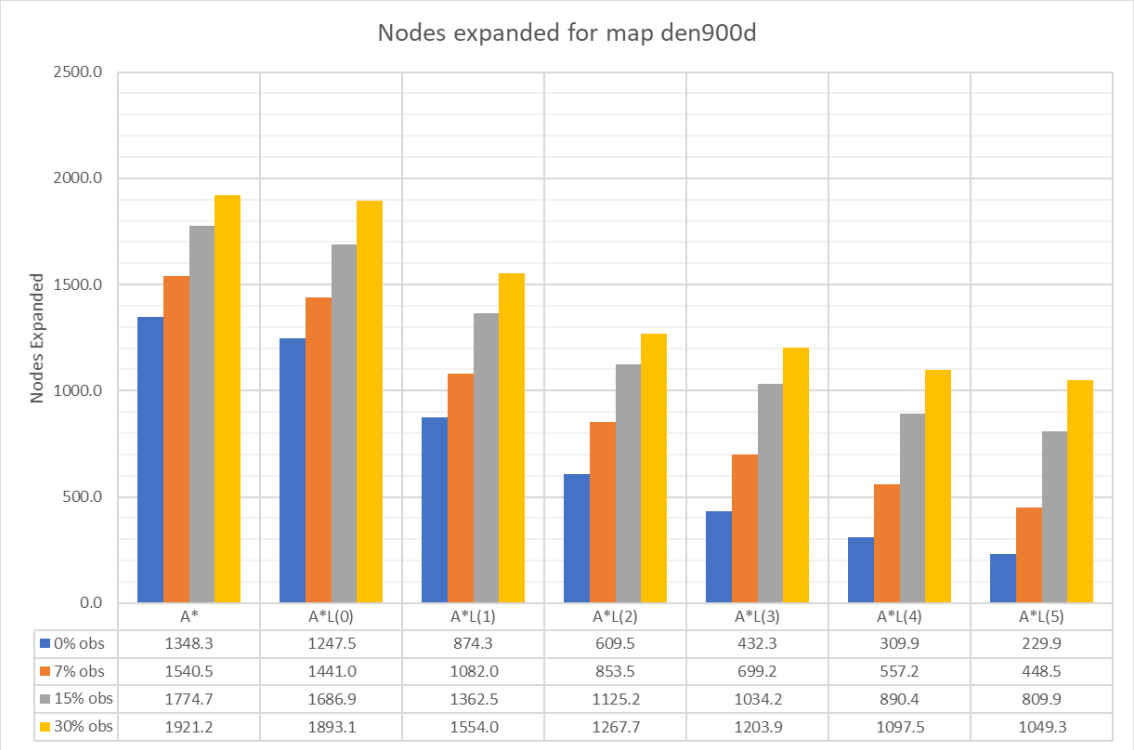
Table 1: Table of experiments

We present our findings based on the size of the map. This lets us correlate performance to map size and path length. Separate charts for varying obstacle chance with time taken, nodes expanded, nodes generated and average number of nodes expanded during lookahead will be shown below. Tables for all charts are available in the indices with average path lengths, nodes expanded, nodes generated, and time taken.

4.2.3.1 Results for Map size 128x128

We use den900d map from Dragon Age Origins as the benchmark map for this experiment. All results are values averaged out across 50 different experiments, 10 random maps with 5 different start and goal nodes. The start and goal nodes are same across maps with different obstacle chances. There is an increase in the path length when the obstacle chance

increases. The map has 5,258 traversable states. The cost of optimal path is same across A* and A*L with different values of k.

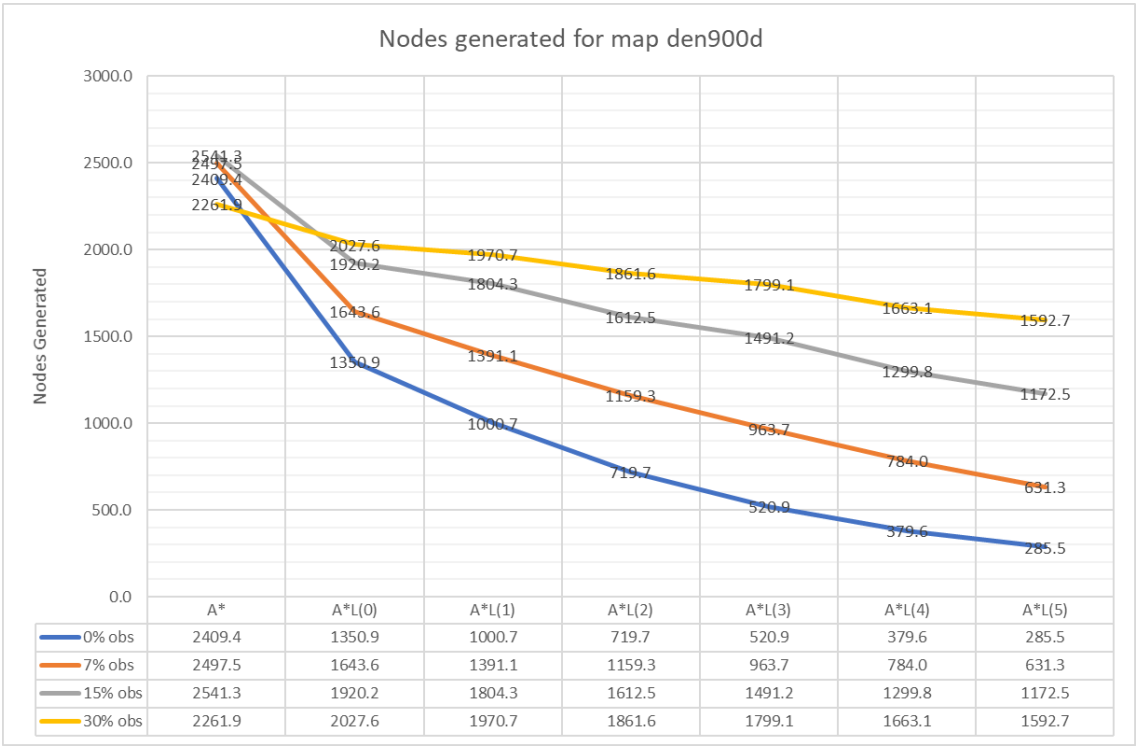


	A*	A*L(0)	A*L(1)	A*L(2)	A*L(3)	A*L(4)	A*L(5)
0% obs	1348.3	1247.5	874.3	609.5	432.3	309.9	229.9
7% obs	1540.5	1441.0	1082.0	853.5	699.2	557.2	448.5
15% obs	1774.7	1686.9	1362.5	1125.2	1034.2	890.4	809.9
30% obs	1921.2	1893.1	1554.0	1267.7	1203.9	1097.5	1049.3

Figure 21: Nodes Expanded 128x128 map

The number of nodes expanded increases as the percentage of obstacles increase because the average path length increases, and the heuristic performs worse when there are more obstacles. We see that for all obstacle chances, the number of nodes expanded by A*L decreases as the lookahead value k increases. There is an 83% decrease in the number of nodes expanded on standard maps when the lookahead value k is equal to 5. There is a 70% decrease in the nodes expanded between A* and A*L with k=5. Similarly, 54.4% decrease on 15% obstacles and a 45% decrease when the number of obstacles on the map is 30% of the traversable terrain.

At the value of k = 0, A*L is similar to A* search with pruning technique applied. There is some decrease in the percentage of nodes expanded, however that can be attributed to the lookahead search as there can be a lot of nodes with f-value 0 along the path to the goal node.



	A*	A*L(0)	A*L(1)	A*L(2)	A*L(3)	A*L(4)	A*L(5)
0% obs	2409.4	1350.9	1000.7	719.7	520.9	379.6	285.5
7% obs	2497.5	1643.6	1391.1	1159.3	963.7	784.0	631.3
15% obs	2541.3	1920.2	1804.3	1612.5	1491.2	1299.8	1172.5
30% obs	2261.9	2027.6	1970.7	1861.6	1799.1	1663.1	1592.7

Figure 22: Nodes Generated 128x128 map

Nodes generated are the total number of nodes that are stored in the memory. As the nodes generated is directly corelated to the space complexity in A*, it makes sense to see that for A* search lesser nodes are generated when obstacles are present. This is because of presence of obstacles decreases the branching factor and the number of available traversable states.

As the pruning technique is used, the effective branching factor decreases, and a smaller number of nodes are generated as shown in the figure above. The percentage of nodes saved

is highest for 0% obstacles added to the map. Because lookaheads are done for nodes generated, lower number of nodes generated correspond to lesser lookahead in A*L and therefore lesser time consumed. The percentage of nodes saved by A*L decreases when the obstacle chance increases.

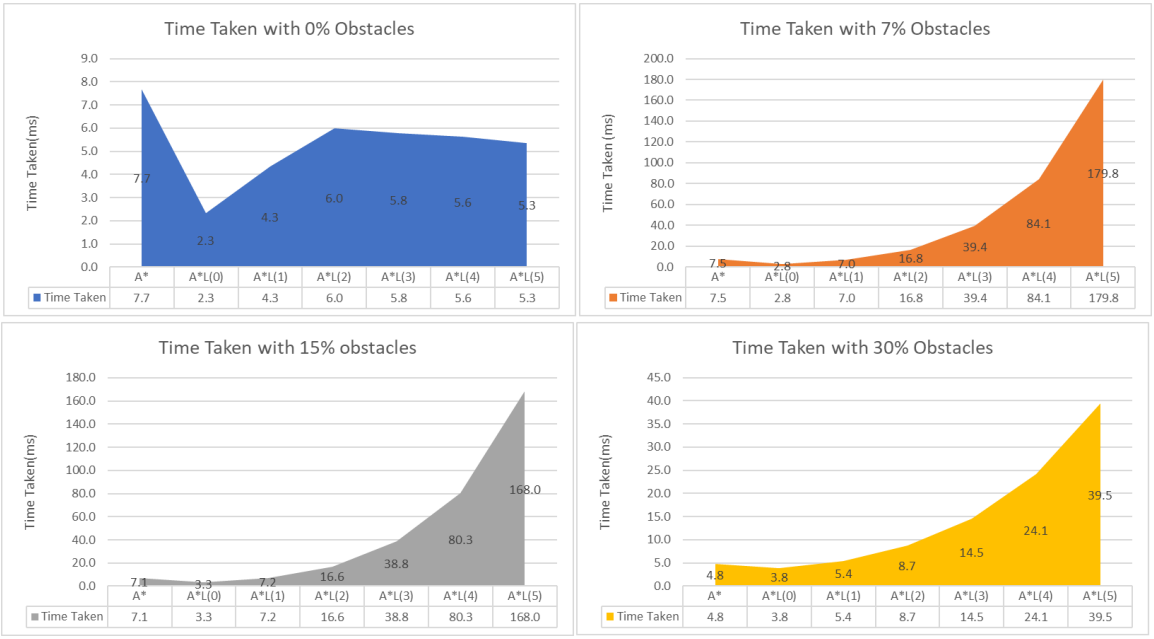


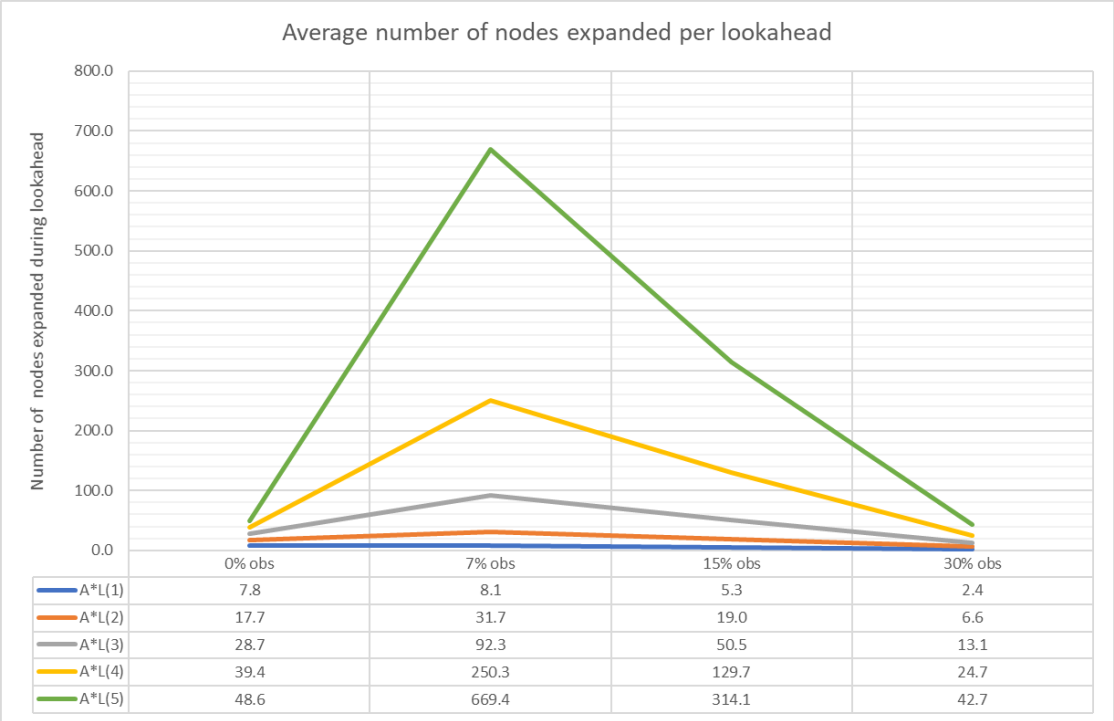
Figure 23: Time taken for 128 x 128 map

For time taken, A*L performs best when there are no extra obstacles added to the map. The version of A*L shown above uses the pruning technique we described earlier. This shows that the pruning technique successfully decreases the time taken and the overhead of doing DFS lookahead. A*L with pruning technique is comparable to A* search or better for the 128x128 map.

We start seeing an exponential increase in the time taken as k increases when we add obstacles. In general, as the value of k increases the time taken increases too. However,

there is a 2500% increase in time taken when 7 % obstacles are present and 2300% increase when there are 15% additional obstacles. The reason for this increase is due to obstacles being uniformly and sparsely distributed. This is the worst-case scenario for the pruning technique as forced neighbors of the nodes are generated frequently. This causes nodes that would normally have a single neighbor to have three neighbors. We also see that when increasing the obstacle chance to 30% the time taken at k= 5 drops down to 822%.

At each obstacle chance there is a trade-off at certain value k where we achieve decrease in node generated with an acceptable increase in time taken versus A* search.



	0% obs	7% obs	15% obs	30% obs
A*L(1)	7.8	8.1	5.3	2.4
A*L(2)	17.7	31.7	19.0	6.6
A*L(3)	28.7	92.3	50.5	13.1
A*L(4)	39.4	250.3	129.7	24.7
A*L(5)	48.6	669.4	314.1	42.7

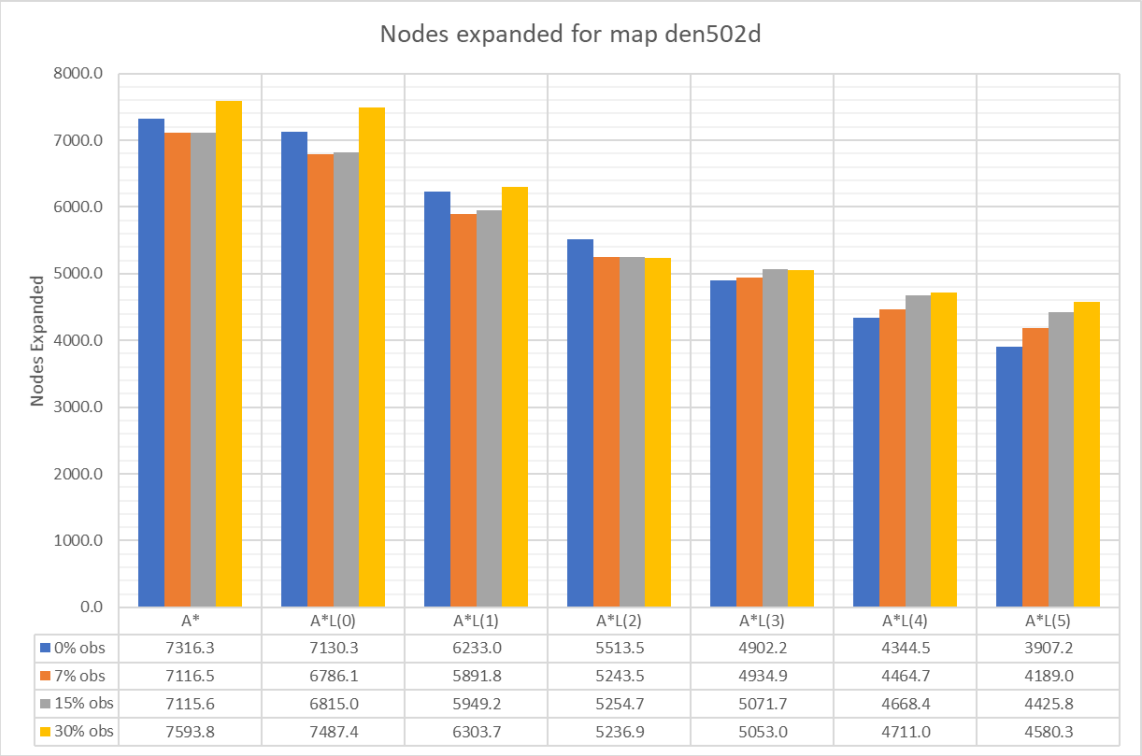
Figure 24: Avg. nodes expanded per lookahead 128x128 map

The average number of nodes expanded per lookahead shows the correlation of time to the lookahead step. As we mentioned before, adding obstacles increases the branching factor in the DFS lookaheads. Later, we show how having sparsely distributed obstacles also re-

introduces cycles in the DFS. We also see that as the value of k increases the number of lookaheads done increases $O(b^l)$, where l is the depth of lookahead with cost k. However, for 0% added obstacles, the time taken decreases at k=5, whereas the average number of nodes expanded per lookahead increases because there are far fewer nodes generated (the nodes generated graph). Therefore, the time taken is dependent of a combination of nodes generated and lookahead nodes expanded.

4.2.3.2 Results for Map size 211x251

We use den502d map from Dragon Age Origins as the benchmark map for this experiment. Like the previous experiment, all values are averaged out across 50 experiments. The number of traversable states in the map is 27,235. The cost paths average out the same across all versions of A*L and A* search so we can conclude that the paths generated by respective algorithms are optimal. The average cost path increases as the percentage of obstacles in the map increases from 222 to 272.



	A*	A*L(0)	A*L(1)	A*L(2)	A*L(3)	A*L(4)	A*L(5)
0% obs	7316.3	7130.3	6233.0	5513.5	4902.2	4344.5	3907.2
7% obs	7116.5	6786.1	5891.8	5243.5	4934.9	4464.7	4189.0
15% obs	7115.6	6815.0	5949.2	5254.7	5071.7	4668.4	4425.8
30% obs	7593.8	7487.4	6303.7	5236.9	5053.0	4711.0	4580.3

Figure 25: Nodes Expanded 211x251 map

For nodes expanded on 0% obstacles, with k=5, $A * L(5)$ has 46% less nodes expanded. At 7%, 15% and 30% obstacles, 41%, 37% and 39% less nodes expanded compared to A*. Once again as the value of k increases in A*L the number of nodes expanded decreases.

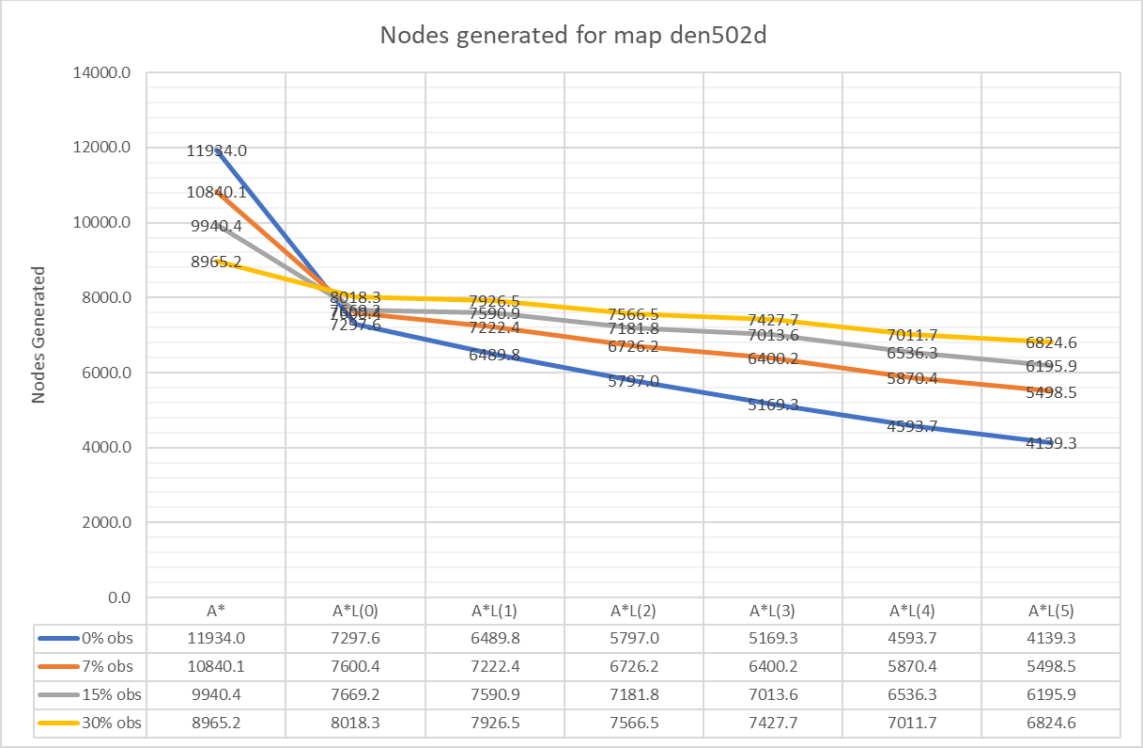


Figure 26: Nodes Generated 211x251 map

There is a similar trend for the nodes generated. Using the pruning technique reduces the number of nodes generated. As the lookahead cost k increases, the total number of nodes generated decreases. This is true regardless of the percentage of obstacles. However, the percentage of space saved decreases as the percentage of obstacles increases.

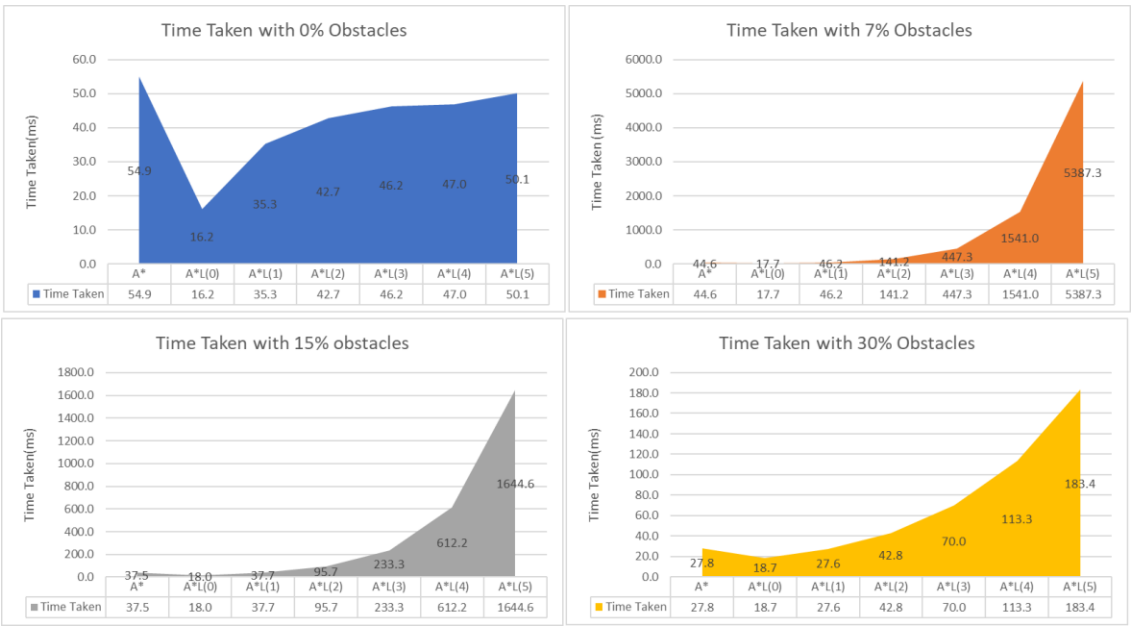


Figure 27: Time taken for 211 x 251 map

Once again, at 0% obstacle chance, the time taken for all values of k with modified A*L is better than A* search. There is an exponential increase in the time taken for 7% chance and 15% chance and less exponential increase for 30% obstacle chance.

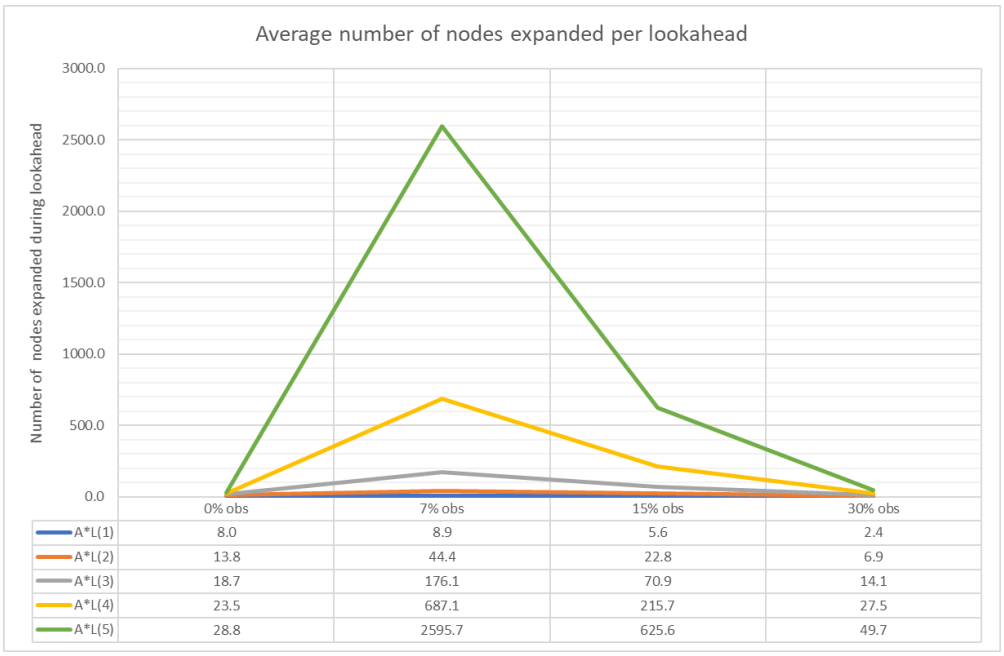
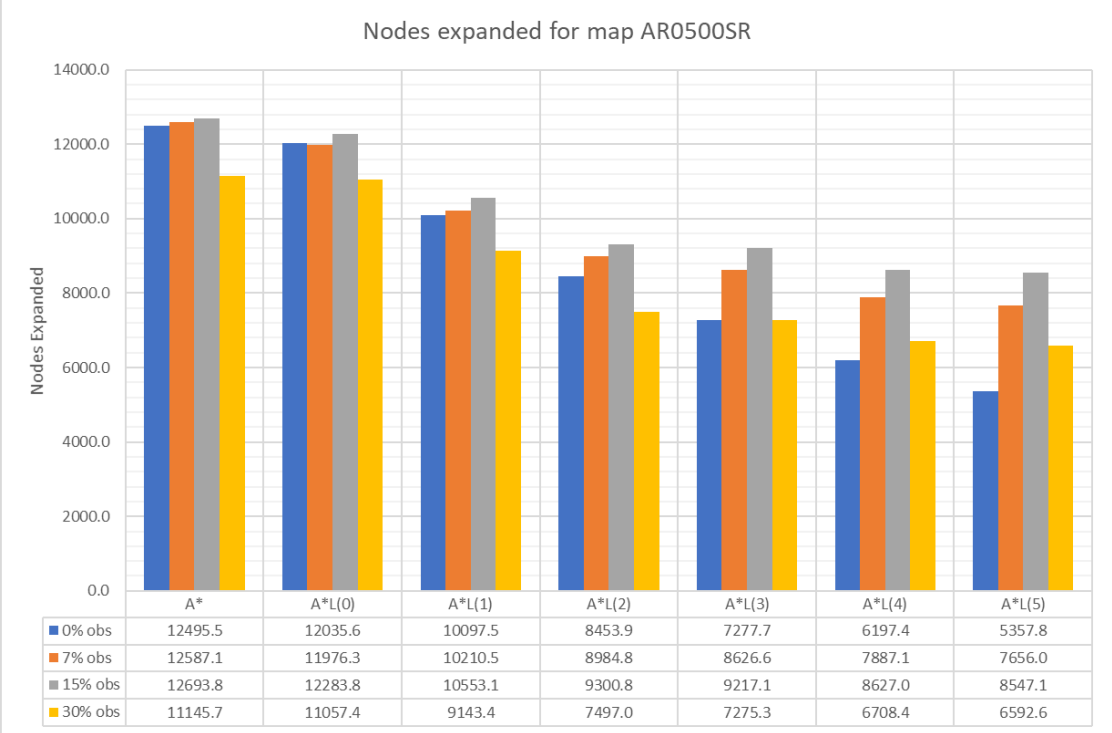


Figure 28:Avg. nodes expanded per lookahead 211x251 map

The average number of nodes expanded per lookahead increases as k increases. The average number of nodes expanded per lookahead peaks at 7% and at k=5, which would explain the increase in time taken when 7% obstacles are randomly added. The average nodes expanded per lookahead decreases at 30% obstacles added because the obstacles are less sparse and there are lesser states remaining in the map.

4.2.3.3 Results for map size 320x320

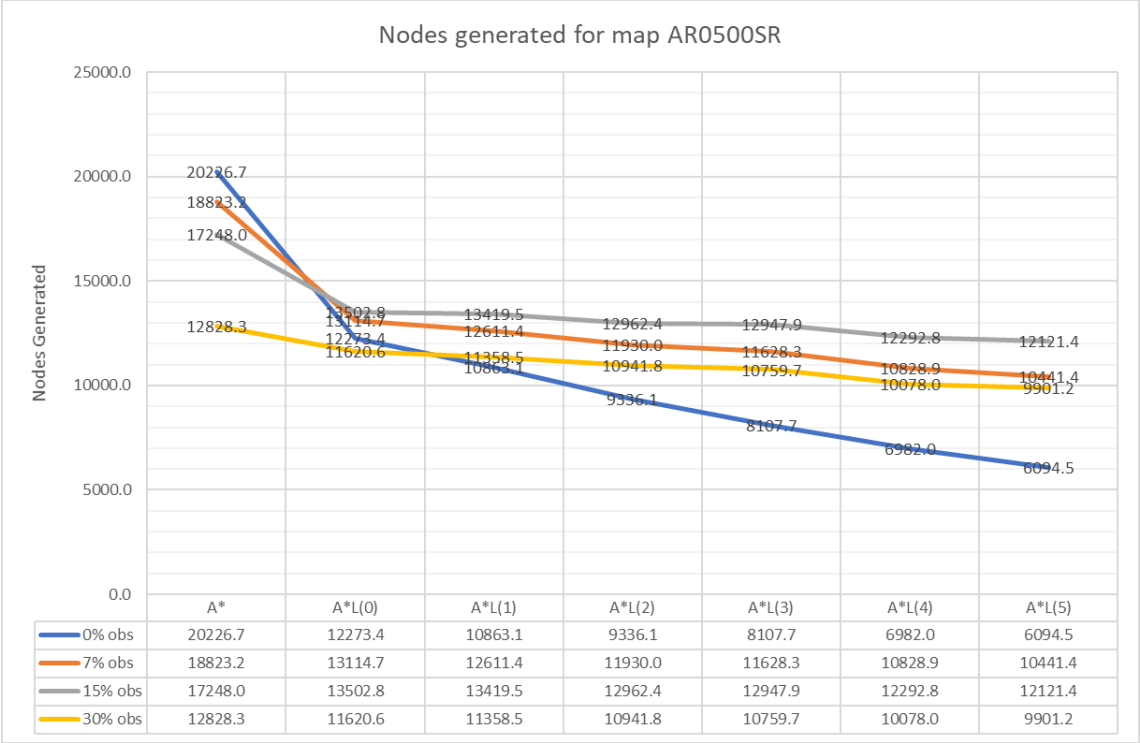
We use AR0500SR map from Baldur's Gate as the benchmark map for this experiment. Like the previous experiment, all values are averaged out across 50 experiments. There are 29,160 traversable states for this map. The average path cost increases from 382 to 505 as the percentage of obstacles added increases.



	A*	A*L(0)	A*L(1)	A*L(2)	A*L(3)	A*L(4)	A*L(5)
0% obs	12495.5	12035.6	10097.5	8453.9	7277.7	6197.4	5357.8
7% obs	12587.1	11976.3	10210.5	8984.8	8626.6	7887.1	7656.0
15% obs	12693.8	12283.8	10553.1	9300.8	9217.1	8627.0	8547.1
30% obs	11145.7	11057.4	9143.4	7497.0	7275.3	6708.4	6592.6

Figure 29: Nodes expanded for 320 x 320 map

Like previous experiments, the highest decrease in nodes expanded is on 0% obstacles added at 57% for k = 5, 39% decrease on 7% additional obstacles, 32% decrease on 15% obstacles and 40% decrease when additional 30% obstacles are added.



	A*	A*L(0)	A*L(1)	A*L(2)	A*L(3)	A*L(4)	A*L(5)
0% obs	20226.7	12273.4	10863.1	9336.1	8107.7	6982.0	6094.5
7% obs	18823.2	13114.7	12611.4	11930.0	11628.3	10828.9	10441.4
15% obs	17248.0	13502.8	13419.5	12962.4	12947.9	12292.8	12121.4
30% obs	12828.3	11620.6	11358.5	10941.8	10759.7	10078.0	9901.2

Figure 30:Nodes generated for 320 x 320 map

As with previous maps, there is a decrease in the number of nodes generated by A* search as the obstacles increase. Meanwhile, for A*L the number decreases as the value of k increases. The percentage save in compared to A* search is greatest when k = 5 and when there are 0% obstacles present in the map.

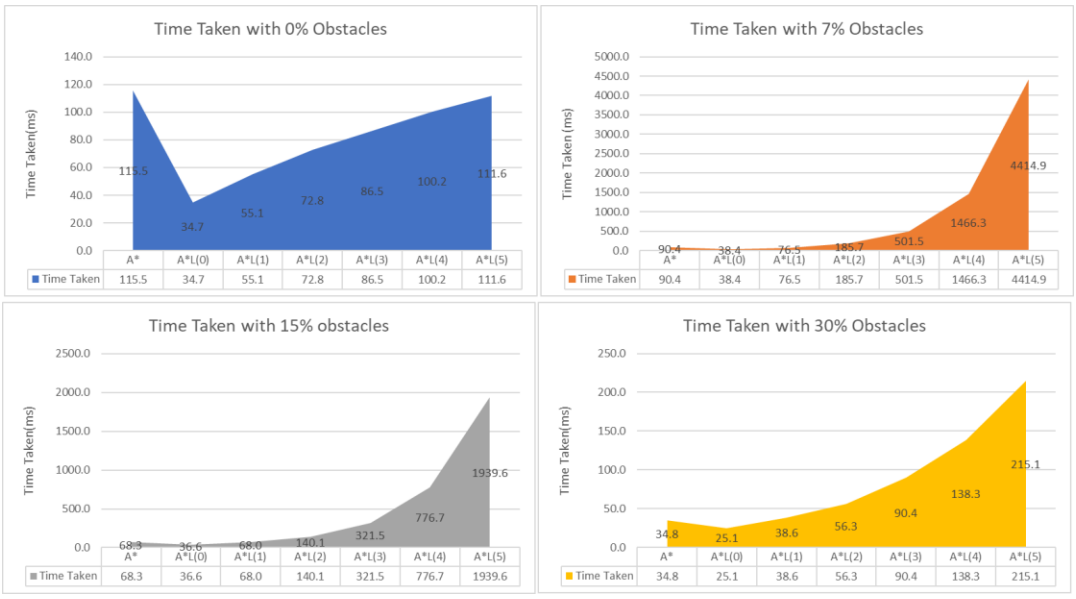


Figure 31: Time taken on 320 x 320 map

We see similar trends for time taken as previous experiments here. There is an increase in time for all cases of A*L when k increases. The increase in exponential for 7% and 15% added obstacles. At 0%, while there is an increase in time taken when *k* increases, it is still better than standard A* search.

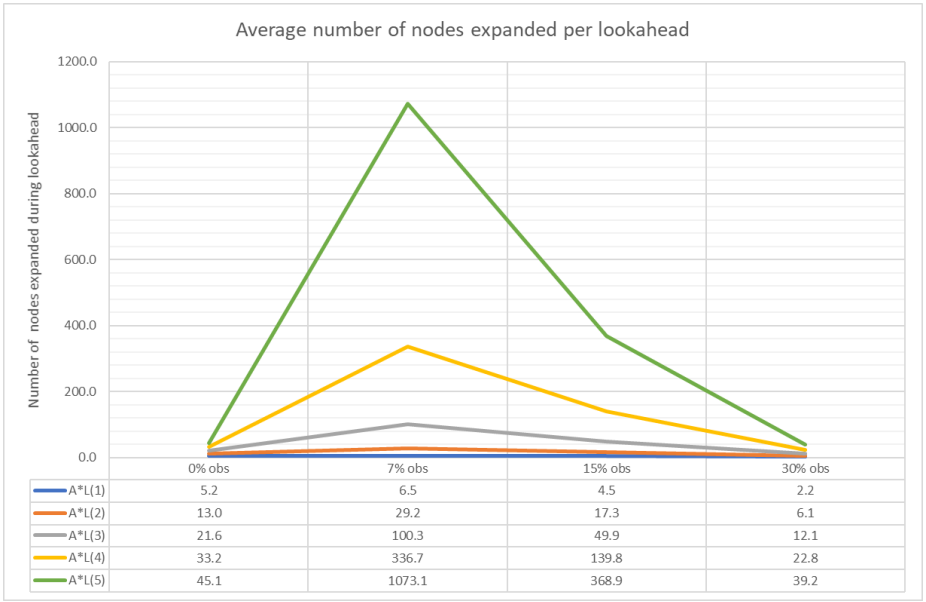


Figure 32:Avg. nodes expanded per lookahead for 320 x 320 map

Similarly, there's an increase in the average number of nodes expanded per lookahead as the value of k increases. Like with previous maps, when 7% obstacles are randomly added we see the performance of A*L worsens.

4.2.3.4 Results for map size 384x384

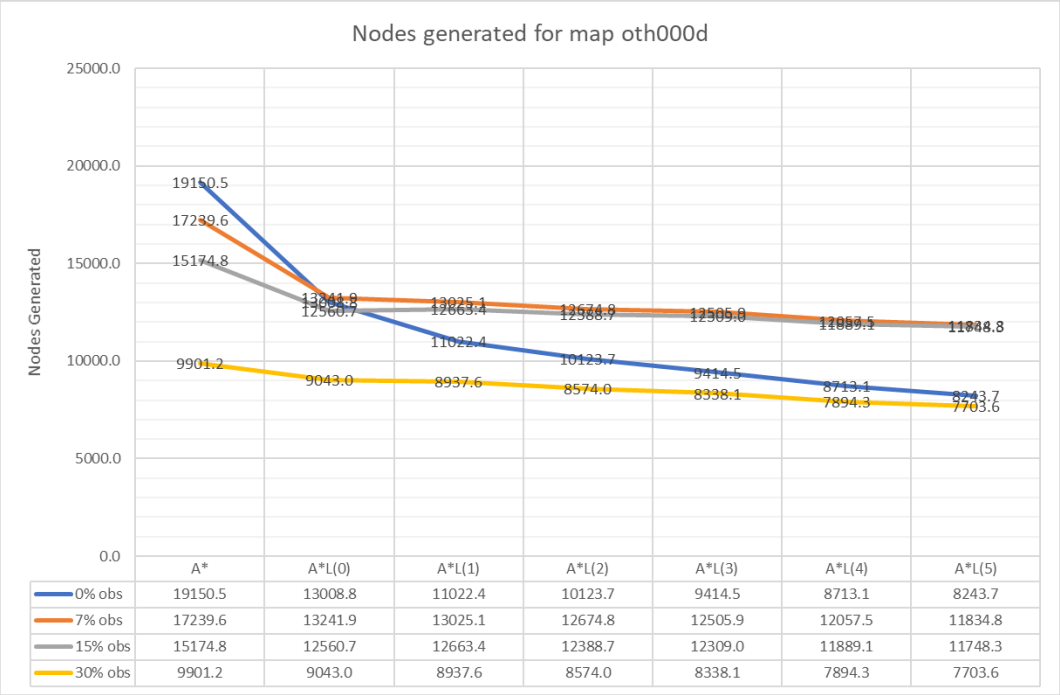
We use ooth000d map from Dragon Age Origins as the benchmark map for this experiment. The map has 17,601 traversable states. The map is different because the whole map has is a single gigantic path and has less free space than other maps. There are however lower number of natural obstacles along the path. The average cost path increases from 523 to 664 as the percentage of obstacles added increases.

	A*	A*L(0)	A*L(1)	A*L(2)	A*L(3)	A*L(4)	A*L(5)
0% obs	13015.3	12917.5	10534.5	9308.2	8619.1	7942.6	7493.5
7% obs	12333.4	12160.3	11055.7	10044.2	9756.1	9241.9	9022.7
15% obs	11641.7	11490.1	10409.5	9243.2	8997.2	8547.9	8415.0
30% obs	8676.2	8641.7	7281.4	5940.9	5636.6	5262.8	5120.3

Figure 33: Nodes expanded for 384 x 384 map

A similar trend follows for decrease in nodes expanded as value of k increases in A*L. For 0% added obstacles, there is a decrease in the number of nodes expanded with 42%

decrease in the number of nodes expanded at k = 5. There is clear indication of decrease in the percentage of nodes expansions saved as the percentage of obstacles increase



	A*	A*L(0)	A*L(1)	A*L(2)	A*L(3)	A*L(4)	A*L(5)
0% obs	19150.5	13008.8	11022.4	10123.7	9414.5	8713.1	8243.7
7% obs	17239.6	13241.9	13025.1	12674.8	12505.9	12057.5	11834.8
15% obs	15174.8	12560.7	12663.4	12388.7	12309.0	11889.1	11748.3
30% obs	9901.2	9043.0	8937.6	8574.0	8338.1	7894.3	7703.6

Figure 34:Nodes generated for 384 x 384 map

Likewise, for nodes generated, A* search generates the least number of nodes with 30% added obstacles. Just like previous experiments, for A*L, the most percentage of space saved is at 0% obstacles added and the percentage of nodes saved across $k = 0\ to\ 5$ decreases as the percentage of obstacles added increases.

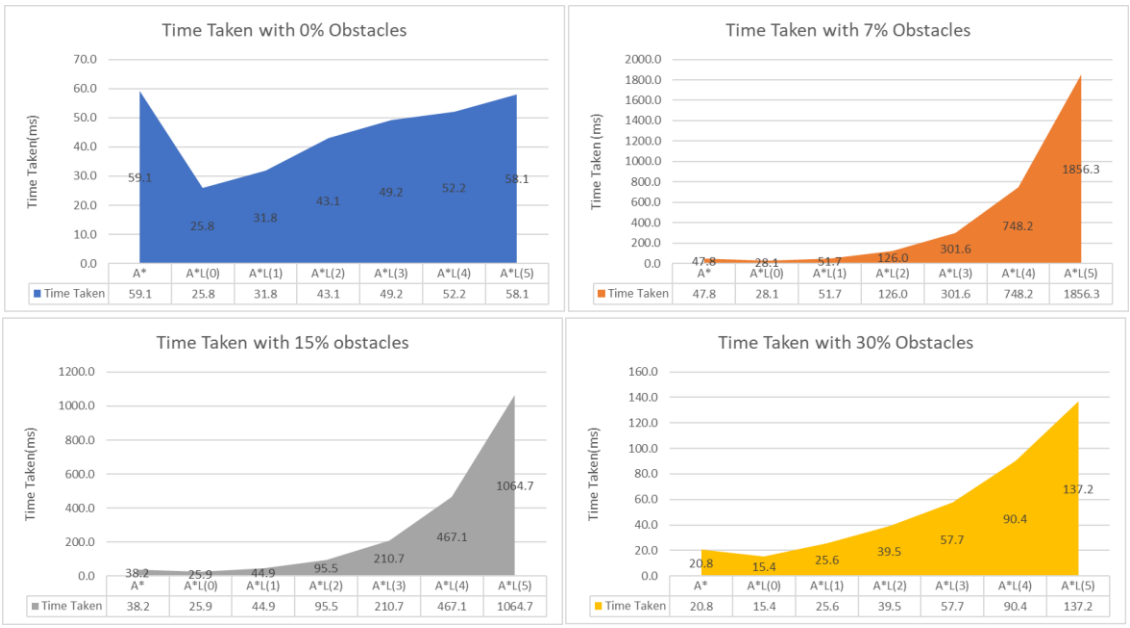


Figure 35: Time taken on 384 x384 map

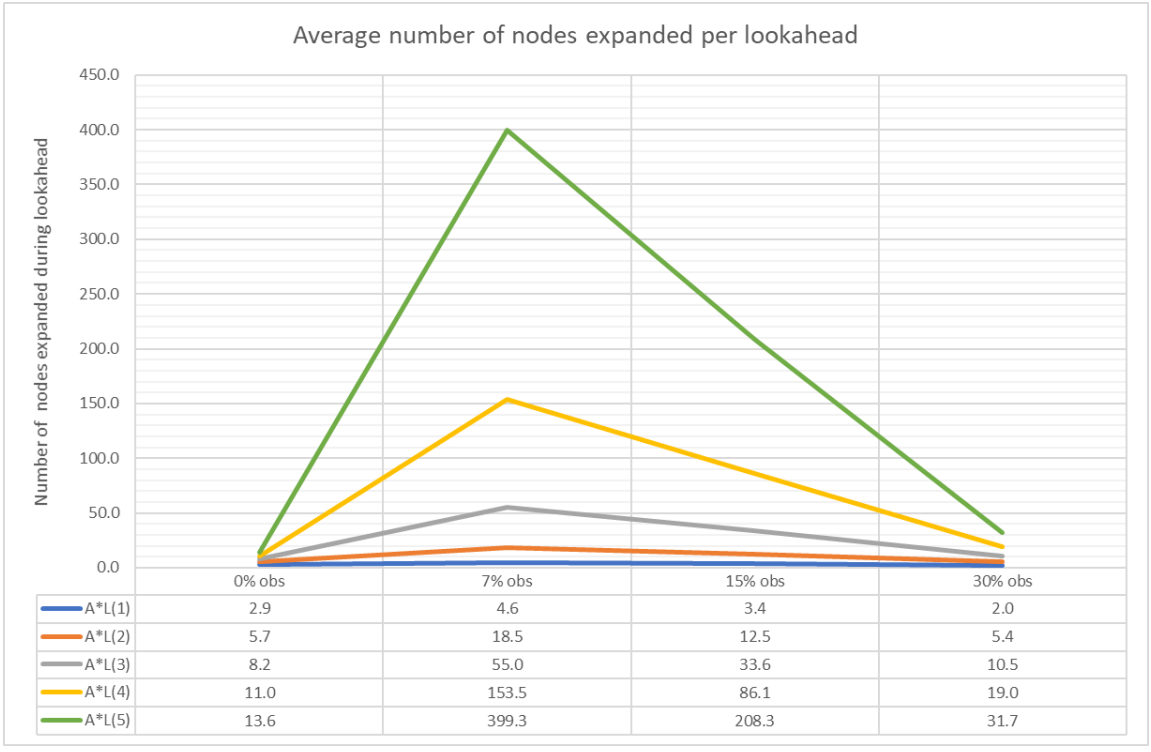
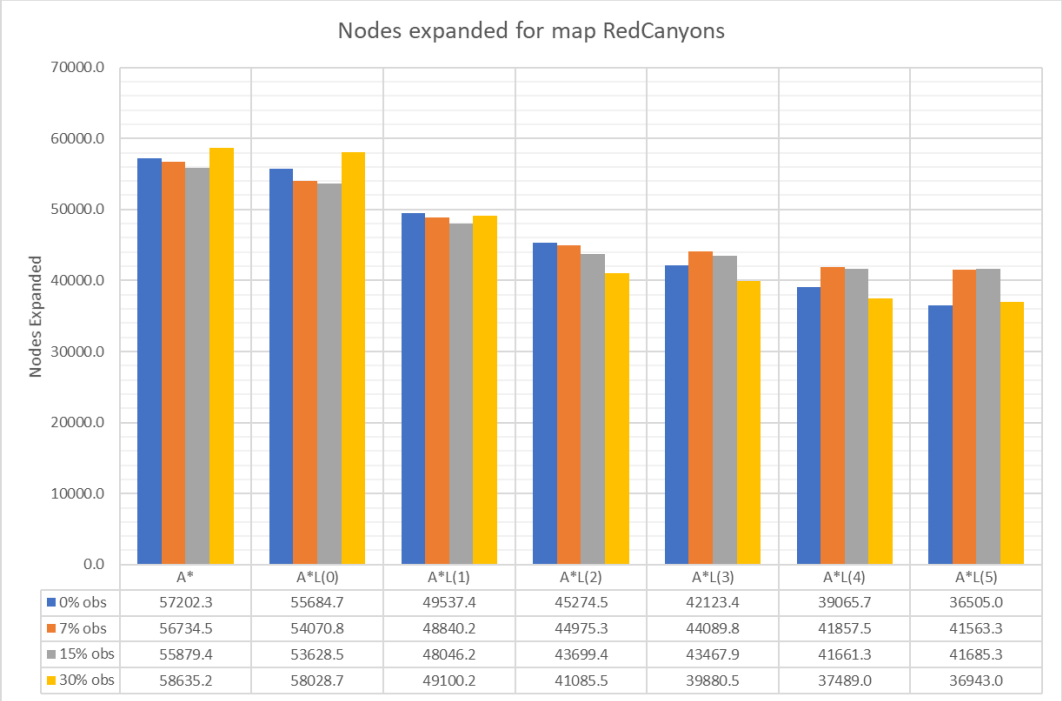


Figure 36:Avg. nodes expanded per lookahead for 384 x 384 map

The time taken by A*L at k=5 is similarly worst at 7% additional obstacles. Which is explained by the increase in average number of nodes expanded per lookahead at 7% obstacles. For this map, at 0% obstacles the time taken is slightly more for A*L with k = 5 than A* search. We can explain this by looking at the number of traversable states compared to the size of the map. The oth000d only has 17,601 traversable states despite being a 384 x 384 map. In comparison, den502d has 27,235 traversable states while being a 211x251 size map. This is because oth000d has significantly more obstacles already present in the map.

4.2.3.5 Results for map size 512x512

We use RedCanyons map from Starcraft as the benchmark map for this experiment. All values are averaged out across 50 experiments. The map is the largest we performed experiments for and has 174,722 traversable states. The average cost of path increases as we increase the percentage of added obstacles from 628 to 793. The map already has some number of obstacles present in it; however, the obstacles are a present as cluster.



	A*	A*L(0)	A*L(1)	A*L(2)	A*L(3)	A*L(4)	A*L(5)
0% obs	57202.3	55684.7	49537.4	45274.5	42123.4	39065.7	36505.0
7% obs	56734.5	54070.8	48840.2	44975.3	44089.8	41857.5	41563.3
15% obs	55879.4	53628.5	48046.2	43699.4	43467.9	41661.3	41685.3
30% obs	58635.2	58028.7	49100.2	41085.5	39880.5	37489.0	36943.0

Figure 37: Nodes expanded for 512 x 512 map

$A * L(5)$ decreases the number of nodes expanded at 0% obstacle chance by 36%. The largest decrease in the number of nodes expanded when using A*L can be seen at k=5 when compared to standard A* search. And as expected, the number of nodes expanded decreases as the value of k increases.

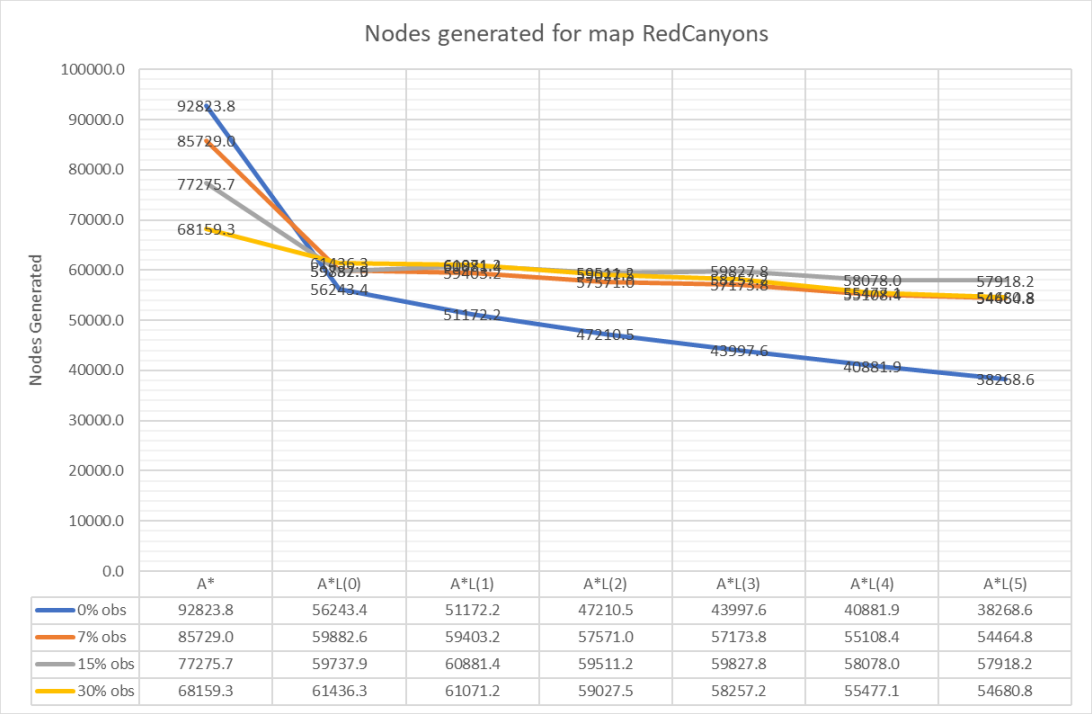


Figure 38:Nodes generated for 512 x 512 map

For normal A* search, the number of nodes generated decreases as the percentage of added obstacles increase. Added obstacles mean that there are less available traversable nodes in the map for A* search. As for A*L, as the value of k increases the total number of nodes generated decreases. This is true for any percentage of obstacles. And just as previous experiments, 0% obstacles have the largest percentage of memory saved for A*L.

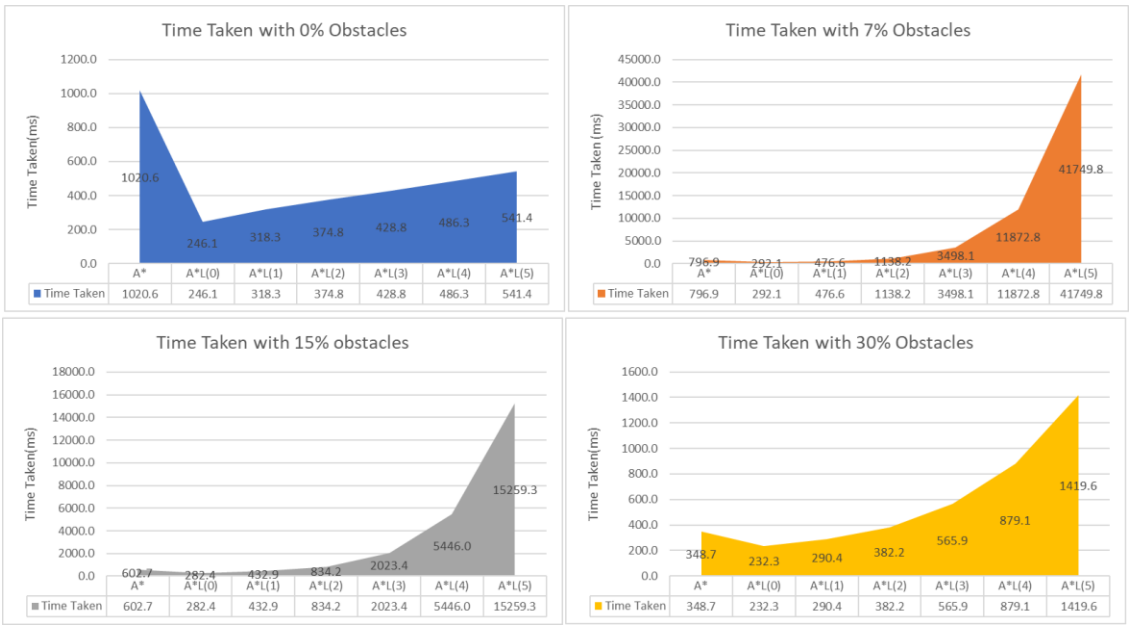


Figure 39: Time taken on 512 x 512 map

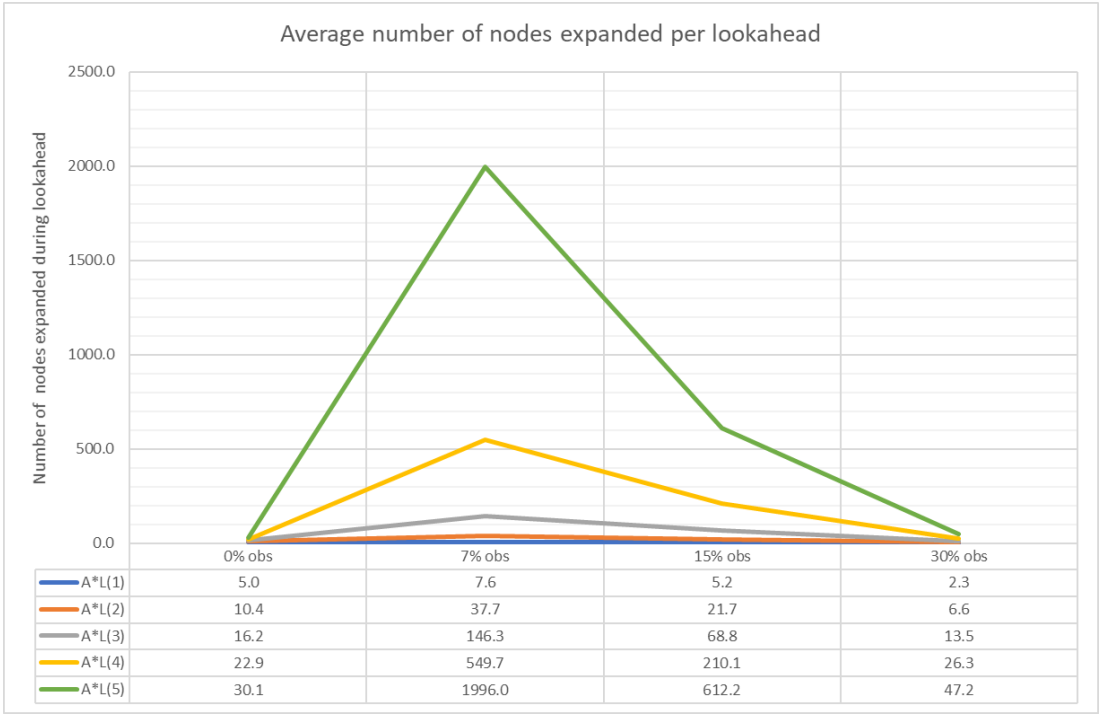


Figure 40:Avg. nodes expanded per lookahead for 512 x 512 map

The time taken at 0% obstacles is significantly lower even for $A * L(5)$, saving close to 47% in time taken. Startcraft maps are large maps with clusters of obstacles present in them.

Unlike uniformly and sparsely distributed obstacles, clustered obstacles remain pruned because to have a path around it a large detour must be taken, making them unlikely to be part of the shortest path. Just like previous maps, the average number of nodes added at the lookahead stage is highest when 7% obstacles are present resulting in larger amount of time taken.

*4.3 Unpruned vs Pruned A*L (50 x 50 map)*

The directional pruning technique is the method we use to get improvement in time taken for A*L. Without the pruning technique in A*L the time taken in significantly larger as the branching factor during DFS lookahead is exponential in nature. Because the time taken is significantly large, we are only able to perform experiments with a small sized map of size 50 x 50. We also restrict the number of experiments to 20 and the value of k to 0, 1 and 2. The start and end value for 20 experiments come from scenario files.

The map we used for this experiment is arena from Dragon Age Origins. The map has 2,054 traversable states. We do not add any extra obstacles to the map. This experiment showcases our problem statement of having an unpruned A*L and its comparison with pruned A*L. The start and end points for these experiments come from scenario files and are already available. Both algorithms have the same start and end nodes.

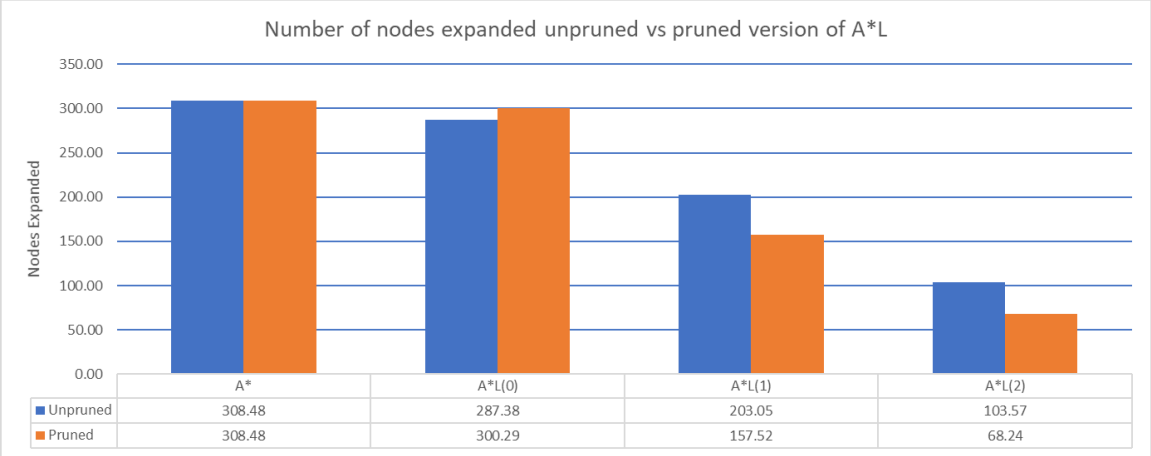


*Figure 41:Nodes expanded for pruned vs unpruned A*L*

The number of nodes expanded for A* is same because we don't use the pruning technique for A* search. The pruned version performs better than the unpruned version for the number of nodes expanded.

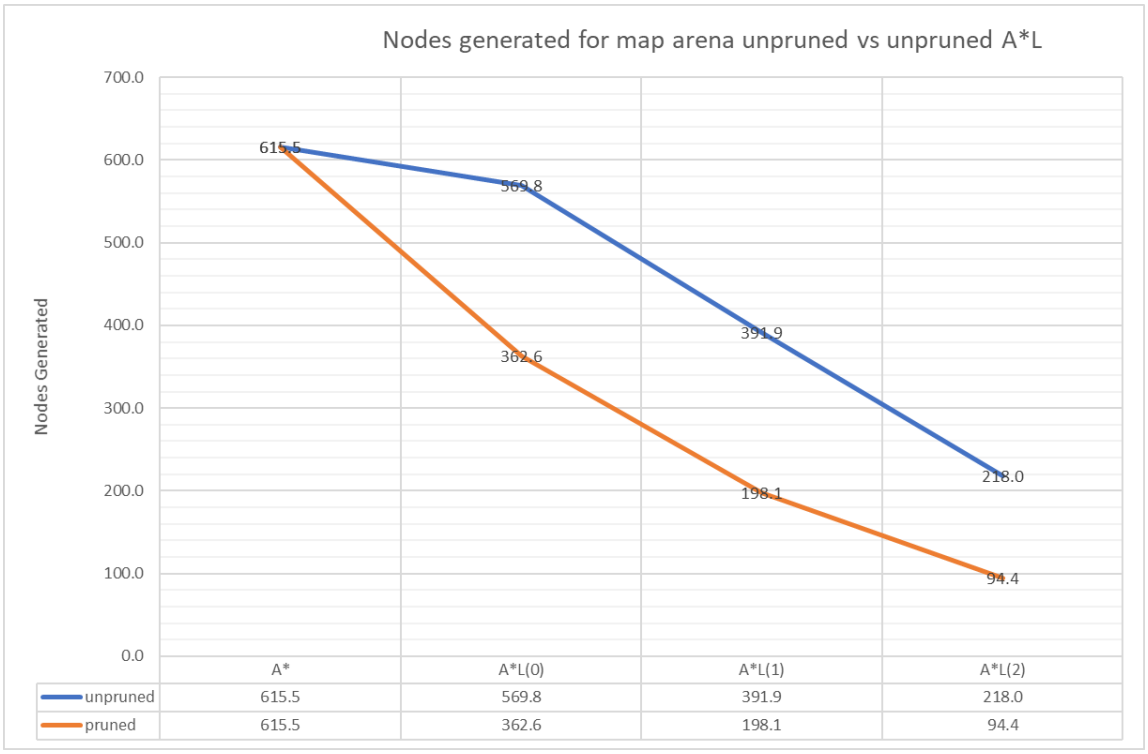


*Figure 42:Nodes generated for pruned vs unpruned A*L*

The pruned version generates lesser nodes than unpruned version at all versions of A*L. This is because the effective branching factor is lower for the pruned version. As the value of k increases, we save more on the number of nodes generated on both algorithms.

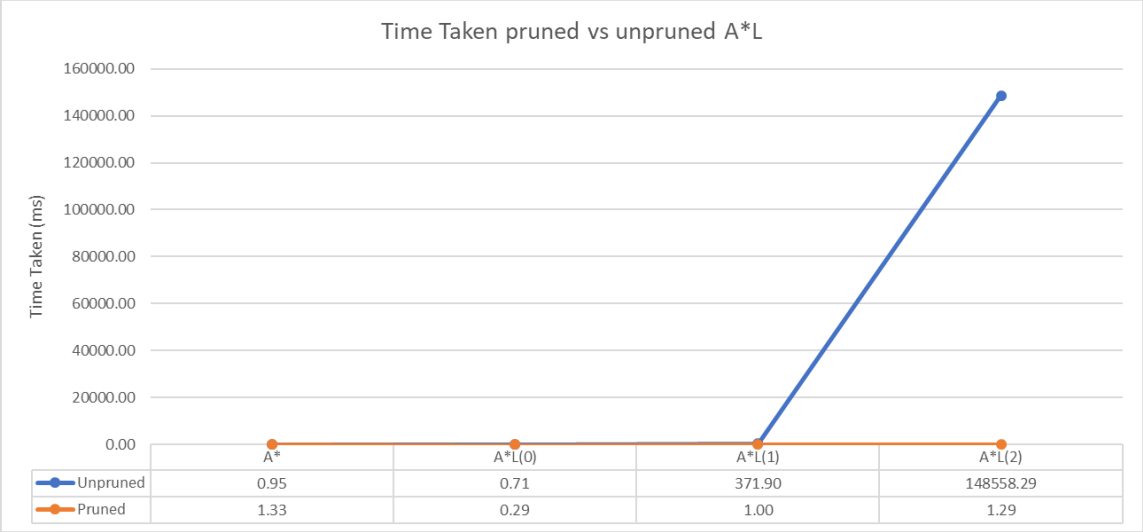


*Figure 43: Time taken on pruned vs unpruned A*L*

As the value of k increases for A*L, the time increases exponentially. The unpruned version of A*L(1) takes 371.9ms on average compared to 1ms average of pruned version. The unpruned version of A*L(2) takes over 148 seconds on average to run the algorithm.

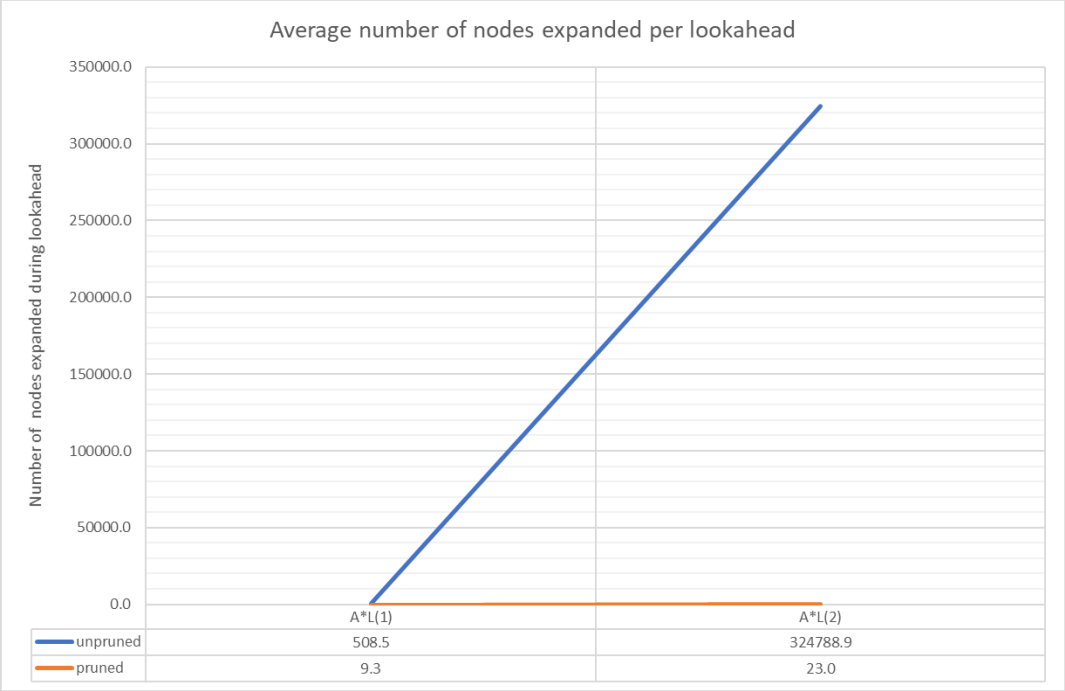


*Figure 44:Average nodes expanded per lookahead on pruned vs unpruned A*L*

As we can see from the diagram above, the number of nodes expanded at the lookahead stage of the unpruned A*L is significantly larger than the number nodes expanded at the lookahead stage of the pruned algorithm.

4.4 Summary

A*L saves more memory than A* search. As the value of k increases, the number of nodes generated decreases. This is the general trend for all experiments. As the cost of path increases, the percentage of nodes saved decreases. This is because the value of k relative to the cost of the path lowers as the cost of path increases. We see that for smaller maps there is a significant decrease in the percentage of nodes generated for A*L(5). Looking at the results for 128 x 128 map, as the percentage of obstacles increases, A*L starts to save a smaller number of nodes. At 30% obstacles for 128 x 128 map, the average path cost is

154 and the percentage saved is 45% whereas for the 211 x 251 map, at 0% the average path cost is 222.33 while the percentage saved is 46%.

The pruning technique also saves memory. If we consider the node generated graphs for A*L(0), we see that the number of nodes generated decrease. A*L(0) is similar to A* search where the lookaheads are done up to a cost of $f(current) + 0$. This will likely do lookaheads for very few nodes to almost no nodes. The pruning technique reduces the branching factor b of each node expanded from 8 to 2. The number of nodes expanded for A*L(0) is similar to that of A* search whereas the number of nodes generated significantly decreases.

Obstacles play a large role in how much memory is saved. When obstacles are clustered at certain locations it doesn't affect the memory saved as much as when the obstacles are distributed uniformly. Increase in obstacles uniformly decrease the heuristic performance. Decreasing heuristic performance increases the relative error of the heuristic function thereby making A* perform worse in terms of nodes expanded (Korf, 2000). For example, for completely blank map, if the path is exactly a diagonal the error in heuristic for Euclidean distance is 0. This means that when a lookahead search is done from the first node, even at k = 1 it will find the goal making the number of nodes expanded as low as 1. When obstacles are added along the path, the error in heuristic increases so more nodes need to be expanded to reach the goal. Larger number of nodes expanded means a greater number of nodes being generated. This is compounded by the increase in map size. When a percentage of obstacles are added to a larger map, the heuristic performs worse than when obstacles are added to a smaller map. This would explain the decrease in amount saved when obstacles are added.

A* with lookahead does not help with time taken. In fact, because the lookahead nodes are repeated often, they end up as an overhead. The time taken to expand singular node in lookahead part of the algorithm is a lot lower as they don't need to be saved, in fact they can be evaluated, expanded and discarded. They don't have operations on $open\ list$ and $closed\ list$. Even when a stack is used to store the best path the time taken for each operation is $O(1)$. For A*L(0) on all maps, the time taken is similar to or better than A* which has no lookahead nodes expanded. This shows that we can achieve better performance with regards to space and time when we use the pruning technique alongside A*L.

The key point is to keep the number of lookahead nodes low as they can grow exponentially. The directional pruning technique reduces the lookahead nodes expanded and nodes generated significantly. This directional pruning technique reduces the effective branching factor b_e from 8 to 2. This is also the reason at every experiment A*L(0) has better time taken than standard A* search. The directional pruning also reduces the branching factor for lookahead search, but more importantly it eliminates redundancy and symmetry in DFS too. When obstacles are present, forced neighbors are generated increasing the branching factor which increases the nodes generated. As more nodes are generated per expansion, more lookaheads are performed. And each lookahead has a higher branching factor because of obstacles.

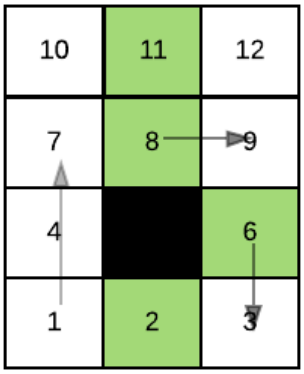


Figure 45: Generation of cycles in lookahead stage

Let us assume we want to expand node 7 with parent node 4. Node 8 and 11 are forced neighbors generated because of the obstacle whereas, 10 is the natural neighbor. If there was no obstacle at node 5, only node 10 would have been expanded. When node 9 needs to be expanded, node 6 generated as its forced neighbor. Node 2 is generated as forced neighbor of node 3 later and node 4 is generated as forced neighbor of node 1. This cycle does not take place with A* part of the algorithm because A* stores the visited and expanded nodes. However, even for A* search a greater number of nodes are generated on each of these expansions.

This affects the lookahead part of the algorithm. As the percentage of obstacles in the map increase, such singular random obstacles decrease. Large number of obstacles means that obstacles tend to cluster together densely, and such cycles don't happen often. High density of obstacles also means lesser number of nodes to be generated in the map. This is the reason why for all cases the time taken at 7% additional obstacle and 15% additional obstacle are the highest. This is also the reason why the time taken decreases at 30% obstacles chance at A*L(5).

We use the average number of nodes expanded per lookahead cycle to verify if this is true. For all maps, we see that as value of k increases the average number of DFS lookahead nodes expanded increase. We also see the average number of DFS lookaheads done is highest at 7% and 15% which proves our previous explanation of the effect of obstacles on lookahead.

CHAPTER 5:

Conclusion and Future Work

For our thesis, we explored a variant of A* search on the grid-based domain. This variant of A* called A* with Lookahead, allows us to save space in A* by doing depth-first lookaheads from the frontier of A* and generating new frontiers. The space complexity of the algorithm is $O(b^{d-l})$ compared to the space complexity of A* search which is $O(b^d)$. We proposed the use of this algorithm for grid-based pathfinding domain.

We found that using A* with lookahead for a grid-based domain increases the time complexity of the algorithm significantly. The time complexity of this algorithm is $O(b^l \times b_e^{d-l})$. The branching factor b for depth-first lookaheads tends to be larger than the effective branching factor b_e on A* search. We use an optimality preserving pruning technique to bring down both the branching factor and the effective branching factor of the algorithm. This pruning technique brings down the branching factor b from 8 to 2(average).

In our experiments with different sized maps, varying values of cost k and percentage of added obstacles, we found that adding obstacles to the map made A* with Lookaheads slightly worse in terms of the percentage of space saved. Furthermore, the presence of obstacles increased both branching factors of the algorithm thereby increasing the time taken.

To conclude, our approach of a combination of A* with Lookahead and the grid-based pruning scheme performs better than A* for all maps where the obstacles are naturally placed (0% added obstacles/ actual game maps). It performs the worst when obstacles are uniformly and sparsely placed (randomly placed obstacles with chance 7% and 15%). And

even when placed this way, there exists a value of k for which improvement in both time and space complexity can be achieved. There is a slight improvement at $k = 1$ and more space saved at $k = 2$ with a slightly worse time taken.

When experimenting with an unpruned version of A* with Lookahead, it wasn't possible to run experiments with higher values of k due to time constraints. We conclude that the unpruned version is therefore not a feasible approach even though it also manages to save space.

Future work could be to randomly generate clusters of obstacles with a 7% and 15% chance and see its effect on the performance of the algorithm. The pruning technique seems to significantly improve the runtime performance and would be promising to apply to other variants of A* that use iterative deepening to save space like IDA* and RBFS.

We store the best path found so far up to the goal nodes. If we store the best path found regardless of the node, we might be able to reuse lookaheads from the previous search drastically reducing the time taken by the algorithm. This could, however, come at a cost of increased memory requirements.

APPENDICES

	Obstacle Chance %	A* average path	Nodes Expanded	Nodes Generated	Lookahead Nodes expanded	Time Taken(ms)
A* Search	0.0	523.4	13015.3	19150.5	0.0	59.1
	7.0	530.9	12333.4	17239.6	0.0	47.8
	15.0	554.7	11641.7	15174.8	0.0	38.2
	30.0	664.4	8676.2	9901.2	0.0	20.8
A* with Lookahead k =0	0.0	523.4	12917.5	13008.8	9.0	25.8
	7.0	530.9	12160.3	13241.9	3.5	28.1
	15.0	554.7	11490.1	12560.7	3.3	25.9
	30.0	664.4	8641.7	9043.0	2.0	15.4
A* with Lookahead k =1	0.0	523.4	10534.5	11022.4	32079.5	31.8
	7.0	530.9	11055.7	13025.1	60476.1	51.7
	15.0	554.7	10409.5	12663.4	43431.6	44.9
	30.0	664.4	7281.4	8937.6	17952.1	25.6
A* with Lookahead k =2	0.0	523.4	9308.2	10123.7	57992.6	43.1
	7.0	530.9	10044.2	12674.8	234679.6	126.0
	15.0	554.7	9243.2	12388.7	154597.2	95.5
	30.0	664.4	5940.9	8574.0	46711.9	39.5
A* with Lookahead k =3	0.0	523.4	8619.1	9414.5	76922.7	49.2
	7.0	530.9	9756.1	12505.9	688008.8	301.6
	15.0	554.7	8997.2	12309.0	413320.9	210.7
	30.0	664.4	5636.6	8338.1	87345.7	57.7
A* with Lookahead k =4	0.0	523.4	7942.6	8713.1	95536.4	52.2
	7.0	530.9	9241.9	12057.5	1851251.0	748.2
	15.0	554.7	8547.9	11889.1	1023738.6	467.1
	30.0	664.4	5262.8	7894.3	149684.1	90.4
A* with Lookahead k =5	115.7	229.9	285.5	13886.2	5.3	58.1
	118.1	448.5	631.3	422605.1	179.8	1856.3
	124.1	809.9	1172.5	368324.9	168.0	1064.7
	154.2	1049.3	1592.7	67952.7	39.5	137.2

Table 2:Full table of results for 128x128 map

	Obstacle Chance %	A* average path	Nodes Expanded	Nodes Generated	Lookahead Nodes expanded	Time Taken(ms)
A* Search	0.0	222.3	7316.3	11934.0	0.0	54.9
	7.0	224.4	7116.5	10840.1	0.0	44.6
	15.0	230.6	7115.6	9940.4	0.0	37.5
	30.0	272.4	7593.8	8965.2	0.0	27.8
A* with Lookahead k =0	0.0	222.3	7130.3	7297.6	0.6	16.2
	7.0	224.4	6786.1	7600.4	7.2	17.7
	15.0	230.6	6815.0	7669.2	4.7	18.0
	30.0	272.4	7487.4	8018.3	2.6	18.7
A* with Lookahead k =1	0.0	222.3	6233.0	6489.8	52099.1	35.3
	7.0	224.4	5891.8	7222.4	64556.5	46.2
	15.0	230.6	5949.2	7590.9	42647.8	37.7
	30.0	272.4	6303.7	7926.5	19146.5	27.6
A* with Lookahead k =2	0.0	222.3	5513.5	5797.0	79915.1	42.7
	7.0	224.4	5243.5	6726.2	298542.7	141.2
	15.0	230.6	5254.7	7181.8	163940.3	95.7
	30.0	272.4	5236.9	7566.5	52226.7	42.8
A* with Lookahead k =3	0.0	222.3	4902.2	5169.3	96531.9	46.2
	7.0	224.4	4934.9	6400.2	1126917.4	447.3
	15.0	230.6	5071.7	7013.6	497063.7	233.3
	30.0	272.4	5053.0	7427.7	105090.1	70.0
A* with Lookahead k =4	0.0	222.3	4344.5	4593.7	108072.2	47.0
	7.0	224.4	4464.7	5870.4	4033662.2	1541.0
	15.0	230.6	4668.4	6536.3	1409902.0	612.2
	30.0	272.4	4711.0	7011.7	192655.8	113.3
A* with Lookahead k =5	0.0	222.3	3907.2	4139.3	119132.4	50.1
	7.0	224.4	4189.0	5498.5	14272515.9	5387.3
	15.0	230.6	4425.8	6195.9	3876279.0	1644.6
	30.0	272.4	4580.3	6824.6	338879.8	183.4

Table 3:Full table of results for 211x251 map

	Obstacle Chance %	A* average path	Nodes Expanded	Nodes Generated	Lookahead Nodes expanded	Time Taken(ms)
A* Search	0.0	382.2	12495.5	20226.7	0.0	115.5
	7.0	390.0	12587.1	18823.2	0.0	90.4
	15.0	409.6	12693.8	17248.0	0.0	68.3
	30.0	505.2	11145.7	12828.3	0.0	34.8
A* with Lookahead k =0	0.0	382.2	12035.6	12273.4	23.5	34.7
	7.0	390.0	11976.3	13114.7	41.6	38.4
	15.0	409.6	12283.8	13502.8	21.8	36.6
	30.0	505.2	11057.4	11620.6	7.7	25.1
A* with Lookahead k =1	0.0	382.2	10097.5	10863.1	56362.3	55.1
	7.0	390.0	10210.5	12611.4	82122.3	76.5
	15.0	409.6	10553.1	13419.5	60087.5	68.0
	30.0	505.2	9143.4	11358.5	24579.4	38.6
A* with Lookahead k =2	0.0	382.2	8453.9	9336.1	121130.2	72.8
	7.0	390.0	8984.8	11930.0	348532.3	185.7
	15.0	409.6	9300.8	12962.4	223672.3	140.1
	30.0	505.2	7497.0	10941.8	66753.4	56.3
A* with Lookahead k =3	0.0	382.2	7277.7	8107.7	175494.5	86.5
	7.0	390.0	8626.6	11628.3	1166550.8	501.5
	15.0	409.6	9217.1	12947.9	646106.2	321.5
	30.0	505.2	7275.3	10759.7	129865.6	90.4
A* with Lookahead k =4	0.0	382.2	6197.4	6982.0	231862.7	100.2
	7.0	390.0	7887.1	10828.9	3645561.4	1466.3
	15.0	409.6	8627.0	12292.8	1719055.4	776.7
	30.0	505.2	6708.4	10078.0	229650.3	138.3
A* with Lookahead k =5	0.0	382.2	5357.8	6094.5	274692.4	111.6
	7.0	390.0	7656.0	10441.4	11204801.3	4414.9
	15.0	409.6	8547.1	12121.4	4471608.5	1939.6
	30.0	505.2	6592.6	9901.2	387755.4	215.1

Table 4:Full table of results for 320x320 map

	Obstacle Chance %	A* average path	Nodes Expanded	Nodes Generated	Lookahead Nodes expanded	Time Taken(ms)
A* Search	0.0	523.4	13015.3	19150.5	0.0	59.1
	7.0	530.9	12333.4	17239.6	0.0	47.8
	15.0	554.7	11641.7	15174.8	0.0	38.2
	30.0	664.4	8676.2	9901.2	0.0	20.8
A* with Lookahead k =0	0.0	523.4	12917.5	13008.8	9.0	25.8
	7.0	530.9	12160.3	13241.9	3.5	28.1
	15.0	554.7	11490.1	12560.7	3.3	25.9
	30.0	664.4	8641.7	9043.0	2.0	15.4
A* with Lookahead k =1	0.0	523.4	10534.5	11022.4	32079.5	31.8
	7.0	530.9	11055.7	13025.1	60476.1	51.7
	15.0	554.7	10409.5	12663.4	43431.6	44.9
	30.0	664.4	7281.4	8937.6	17952.1	25.6
A* with Lookahead k =2	0.0	523.4	9308.2	10123.7	57992.6	43.1
	7.0	530.9	10044.2	12674.8	234679.6	126.0
	15.0	554.7	9243.2	12388.7	154597.2	95.5
	30.0	664.4	5940.9	8574.0	46711.9	39.5
A* with Lookahead k =3	0.0	523.4	8619.1	9414.5	76922.7	49.2
	7.0	530.9	9756.1	12505.9	688008.8	301.6
	15.0	554.7	8997.2	12309.0	413320.9	210.7
	30.0	664.4	5636.6	8338.1	87345.7	57.7
A* with Lookahead k =4	0.0	523.4	7942.6	8713.1	95536.4	52.2
	7.0	530.9	9241.9	12057.5	1851251.0	748.2
	15.0	554.7	8547.9	11889.1	1023738.6	467.1
	30.0	664.4	5262.8	7894.3	149684.1	90.4
A* with Lookahead k =5	0.0	523.4	7493.5	8243.7	112280.6	58.1
	7.0	530.9	9022.7	11834.8	4725537.7	1856.3
	15.0	554.7	8415.0	11748.3	2447115.5	1064.7
	30.0	664.4	5120.3	7703.6	244573.0	137.2

Table 5:Full table of results for 384x384 map

	Obstacle Chance %	A* average path	Nodes Expanded	Nodes Generated	Lookahead Nodes expanded	Time Taken(ms)
A* Search	0.0	628.4	57202.3	92823.8	0.0	1020.6
	7.0	636.7	56734.5	85729.0	0.0	796.9
	15.0	655.9	55879.4	77275.7	0.0	602.7
	30.0	793.9	58635.2	68159.3	0.0	348.7
A* with Lookahead k =0	0.0	628.4	55684.7	56243.4	154.4	246.1
	7.0	636.7	54070.8	59882.6	102.0	292.1
	15.0	655.9	53628.5	59737.9	69.3	282.4
	30.0	793.9	58028.7	61436.3	25.8	232.3
A* with Lookahead k =1	0.0	628.4	49537.4	51172.2	257862.1	318.3
	7.0	636.7	48840.2	59403.2	449161.4	476.6
	15.0	655.9	48046.2	60881.4	317762.3	432.9
	30.0	793.9	49100.2	61071.2	140279.7	290.4
A* with Lookahead k =2	0.0	628.4	45274.5	47210.5	488980.3	374.8
	7.0	636.7	44975.3	57571.0	2171024.6	1138.2
	15.0	655.9	43699.4	59511.2	1289564.0	834.2
	30.0	793.9	41085.5	59027.5	389189.2	382.2
A* with Lookahead k =3	0.0	628.4	42123.4	43997.6	712610.8	428.8
	7.0	636.7	44089.8	57173.8	8365801.8	3498.1
	15.0	655.9	43467.9	59827.8	4116165.9	2023.4
	30.0	793.9	39880.5	58257.2	788246.5	565.9
A* with Lookahead k =4	0.0	628.4	39065.7	40881.9	935621.1	486.3
	7.0	636.7	41857.5	55108.4	30292340.5	11872.8
	15.0	655.9	41661.3	58078.0	12199405.1	5446.0
	30.0	793.9	37489.0	55477.1	1457087.6	879.1
A* with Lookahead k =5	0.0	628.4	36505.0	38268.6	1151016.8	541.4
	7.0	636.7	41563.3	54464.8	108711056.4	41749.8
	15.0	655.9	41685.3	57918.2	35457524.8	15259.3
	30.0	793.9	36943.0	54680.8	2579114.2	1419.6

Table 6: Full table of results for 512x512 map

	Type	A* average path	Nodes Expanded	Nodes Generated	Lookahead Nodes expanded	Time Taken(ms)
A* Search	Unpruned	47.7	308.5	615.5	1.0	1.0
	Pruned	47.7	308.5	615.5	1.3	1.3
A* with Lookahead k =0	Unpruned	47.7	287.4	569.8	4.4	0.7
	Pruned	47.7	300.3	362.6	3.4	0.3
A* with Lookahead k =1	Unpruned	47.7	203.0	391.9	199271.7	371.9
	Pruned	47.7	157.5	198.1	1844.9	1.0
A* with Lookahead k =2	Unpruned	47.7	103.6	218.0	70788507.9	148558.3
	Pruned	47.7	68.2	94.4	2167.6	1.3

Table 7: Full table of results for pruned vs unpruned

REFERENCES

Adi Boeta, B. B. (1998). Pathfinding in Games. *Artificial and Computational Intelligence in Games*.

Dainel Harabor, A. G. (2012). The JPS Pathfinding System. *Proceedings of the Fifth Annual Symposium on Combinatorial Search* (pp. 207-208). AAAI.

Daniel Damir Harabor, A. G. (2011). Online Graph Pruning for Pathfinding on Grid Maps. *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence* (pp. 1114-1119). Association for the Advancement of ArtificialIntelligence.

Daniel Harabor, A. B. (2011). Grids, Path Symmetries in Undirected Uniform-Cost. *Proceedings of the Ninth Symposium on Abstraction, Reformulation and Approximation* (p. Daniel HaraborandAdi BoteaandPhilip Kilby). Association for the Advancement of ArtificialIntelligence.

Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics SSC4*.

Holte, R. C. (2010). Common Misconceptions Concerning Heuristic Search. *Association for the Advancement of Artificial Intelligence*.

https://movingai.com/benchmarks/formats.html. (n.d.). Retrieved from https://movingai.com/benchmarks/formats.html.

https://qiao.github.io/. (n.d.). Retrieved from https://qiao.github.io/PathFinding.js/visual/.

Korf, R. E. (2000). Recent Progress in the Design and Analysisof Admissible Heuristic Functions. *Abstraction, Reformulation, and Approximation: 4th International Symposium* (pp. 45–55). Horseshoe Bay, USA: AAAI.

Nareyek, A. (2004). AI in Computer Games. *Queue*, ACM.

Norvig, S. J. (2010). *Artificial Intelligence: A Modern Approach.* Pearson Education Limited 2016.

P.Mehta, S. V. (2015). A Review on Algorithms for Pathfinding in Computer Games. *International Conference on Innovations in Information Embedded and Communication Systems ICIIECS'15.* IEEE.

Patel, A. (2010). Retrieved from http://theory.stanford.edu/~amitp/GameProgramming/.

Roni Stern, T. K. (2010). Using Lookaheads with Optimal Best-First Search. *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence* (pp. 185-190). Association for the Advancement of Artificial.

Sturtevant, N. R. (2012). Benchmarks for Grid-Based Pathfinding. *Transactions on Computational Intelligence and AI in Games*, 144-148.

Zhaoxing Bu, R. S. (2014). A* with Lookahead Re-Evaluated. *Proceedings of the Seventh Annual Symposium on Combinatorial Search* (pp. 44-52). Association for the Advancement of Artificial.

VITA AUCTORIS

NAME:	Shrijan Karmacharya
PLACE OF BIRTH:	Kathmandu, Nepal
YEAR OF BIRTH:	1991
EDUCATION:	Bachelor of Engineering - Information Science and Engineering, Visveshvaraya Technological University, Bangalore, Karnataka, India, 2014 University of Windsor, M. Sc C.S, Windsor, Ontario, Canada, 2020