Pathfinding and Exploration

Carlos Alvarenga
alvar357@umn.edu

November 19, 2018

Abstract

Pathfinding and exploration are important concepts within Artificial Intelligence (AI). As a result, various uninformed and informed pathfinding algorithms have been devised and studied in order to find the most efficient path to traverse when trying to reach a goal state in various types of environments. This project is focused on comparing different types of pathfinding algorithms when trying to solve NBA "name chaining problem" (NCP). Section 1 describes the NBA NCP and Section 2 provides a brief review of related work about algorithms and approaches that may be used to solve the problem. Section 3 describes the approach and algorithms used to solve the problem. Section 4 provides a description of the experiment design and results and section 5 contains an analysis of the experiment results. Finally, section 6 summarizes conclusions garnered from the experiment and describes any related future work that may be done in solving the problem.

1 Introduction

The problem that's being solved in this project is the NBA "Name Chaining Problem" (NCP). Although it isn't one discussed in academia, it's one discussed by avid NBA fans such as myself and is related to AI topics like graph exploration and pathfinding.

The NCP involves finding the longest name that can be formed among NBA players by "chaining" their names as follows: the last name of player 1, which must be identical to Player 2's first name, is combined with player 2's first and last name to form a new name. For example suppose we have 4 players named "Bob Smith" (player 1), "Smith James" (player 2), "James Olsen" (player 3), and "Adam Johnson", respectively. Since player 1's last name is identical to player 2's first name, these two names can be chained to form "Bob *Smith* James". Moreover, since player 2's last name is identical to player 3's first name, the chained name to be increased to include player 3's last name (i.e "Bob Smith James Olsen"). However, player 4's name can't be chained to any of the names since its first name doesn't match the last name of any player and its last name doesn't match the first name of any player. The NCP involves doing this process to find the longest chained name in terms of the number of NBA players whose names can be chained together (not number of characters within a name string). Moreover, the problem can be either solved for only current NBA players or all past and current NBA players. For the purpose of this experiment, the problem was solved for both data sets to obtain more results for analysis.

The NCP's relation to AI is what makes it the most interesting because it's a practical situation where Computer Science algorithms and logic can be applied for problem solving purposes. Specifically, the NCP is related to AI topics like exploration and pathfinding because pathfinding

algorithms can be devised to explore data (i.e. names) to find the longest path (i.e. longest chained name) or path between specific names (i.e. optimal path between the first and final names in a chained name).

2 Related Work

2.1 Introduction

The name chaining problem (NCP) is one that's only considered and discussed by avid NBA fans who frequent the nba subreddit. However, the problem is related to various graph search and pathfinding problems and there are relevant studies on algorithmic search methods that are pertinent to the NCP study. The algorithms employed for this study involve uninformed search algorithms like breadth-first search (BFS), depth-first search (DFS), and iterative deepening depth-first search (or iterative deepening search) (IDS), that are used for unweighted graphs. The NCP study also employs the uninformed search algorithm, Dijkstra's, that's used for weighted graphs. Lastly, the study employs the informed search algorithms A* and iterative deepening A* (IDA*) that are used for weighted graphs. All these mentioned algorithms have been studied and modified to improve their performance in certain scenarios.

2.2 Longest Path Problem

2.2.1 Complexity

In the NCP study, the longest path (i.e. chained name) was found by simply finding all relevant paths and picking the longest path from the set of all paths. A lot study has been done on efficiently solving the longest path problem (LPP), which involves finding a path of maximum length in a given graph. As [5] states, despite all the studies done to solve the problem, approximating the longest path in undirected graphs remains NP-hard. Although [5] provides a greedy algorithm to find long paths in dense graphs, finding the Hamiltonian path (i.e. the path between two vertices of a graph that visits each vertex exactly once) in polynomial time is NP-complete.

2.2.2 Travelling Salesman Problem

The travelling salesman problem (TSP) is one of the most famous Computer Science problems and is related to the LPP. In [4], the authors argue that the TSP can be considered a special case of the LPP. This is due to the fact that the TSP can always be reduced to the problem of finding the longest path between two nodes in a finite, connected graph [4]. Although no algorithm currently exists to solve the LPP, [4] provides "cycle" and "tour" solutions. The cycle version of the TSP involves a graph G that has the following properties: G is finite, connected (but not necessarily complete), and has n nodes. In the tour version of the TSP, The graph G' is constructed from G in that the nodes of G' are the nodes of G. In G', nodes i and j are joined by a directed arc (i,j) if and only if G contains a certain directed path between nodes i and j. The lengths $d'(i,j)$ in G' are the lengths of a shortest path in G between nodes i and j. So, [4] concludes that finding the longest path in G solves the cycle version of the TSP and finding the longest path in G' solves the tour version of the TSP.

2.2.3 Solvable Graphs

There are very few graph classes where the LPP can be efficiently solved. In [10], the authors note that Dijsktra's algorithm can be used to used to find the longest path in a tree in polynomial time. Moreover, a graph class where the Hamiltonian path can be found in polynomial time is a linear graph [10]. Some examples of linear graphs include bipartite permutation and interval biconvex graphs. Finally, [10] provides an extension of Dijkstra's algorithm that can find the longest path (in linear time) in weighted trees and block graphs.

2.3 Uninformed Pathfinding

2.3.1 Boundary Iterative-Deepening Depth-First Search

BFS, DFS, IDS and Dijkstra's are the uninformed search algorithms employed in the experiment to solve the NCP. In [8], the authors provide an alternative uninformed pathfinding algorithm called Boundary iterative-deepening depth-first search (BIDS). BIDS consumes less memory than Dijkstra's algorithm. The expansion redundancy of IDS is also compensated and, consequently, it performs faster than IDS in [8]'s tests. The authors also propose modified BIDS algorithms called bidirectional BIDS and parallel bidirectional BIDS. The former is enhanced for bidirectional searching to expand fewer nodes and reduce the pathfinding run-time. Furthermore, BIDS is enhanced to be where it can find multiple goals on the graph in a single search.

2.3.2 Breadth-First Search Modifications

BFS and DFS are the most rudimentary algorithms employed to solve the NCP and modifications of both these algorithms have been proposed to improve performance in certain situation. In [9], the authors propose a modified BFS algorithm that combines a conventional top-down algorithm along with a novel bottom-up algorithm. The authors demonstrate how performing BFS in a bottom-up fashion, as opposed to a more conventional top-down approach, can dramatically reduce the number of edges traversed. This is due to the fact that a child node needs to find only one parent node instead of each parent having to attempt finding all its children. This bottom-up approach would be useful in searching a large connected component of a low-effective diameter of a graph while a top-down approach would fare better in the beginning and end of such search [9]. The provided hybrid BFS approach combines the two algorithms and uses a heuristic to guide it when to switch between top-down and bottom-up algorithmic approaches. This hybrid scheme then yields the best possible BFS performance for single shared-memory nodes.

2.3.3 Alternatives to Iterative Deepening Search and Depth-First Search

Similar to BFS, modifications of DFS have been proposed for better performance in certain scenarios. In [11], the authors compare the following three heuristic search algorithms: best-first search (B-FS), IDS, and depth-first branch-and-bound (DFBB). Moreover, a modified DFS algorithm called DFS*, which is a hybrid of of IDS and DFBB, is proposed. The effectiveness of the three algorithms are compared in relation to the heuristic branching factor and solution density of the problem. It's concluded by [11] that DFBB performs the best when the solution density is larger than the heuristic branching factor and IDS performs the best when the heuristic branching factor is high and the solution density is low. Both of these algorithms can solve larger problems than

B-FS can because they overcome B-FS' memory limitation since they use a a depth-first search strategy. BFS is useful only when the solution density and heuristic branching factor are extremely low. Ultimately, the proposed hybrid DFS* algorithm represents an upgrade over the three studied algorithms under certain situations. DFS* is a depth-first, admissible search strategy that finds optimal solutions given non-overestimating heuristics [11]. It initially behaves like IDS, but freely increases the cost bounds to minimize repeated node expansions. When a non-optimal solution is found or if the cost bound selected in any iteration of the IDS phase is greater than an alternate upper-bound, DFS* switches over to the DFBB algorithm. Most importantly, DFS* is most useful and performs better than the three studied algorithms when both the heuristic branching factor and solution density are low.

2.3.4 Iterative Deepening Search

IDS is one of the uninformed search algorithms used in the NCP study because it doesn't suffer from the drawbacks of BFS and DFS. As [6] states, BFS can require too much space and while DFS avoids BFS' memory limitations, it can take a lot of time to terminate and may not always find the cheapest path. In [6], the author compares IDS to BFS and DFS to gauge it's performance and benefits and he also claims that IDS is asymptotically optimal along all three dimensions for exponential tree searches. IDS is guaranteed to find the shortest path because it expands all nodes at a given depth before expanding nodes at a greater depth. However, [6] cites some disadvantages of IDS such as its wasted computation prior to reaching the goal and that, similar to DFS in certain graphs, it has to explore all possible paths to a given depth. Nonetheless, IDS is asymptotically optimal compared to BFS and DFS. IDS can be modified and applied to bi-directional search, heuristic best-first search, and two-person game searches [6].

2.4 Informed Pathfinding

2.4.1 A* Modifications

The A* algorithm is widely used due to its use of heuristic functions to efficiently find optimal paths. In [1], the main focus is A* and its different optimized versions. A* repeatedly examines the most promising unexplored location it's seen and continues exploring this location's neighbors if the explored location is not the goal. As [1] notes, A* is applied in different ways depending on the setting and reducing the search space significantly speeds up A*. Several potential optimizations of A* from four different perspectives are described. The first optimized A* is Hierarchical Pathfinding A* (HPA*), which involves breaking down the world hierarchically. The hierarchy is divided into different levels where pathfinding is carried out. The potential problem of this technique is that the cost increases as new layers are added. The second optimized A* is Navigation Mess (NavMesh), which is a popular pathfinding technique in 3D environments. NavMesh is guaranteed to find a near optimal path by searching much less data. The third optimized A* is from a heuristic function perspective, as opposed to a search space perspective as the two previously discussed A* versions are. It's important that the heuristic be "good" in the sense that it never overestimates the true cost, thereby resulting in faster search. Another type of A* optimization is viewed from a memory perspective. In this technique, [1] explains that A* requires a lot of memory to track the progress of the searches. The authors explain how the most popular way to avoid memory waste is to pre-allocate a minimum amount of memory before executing A*. Another technique to reduce memory requirements in A* is to compute the whole path in small pieces, as done in IDA*. The

last A* optimization technique is from a data structures perspective and it involves storing the search graph nodes in efficient data structures, such as a hash table, that support constant time storing and look-up. Moreover, a closed and open list would need to be maintained for the nodes and [1] notes how a priority queue implemented by a binary heap is the best way to maintain an open list.

2.4.2 Iterative Deepening A*

IDS can be combined with A* and [6] does so when proposing Iterative Deepening A* (IDA*). Not only does IDA* find a cheapest path to a solution and uses less space than A*, but it also expands approximately the same number of nodes as A* in a tree search. However, [7] points out the worst-case of IDA* where it'll expand all nodes n whose cost $f(n) = g(n) + h(n) \leq$ *cost threshold*. Moreover, since IDA* does not utilize dynamic programming, it can often end up exploring the same nodes many times and this would make it perform worse than A* under scenarios where it's forced to restart the search many times.

2.4.3 Dynamic Repairing A*

In [2], the authors introduce the idea of re-planning, which is a type of planning that occurs when the current plan either no longer satisfies prerequisites or when some future actions can't be executed. During re-planning, a new plan has to be produced and this can be categorized into two categories, re-planning from scratch and plan repairing. Furthermore, [2] compares A* and a modified version of it called Dynamic Repairing A* (DRA*), which is an algorithm that uses re-planning from scratch and plan-repairing. DRA* is mentioned as an extended version of A* that is useful in situations found in dynamic terrains where the goal-set and the actions costs are modified. Additionally, DRA* addresses two changes in said dynamic environment, goal-set modifications and actions' costs alterations. As the authors note, The overall performance of DRA* depends on the average branching factor, with higher branching factors resulting in worse performance. The conducted experiments described in the paper reveal that DRA* outperforms A* when the following two conditions hold: the percentage of the original plan that has already been executed isn't greater than 40 to 50 percent and the change in the goal-set is not greater than 20 to 50 percent. In other words, DRA* performs best when the changes in the environment are small or moderate. DRA* also outperformed A* in all experiments conducted in re-planning scenarios of modified costs.

2.4.4 Alternatives to A*

Although [3] focuses on explaining the A* algorithm and its place in solving complex pathfinding problems, other pathfinding algorithms like Dijkstra's, BFS, DFS, uniform cost search and heuristic search are discussed. As [3] notes, pathfinding can be divided into two categories, undirected and directed. When considering an undirected approach, BFS and DFS are useful algorithms to solve pathfinding problems. On the other hand, uniform cost search, heuristic search and Dijkstra's algorithm can be used to solve pathfinding problems with directed approaches. In addition to discussing A* and directed and undirected algorithmic approaches, [3] discusses the limitations of current pathfinding techniques with a particular emphasis on handling collisions or interference with dynamic obstacles. This limitation, however, can be solved by "learning algorithms" [3]. The most notable learning algorithm that [3] discusses is the genetic algorithm and it's emphasized how

these advanced learning algorithms can be used to learn pathfinding behaviors, thereby overcoming the limitations of traditional pathfinding algorithms.

2.5 Conclusion

A lot of research and experimentation has been done on the LPP and pathfinding algorithms like BFS, DFS, IDS, Dijkstra's, A* and IDA*. These algorithms have been compared against each other and modified to improve their respective performance in different scenarios that involve different graphs. As a result, there is a general notion of how these algorithms will fare against each other in the NCP experiment. Moreover, the discussed research papers also informed which algorithms are the most feasible and conducive towards solving the NCP and gaining valuable insight about the different approaches implemented in solving said problem.

3 Approach

3.1 Overview

The NCP is solved twice, with each solution corresponding to one of the following data sets:

- Small data set: Information, including full names, of all current nba players.
- Large data set: Information, including full names, of all former and current nba players.

All code for this project was written in the Python language, but before running any algorithms to obtain results, relevant data about the players is extracted from NBA.com and is used to construct the graphs that will be used to run the algorithms. Finally, the project approach can be divided into the following three stages:

1. Use the uninformed algorithms breadth-first search (BFS) and depth-first search (DFS) on unweighted graphs to solve the "Name chaining problem" (NCP) by finding the longest chained name (LCN) in both the small and large data sets.
2. Once the LCN has been found and using only the large data set, the uninformed algorithms BFS, DFS and iterative deepening depth-first search (IDS) are performed on unweighted graphs to find the LCN path to compare their efficiency and performance.
3. Once the LCN has been found and using only the large data set, the uninformed Dijkstra's algorithm, or simply Dijkstra's, and the informed algorithms A* and iterative deepening A* (IDA*) are performed on unweighted and weighted graphs to find the LCN path to compare their efficiency and performance under different types of graph environments, namely unweighted and weighed.

3.2 Stage One: Exploration

3.2.1 Unweighted Graph Construction

After parsing and extracting the players names from the data obtained from NBA.com and storing these names in a list, this newly formatted data is used to construct an unweighted graph. This is done using adjacency lists as follows: for each name x in the data set, the rest of the data set is

search and any name y in this set that has an identical first name as x's last name is added to x's adjacency list. So, an unweighted edge (i.e no cost value associated with the edge (x,y)) is created between x and y. After this process is completed, the graph G will only contain names that have a last name or first name in common with another name.

3.2.2 Breadth-First Search

After constructing the unweighted graph G, BFS can be performed on it. For this stage, BFS is run to find all valid chained names in G. This is done by running the BFS algorithm described in the appendix, section 7.1. Assume the following:

- Let x be a node in the graph G
- Let $S = y_1,y_2,...,y_k$ where k is the number of nodes in the graph minus 1 and the node x is not in S

Then, for each node x, BFS() is run for each node $y \in S$ where x will be the start and y will be the goal node. So, assuming there are k nodes in the graph, BFS() will be run $k*k-1$ times to explore and find all possible paths, or chained names, in the graph.

3.2.3 Depth-First Search

After running BFS, the DFS algorithm described in the appendix, section 7.2 is also run in the exact same way BFS() was described to be run in section 3.2.2. So, DFS and BFS are run the same amount of times with the same (start, goal) node pairs and same unweighted graph.

3.2.4 Longest Path

After performing BFS, all valid paths (i.e. name chains) found in the algorithm are stored in a list. Then, this list is search to find the longest path (i.e. the longest chained name). This is also done after performing DFS in order to assert correctness by confirming that both BFS and DFS find the same paths and contain the correct LCN.

3.3 Stage Two: Unweighted Graph Pathfinding

Unweighted graph construction is done identically to how it's done in the exploration stage (section 3.2). Moreover, since BFS and DFS require a start and goal node and the LCN has already been found in the exploration stage, the start node x will corresponds to the first name in the LCN and the goal node y will corresponds to the final name in the LCN. Having constructed the graph and extracted the starting and final nodes from the LCN, these three components are used to perform BFS and DFS to find the correct LCN named that begins with x and ends with y. The IDS algorithm is also run in this stage to compare more uninformed pathfinding algorithms. This algorithm is described in the appendix, section 7.3 and is run with the same parameters and information as BFS and DFS. The only additional information that IDS requires is the maximum depth and this is obtained by calculating the length of the LCN.

3.4 Stage Three: Weighted Graph Pathfinding

3.4.1 Graph Construction

Parsing and extraction of names from the data obtained from NBA.com is done similar to how it's done in unweighted graph construction (section 3.2.1). Names are also added to each others' adjacency list (when they can be chained) as done in unweighted graph construction. However, the main difference between this stage and the others is that edge weights are added to the graph. For example, if a node y is added to node x's adjacency list, the weight $w(x,y)$ is added to the graph and this corresponds to the cost of the path from x to y. Since the LCN is already known from performing exploration (section 3.2), it's possible to determine path distances depending on how far a node is from the goal state (i.e. final name in the LCN). For example, consider the following:

- Let there be nodes x, y, z, g where g is the goal node.
- Let a path from a node a to node b be denoted by $P(a,b)$.
- Let the x and y both be part of the LCN while z can only be chained to x. Thus, the path $P(x,y)$ is part of the path that is the LCN (i.e. overall path $P(x,g)$) while $P(x,z)$ isn't. Consequently, $w(x,y) < w(x,z)$.

When assigning weights to node pairs, nodes that can be chained and are part of the LCN found in exploration (section 3.2) will have smaller edge weights than other pairs since this is the "optimal" path in the sense that it's the path, or chained name, that forms the LCN. The mechanism described thus far corresponds to the weighted graph constructed. However, this stage also employs the use of an unweighted graph in order to compare algorithmic performance in unweighted and weighted graphs. However, since the employed Dijsktra's, A* and IDA* algorithms require edge weights, the unweighted graph used in stage 2 can't be re-used. So, the unweighted graph constructed for this stage is nearly identical to the one used in stage 2 except that it contains edge weights. However, this graph remains unweighted in the sense that all edge weights are of equal cost, which in this project was chosen to be 1. So, both the weighted graph G and "unweighted" graph G' are used to run the algorithms in this stage and the only difference between them is that G' has equal edge weight costs of value 1 across the whole graph while G has "smarter" edge weights that vary depending on the edge's location relative to the goal node.

3.4.2 Description

The only uninformed algorithm used in this stage is Dijkstra's and is described in the appendix, section 7.4. The informed algorithms used in this stage include A* and IDA*, described in the appendix, sections 7.5 and 7.6, respectively. All these algorithms require a graph, start and goal node just as the algorithms used in stage 2. The start node x and goal node y are extracted from the LCN in the same process as stage 2 (section 3.3). Unweighted and weighted graphs are constructed as described in section 3.4.1. IDA* requires an additional parameter, the maximum depth, and this is obtained by calculating the length of the LCN.

3.4.3 Heuristic Function

Informed algorithms like A* and IDA* require a heuristic function to improve their performance. For this project, the heuristic function was obtained as follows:

- Let there be nodes x, y, z, g where g is the goal node.
- Let a path from a node a to node b be denoted by $P(a,b)$.
- Let a weight cost associated with an edge between nodes a and b be denoted by $w(a,b)$.
- Let the distance returned from a heuristic function with nodes a and b be denoted by $h(a,b)$.
- Let x and y, both be part of the LCN. z can be chained to x, but isn't part of the LCN. Thus, the path $P(x,y)$ is part of the path that is the LCN (i.e. overall path $P(x,g)$) while $P(x,z)$ isn't. Consequently, $w(x,y) < w(x,z)$.
- Since x and y are both part of the LCN, $h(x,y) = \text{cost}(P(\textit{startnode},g)) + 1$. In other words, the heuristic function will return the sum of the cost of the path traversed from the start node until x and 1 since y is the next name in LCN after x.
- On the other hand, since z isn't part of the LCN, $h(x,z) = \text{cost}(P(\textit{startnode},g)) + w(x,z) + 1$. In other words, the heuristic function will return the sum of the cost of path traversed from the start node until x, the weight of the edge (x,z) and 1.

4 Experiment Design and Results

4.1 Experiment Setup and Goal

There are a total of 6 algorithms used for this project. The 4 uninformed algorithms are BFS, DFS, IDS and Dijkstra's. The 2 informed algorithms are A* and IDA*. Each time an algorithm is run, its run-time is recorded using the Python time() function. Moreover, print statements are added to the algorithms shown in the appendix, section 7 in order to also record how many "searches" are made. A search corresponds to a node (i.e. name) that's extracted from the graph and examined or compared. This is done in order to gauge how many wrong paths, or names, and unnecessary searches are made by the algorithms to compare their accuracy. So, the experiment is designed to test and compare the efficiency and accuracy of all the algorithms. The run-times and searches for a given algorithm are obtained by averaging the results across three performed trials.

In stage 1, only the average run-times of BFS and DFS are recorded in order to compare their efficiency in calculating all valid paths to find the longest one, or the longest chained name (LCN). Moreover, the algorithms are performed on both the small and large data sets.

In the rest of the stages involving unweighted and weighted graphs, all 6 algorithms are performed only on the large data set because it contains much more data (hundreds of records more) since it includes all past and current players. Both the average run-time and number of searches are recorded across three trials. One important note is that since all algorithms require a start and goal node, the start node is always chosen by extracting the first name of the LCN found in stage one, which was found to be "ronnie lester". However, invalid and valid names are chosen as goal states. The valid state is "dillard crocker", which is the last name extracted from the LCN calculated in stage 1. The invalid state is "lebron james", which a name of a player that's not in the LCN found in stage 1. So, the algorithms used in stages 2 and 3 contain a constructed graph, valid start state (i.e. "ronnie lester"), and either an invalid goal state (i.e. "lebron james") or valid goal state ("dillard crocker") in order to gauge their performance in different environments, namely

unweighted and weighted graphs, and in finding a valid LCN or exhausting all possibilities before knowing that a valid LCN can't be found given the invalid goal state.

4.2 Results

4.2.1 Stage One Results

Table 1: Overall Results for Stage 1

Algorithm	Data Set	Average Run-time
BFS	Small	0.0455 seconds
DFS	Small	5.786 seconds
BFS	Large	39.16 seconds
DFS	Large	2 hours & 6 minutes

Both BFS and DFS found the same LCNs for the small and large data set. The following summarizes these results:

- Small data set LCN: ['chris paul', 'paul george', 'george hill']
- Large data set LCN: ['ronnie lester', 'lester conner', 'conner henry','henry james', 'james thomas', 'thomas jordan', 'jordan mickey', 'mickey dillard', 'dillard crocker']

4.2.2 Stage Two Results

Table 2: Overall Results for Stage 2 Involving an Unweighted Graph

Algorithm	Goal State	Average Run-time (seconds)	Searches
BFS	"dillard crocker"	0.00102	145
DFS	"dillard crocker"	0.00512	77
IDS	"dillard crocker"	0.00276	512
BFS	"lebron james"	0.00150	145
DFS	"lebron james"	0.00468	145
IDS	"lebron james"	0.00296	579

4.2.3 Stage Three Results

Table 3: Overall Results for Stage 3 Involving an Unweighted Graph

Algorithm	Goal State	Average Run-time (seconds)	Searches
Dijsktra's	"dillard crocker"	0.00133	27
A*	"dillard crocker"	3.13e-04	18
IDA*	"dillard crocker"	0.00136	60
Dijsktra's	"lebron james"	0.00128	105
A*	"lebron james"	0.00156	105
IDA*	"lebron james"	0.00125	60

Table 4: Overall Results for Stage 3 Involving the Weighted Graph

Algorithm	Goal State	Average Run-time (seconds)	Searches
Dijsktra's	"dillard crocker"	2.653e-03	12
A*	"dillard crocker"	1.423e-04	9
IDA*	"dillard crocker"	0.0013	45
Dijsktra's	"lebron james"	0.00132	105
A*	"lebron james"	0.00147	105
IDA*	"lebron james"	0.00165	45

5 Analysis

5.1 Stage One Analysis

Looking at table 1 in section 4.2.1, it can be seen that DFS was significantly more efficient in exploring and finding all possible paths to find the longest chained name (LCN). The main reason for this is graph construction. The performance of DFS and BFS can heavily depend on the structure of the search tree, or graph, and the number of possible solutions. The reason for why BFS performed significantly better than DFS in this stage is that goal states (i.e. final name in the chained name) are very far from the root, or starting, node. If solutions are very deep and or rare, DFS can meander around forever. For example, say that DFS is performed on the start node x and goal node g. If x has several elements in its adjacency list l, but g is the last element in l and doesn't appear anywhere else in the graph search. DFS may take a long time searching all elements in l before reaching g because these element may be "deep" in the sense that they have several elements in their own adjacency list. So, DFS will explore nodes until maximum depth and this may take a considerable amount of time. BFS, on the other hand, explores the graph in levels, so all the elements in l will be explored before delving into their own adjacency lists. As a result, g will be found significantly quicker when exploring the first level of l.

5.2 Stage Two Analysis

5.2.1 Complexity

Similar to stage 1, table 2 in section 4.2.2 shows how DFS suffers the worst run-time, regardless of whether or not there's a valid goal state. This is due to the fact that BFS and IDS don't suffer DFS' problem of meandering in deep branches that don't contain the goal state. Although IDS performs a depth-first searches, it's cut off at certain depths if the goal isn't found in order to avoid extensively searching non-goal paths. In terms of run-time, IDS proves to be an improvement over DFS and doesn't differ much from BFS.

5.2.2 Accuracy

Table 2 also shows how there are always as many or more searches done when there's an invalid goal state as when there's a valid one. This is due to the fact that more of the graph is searched when a goal can't be found to form the LCN. More significantly, unlike run-time, DFS has the best performance in terms of the amount of searches it makes. Say that we have a start node x, goal node g, and adjacency list l of x. DFS will performs less searches than BFS because it doesn't have

to search an entire level of l before moving on to the next step. For example, since the goal state is at level 9 (looking at the LCN found in section 4.2.1), DFS would perform less searches than BFS if the LCN is found through the first element of l since it'll only have to search the branch concerned with the first element of l instead of having to search the entire first level of l and then continue this process until the ninth level as it would be done in BFS. IDS requires significantly more searches than BFS and IDS because, since the goal state is at level 9, it'll have to perform DFS starting all the way from the root note 9 times. Consequently, it performs many redundant searches since it'll continuously restart the search until the ninth level is reached where the goal state is reached.

5.3 Stage Three Analysis

5.3.1 Complexity

Tables 3 and 4 in section 4.2.3 show that A* has the best run-time, regardless of the type of graph. With an unweighted graph, Dijsktra's and IDA* have nearly identical run-times. The reason that A* is the most efficient of all algorithms performed in this experiment is its use of a heuristic function. So, even when a graph is unweighted with identical edge costs between all connected nodes, A*'s "good" heuristic function provides it with more informed guidance as to which path should be taken to reach the goal. Dijkstra's algorithm is the one that benefits the most from a weighted graph in terms of run-time. This is due to the fact that in a weighted graph, Dijsktra's can choose paths of smaller costs that lead to the goal. However, in an unweighted graph, it isn't better than A* or significantly more efficient than other uninformed algorithms like BFS and DFS because it doesn't have a heuristic function like A*'s to make informed decisions on which path to choose. IDA* doesn't see such efficiency disparities between unweighted and graphs as Dijsktra's does. However, it's usually always the least efficient out of three algorithms, albeit by a small margin, since it has to restart searches when exploring various levels of the graph. Overall, the most efficient version of an algorithm in the experiment is A* when run on a weighted graph since it makes use of its heuristic function and weighted edges to perform informed pathfinding to avoid wrong paths as much possible when searching for the goal state.

5.3.2 Accuracy

Similar to stage 2, the number of searches performed by the algorithms increases or remains the same whenever run with an invalid goal state because the graph has to be searched more extensively to exhaust all paths and learn that the LCN cannot be found.

As was the case with the run-time, A* is the most the most efficient in the amount of searches when there's a valid goal state because it performs the least amount of searches. Moreover, it performs less searches in a weighted graph than an unweighted graph. Although its heuristic function allows it to perform the least amount of searches in an unweighted graph, A* is completely optimized in a weighted graph with a valid goal state where it only performs 9 search total (Table 4) and these searches corresponds to 9 elements in the LCN. So, it doesn't waste any searches since it does zero unnecessary searches. The combination of a heuristic function and weighted edges allows A* to perform only the minimum amount of searches required to form the LCN.

Once again, Dijkstra's algorithm benefits the most from a weighted graph in terms of searches performed. In an unweighted graph with a valid goal state, it performs 27 searches (i.e. 18 unnecessary searches); however it only perform 15 searches, or 5 unnecessary searches, in a weighted

graph with a valid goal state. Dijsktra's doesn't make use of a heuristic function so a weighted graph allows the algorithm to make more informed decisions on the optimal path to choose and optimize its performance.

IDA*, as expected, performs the most amount of searches when there's a valid goal state because it performs many redundant search since it's required to restart searches from scratch when searching new levels of the graph. However, interestingly enough, it performs the least amount of searches when there's an invalid goal state. This can be attributed to the fact that since there isn't a valid LCN between the start and a goal state, it avoids A*'s and Dijkstra's problem of meandering at deep levels, or paths, that don't contain the goal. As a result, IDA* is much quicker in realizing that there's no solution since an invalid goal state was provided.

6 Conclusion and Future Work

Upon analyzing all the generated results, it can be concluded that BFS performs significantly better than DFS in exploring the graph and finding all relevant paths to solve the NBA "name chaining problem" (NCP) without knowledge of the correct starting and goal nodes. However, given correct starting and goal states, A* performs the best in finding the longest chained name (LCN) because of its use of a heuristic function. Assuming a correct starting state, it's the most optimized in a weighted graph with a valid goal state where it performs no unnecessary searches to find the LCN. Uninformed algorithms like BFS, DFS and IDS perform the worst in this case since they operate on unweighted graphs. Dijkstra's and IDA* both perform better than the three aforementioned algorithms and are the most optimized when performed on weighted graphs. Dijsktra's algorithm, in particular, significantly improves performance with a weighted graph since it doesn't contain a heuristic function, but unlike the other uninformed algorithms in this project, can make use of edge costs to make informed pathfinding decisions. So, given a correct goal state, an informed algorithm like A* would be the most efficient and useful in solving the NCP problem. IDA*, on the other hand, proved to be the most efficient in realizing that there isn't a valid chained name when an invalid goal state is provided. So, when one is uncertain about the data and if valid chained names exist between the names of certain players, IDA* would be as useful, if not more useful, than A* in solving the NCP.

This project can be expanded upon by expanding the exploration portion in stage 1 to include A* and IDA*. Knowing how well these algorithms performed given a valid starting state and invalid/valid goal states, it would be interesting to repeat the process performed in stage 1 for these algorithms to determine how they fare against BFS and DFS in finding all possible paths to find the LCN. Furthermore, comparing more algorithms such as the ones described in section 2 (e.g. dynamic repairing A*, best-first search, DFS*, etc) can allow one to gauge which algorithms are best equipped to solve this problem in the most efficient manner. The experiment can also be improved in how the LCN path is found. Instead of dividing this process into two steps, namely using BFS and DFS to find all paths and then extracting the longest path through looping mechanisms, suggestions can be taken from work done on the longest path problem (section 2.2) to construct the graph in a certain manner and employ a more complex algorithm like modified Dijsktra's to combine the two steps into one algorithm. Moreover, algorithms specialized in finding multiple goals such multi-goal BIDS (section 2.3.1) can also be utilized to find multiple chained names more efficiently in one search. This problem can also be expanded beyond the current environment involving NBA players. It can be applied to other sports or other areas with more intricate data where a similar

pattern between the data exists that can be exploited to gain valuable insight about the data or to employ different algorithms and AI principles to learn about their application and relevance in different scenarios.

References

[1] X. Cui and H. Shi. A*-based pathfinding in modern computer games. *International Journal of Computer Science and Network Security*, 11(1):125–130, 2011.

[2] F. Gouidis, T. Patkos, G. Flouris, and D. Plexousakis. Dynamic repairing a*: a plan-repairing algorithm for dynamic domains. pages 363–370, 01 2018.

[3] R. Graham, H. McCabe, and S. Sheridan. Pathfinding in computer games. *The ITB Journal*, 4(2):6, 2003.

[4] W. W. Hardgrave and G. L. Nemhauser. On the relation between the traveling-salesman and the longest-path problems. *Operations Research*, 10(5):647–657, 1962.

[5] D. Karger, R. Motwani, and G. D. Ramkumar. On approximating the longest path in a graph. *Algorithmica*, 18(1):82–98, 1997.

[6] R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial intelligence*, 27(1):97–109, 1985.

[7] R. E. Korf, M. Reid, and S. Edelkamp. Time complexity of iterative-deepening-a. *Artificial Intelligence*, 129(1-2):199–218, 2001.

[8] K. L. Lim, K. P. Seng, L. S. Yeong, L.-M. Ang, and S. I. Chng. Uninformed pathfinding: A new approach. *Expert Systems with Applications*, 42(5):2722–2730, 2015.

[9] S. B. K. A. D. Patterson. Direction-optimizing breadth-first search. *SC12, November*, pages 10–16, 2012.

[10] R. Uehara and Y. Uno. Efficient algorithms for the longest path problem. In *International Symposium on Algorithms and Computation*, pages 871–883. Springer, 2004.

[11] N. R. Vempaty, V. Kumar, and R. E. Korf. Depth-first versus best-first search. In *AAAI*, pages 434–440, 1991.

7 Appendix

7.1 Breadth-First Search

```
def BFS(graph, start, end):
    temp_path = [start]
    q = []
    q.append(temp_path)
    while q:
        tmp_path, q = pop(q)
        last_node = tmp_path[-1]
        if last_node == end:
            paths.append(tmp_path)
        for link_node in graph[last_node]:
            if link_node not in tmp_path:
                new_path = tmp_path + [link_node]
                q.append(new_path)
```

Figure 1: BFS algorithm Python code.

7.2 Depth-First Search

```
def DFSUtil(graph, src, dst, visited, path):
    visited[src] = True
    path.append(src)
    if src == dst:
        paths.append(path)
    else:
        for i in graph[src]:
            if visited[i] == False:
                DFSUtil(graph, i, dst, visited, path)
    path.pop()
    visited[src]= False


def DFS(graph, src, dst):
    visited = {}
    for player in players:
        visited[getName(player)] = False
    path = []
    DFSUtil(graph, src, dst, visited, path)
```

Figure 2: DFS algorithm Python code.

7.3 Iterative Deepening Depth-First Search

```python
def IDSUtil(graph, src, target, maxDepth, path):
    if src == target :
        return True
    if maxDepth <= 0 :
        return False
    for i in graph[src]:
        if IDSUtil(graph, i, target, maxDepth-1, path + [i]):
            return True
    return False

def IDS(graph, src, target, maxDepth):
    path = [src]
    for i in range(maxDepth):
        if IDSUtil(graph, src, target, i, path):
            return True
    return False
```

Figure 3: IDS algorithm Python code.

7.4 Dijkstra's

```python
def dijkstra(graph, start, target):
    frontier = PriorityQueue()
    frontier.put(start, 0)
    ans = []
    came_from = {}
    cost_so_far = {}
    came_from[start] = None
    cost_so_far[start] = 0

    while not frontier.empty():
        current = frontier.get()
        ans += [current]
        if current == target:
            break
        for next in graph.neighbors(current):
            new_cost = cost_so_far[current] + graph.cost(current, next)
            if next not in cost_so_far or new_cost < cost_so_far[next]:
                cost_so_far[next] = new_cost
                frontier.put(next, new_cost)
                came_from[next] = current
```

Figure 4: Dijkstra's algorithm Python code.

7.5 A*

```python
def a_star_search(graph, start, goal):
    frontier = PriorityQueue()
    frontier.put(start, 0)
    came_from = {}
    cost_so_far = {}
    came_from[start] = None
    cost_so_far[start] = 0

    while not frontier.empty():
        current = frontier.get()

        if current == goal:
            break

        for next in graph.neighbors(current):
            new_cost = cost_so_far[current] + graph.cost(current, next)
            if next not in cost_so_far or new_cost < cost_so_far[next]:
                cost_so_far[next] = new_cost
                priority = new_cost + heuristic(goal, next)
                frontier.put(next, priority)
                came_from[next] = current
    return came_from, cost_so_far
```

Figure 5: A* algorithm Python code.

7.6 Iterative Deepening A*

```python
def IDAUtil(graph, start, goal, maxDepth):
    frontier = PriorityQueue()
    frontier.put(start, 0)
    came_from = {}
    cost_so_far = {}
    came_from[start] = None
    cost_so_far[start] = 0

    while not frontier.empty():
        current = frontier.get()
        if current == goal:
            return True
        if maxDepth <= 0 :
            return False
        for next in graph.neighbors(current):
            new_cost = cost_so_far[current] + graph.cost(current, next)
            if next not in cost_so_far or new_cost < cost_so_far[next]:
                cost_so_far[next] = new_cost
                priority = new_cost + heuristic(goal, next)
                frontier.put(next, priority)
                came_from[next] = current
        maxDepth = maxDepth - 1

def IDA(graph, src, target, maxDepth):
    path = [src]
    for i in range(maxDepth):
        if IDAUtil(graph, src, target, i):
            return True
    return False
```

Figure 6: IDA* algorithm Python code.