March 2018

Graph-Based Sports Rankings

Daniel Alfred
Worcester Polytechnic Institute

Matthew Beader
Worcester Polytechnic Institute

Matthew Jackman
Worcester Polytechnic Institute

Ryan Patrick Walsh
Worcester Polytechnic Institute

A Major Qualifying Project Report

ON

Graph-Based Sports Rankings

Submitted to the Faculty of

Worcester Polytechnic Institute

In Partial Fulfillment of the Requirement for the Degree of Bachelor of Science

By

Daniel Alfred
Matthew Beader
Matthew Jackman
Ryan Walsh

Under the Guidance of
Professor Gábor N. Sárközy
Professor Craig E. Wills

March 23rd, 2018

MQP-CEW-1801

Abstract

Sports rankings are a widely debated topic among sports fanatics and analysts. Many techniques for systematically generating sports rankings have been explored, ranging from simple win-loss systems to various algorithms. In this report, we discuss the application of graph theory to sports rankings. Using this approach, we were able to outperform existing sports rankings with our new four algorithms. We also reverse-engineered existing rankings to understand the factors that influence them.

Acknowledgements

We would first like to thank Worcester Polytechnic Institute for providing us the opportunity to gain hands-on experience through its project-based curriculum. We would also like to extend our thanks to our advisors Professor Gábor Sárközy and Professor Craig Wills for their invaluable guidance, support, and feedback throughout our time working on this project. Lastly, we would like to thank all of our friends and family who have supported us and our education over the past four years.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

Sports and sports rankings have attracted the interest and attention of people across the world for decades. For years, people have tuned into sporting events broadcast over radio or television, or purchased tickets to see them live. In North America alone, the sports industry was worth $60.5B in 2014, and is expected to continue growing [11]. Recent technological developments have allowed the spread of sports discussion from talk radio and television to hobbyist websites. Furthermore, the introduction of online fantasy leagues where users can assemble teams of their favorite players and track their progress have become common with sports enthusiasts and their friends and coworkers. In 2015, American Express expected nearly 75 million Americans to participate in fantasy football, spending almost $4.6B [3].

In addition to the developing following in sports, particular interest has circulated around sports rankings. To best understand sports rankings, we need to understand what a ranking represents in general. For our purposes, a ranking can be defined as a unique ordering of some collection of entities, with an implied hierarchy of which entities are better than others based upon a given comparison metric. In the context of sports, a ranking is an ordering of teams or players based on the metric of their current performance against other teams or players in recent games. Sports rankings can come from many different sources, from analysts at news stations and sports networks to hobbyists. General comparison and performance metrics applicable across many sports include win-loss ratio and points scored within games. The differences in how each person or system weigh these factors drive much of the speculation that goes into sports ranking analysis and impact the finalized ranking each person or system produces.

For this project, we explore sports rankings from a different perspective. Our goal is to apply graph theory to sports rankings. In computer science, a graph is a data structure which can represent information in the form of nodes and edges, where nodes signify different entities, and edges indicate links or associations between nodes. Specifically, we aim to acquire and model sports data for exploration in graph theory to better understand how sports rankings are formulated. For our purposes, teams are represented as nodes in a graph, while matchups and games between these teams are represented as the directed edges between them. When given a ranking, forward edges within the graph are those that agree with the ordering in the ranking, and backward edges within the graph are those that disagree with the ordering in the ranking. The crux of our approach is based upon the Minimum Feedback Arc Set problem [12], [30], a well-known problem in theoretical computer science. Due to the computational complexity of the Minimum Feedback Arc Set problem, determining the optimal placement of teams in sports rankings using graphs is not computationally feasible in a reasonable timeframe without approximations.

Over the course of this project, we provide several contributions to rank generation in the context of sports rankings. Specifically, we:

1. apply graph theory and the Minimum Feedback Arc Set problem to sports rankings with a unique approach different from the more common formulaic ranking approach,
2. implement rank generation through a brute force approach and three approximation algorithms for the Minimum Feedback Arc Set problem,
3. reverse-engineer sports rankings published online utilizing our graph data structures, and
4. outperform existing sports rankings using our own algorithms according to our evaluation metric.

In this project, we successfully outperformed each existing and external ranking we tested according to our metrics in 2016-17 National Football League (NFL), 2016-17 NCAA Division I Football Bowl Subdivision (CFB), 2014-15 National Hockey League (NHL), and 2015 Major League Baseball (MLB) seasons using our algorithms and graph-based approach. Our algorithms are successful in these test cases, where most of them consistently outperformed all of our external rankings. For example, for 2016-17 NFL ranking results, existing rankings from ESPN, NFL.com, and Sports Illustrated produce out-of-order rankings with roughly 16% discrepancy between rankings and game results, while the best of our algorithms produce rankings with only 13% discrepancy. Additionally, we apply this metric and the metric of rank differential to reverse-engineer external rankings and determine how important they consider certain factors when developing rankings.

In this report, we detail the background knowledge necessary for this project as well as the approaches, applications, and tests we performed to develop our contributions. Chapter 2 introduces the relevant background information to our report, including graph theory, the Minimum Feedback Arc Set problem, a synopsis of our algorithms, traditional ranking approaches, and factors for consideration in ranking algorithms. Chapter 3 discusses our design considerations for the programs and scripts we developed to support these algorithms, as well as methodologies explored with respect to sports ranking factors. We then transition from our design of these features to implementation in Chapter 4, noting any modifications encountered during implementation of the design. Chapter 5 reviews test cases we conducted on our base implementation alongside analysis of the results. Chapter 6 describes our approach to handling edge weights within the graph, Chapter 7 explores different methods of generating and handling reduced-size rankings, and Chapter 8 introduces a new post-processor and ranking algorithm we developed. Our full sports test cases are analyzed and discussed in Chapter 9, and our project conclusion and suggested future work are included in Chapter 10. Appendix A contains a glossary for the terminology used in this report and Appendix B displays additional heatmap evaluation of existing rankings.

Chapter 2: Background

In this chapter, we present the relevant information necessary to understand this project and our developments. We begin with a brief introduction of graph theory and the definitions and concepts used throughout this report. We introduce the Minimum Feedback Arc Set problem, the foundation for this project, and its application towards sports rankings. Different metrics for evaluating sports rankings when considering the Minimum Feedback Arc Set problem are then explored. Afterwards, three of our four ranking algorithms are introduced alongside their origins and applications. We then summarize more traditional, formulaic methods of generating rankings. Finally, we close the chapter with discussion about potential factors considered within these ranking formulas.

2.1: Graph Theory

In this section, we introduce graph theory with the concepts and definitions described in [22]. A **graph** is G = (V, E) consisting of V, a nonempty set of **vertices** (or **nodes**) and E, a set of edges. Each **edge** has either one or two vertices associated with it, called its **endpoints**. An edge is said to connect two endpoints. For example, a graph structure can be used to model the location of cities within a region. Each city is represented as a node, which can contain information such as the city's name and its population, while each edge represents the existence of a link between the two nodes or cities it connects. With this information, search algorithms can be applied on the graph to determine if a path exists between two given cities.

One important property of graphs relating to this project is whether a graph is unweighted or weighted. An **unweighted graph** does not assign any values to its edges, and uses its edges just to represent association and connection. A **weighted graph** contains edges that each have a numerical value as an "edge weight." An edge weight has many uses; for instance, edge weights can be applied to the city example above to denote the distances between cities, allowing for algorithms to determine which paths should be taken when traveling between cities if several exist.

Another important property of graphs is whether they are directed or undirected. A **directed graph** is defined as G = (V, E) with a nonempty set of vertices V and a set of directed edges E. Each **directed edge**, or **arc**, is associated with an ordered pair of vertices. The directed edge associated with the ordered pair (u, v) is said to start at u and end at v. The **indegree** of a vertex v is the number of edges with v as their ending vertex, and the **outdegree** of a vertex v is the number of edges with v as their starting vertex. Figure 1 is an example of a directed graph, where each directed edge contains an arrowhead pointing towards the ending node from the starting node. Following Figure 1, we can use the directed edge from node A to get to node C, but we cannot use the edge pointing to node E to get to node C because the relation is not bi-directional. Conversely, an undirected graph is bi-directional, where every edge implies that the source and destination nodes are reachable from each other using only that edge.

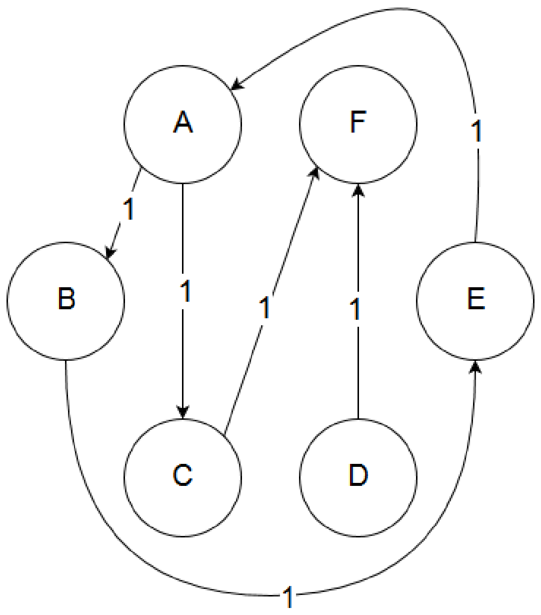


Figure 1: A sample directed cyclic graph

Additionally, graphs can be classified on whether or not they contain cycles. To understand cycles, we must first define a path. A **path** from vertices A to B for a directed graph is a sequence of edges $(x_0, x_1), (x_1,x_2), (x_2,x_3), \ldots , (x_{n-1}, x_n)$ in G, where n is a nonnegative integer, and $x_0 = A$ and $x_n = B$, that is, a sequence of edges where the terminal vertex of an edge is the same as the initial vertex in the next edge in the path. A path $x_0, x_1, x_2, \ldots , x_n$ is defined to have a length n. A **cycle** is a path of length $n \geq 1$ that begins and ends at the same vertex.

The concept of cycles within graphs can be used to define cyclic and acyclic graphs. A **cyclic graph** contains one or more cycles, while an **acyclic graph** does not contain any cycles. Figure 2 depicts a graph with a cycle containing nodes A, B, and C, as there is a path for each node in the graph which begins and ends at the same node. However, if we were to remove the directed edge from node C to A in Figure 2, the graph would be acyclic as shown in Figure 3.

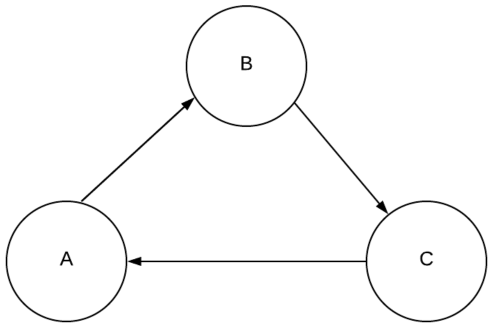


Figure 2: A basic cyclic graph

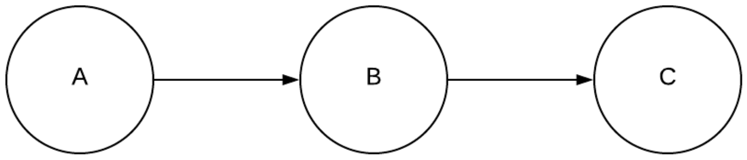


Figure 3: A basic acyclic graph

Using the collection of edges within a directed graph, a ranking of its vertices can be generated. A **ranking** of a directed graph can be defined as a unique ordering of some collection of entities, with an implied hierarchy of which entities are better than others based upon a given comparison metric. If given a ranking for a directed cyclic graph, the distinction must be made between forward edges and backward edges. A **forward edge** is an edge that agrees with the ranking; its existence indicates that the ordering of one node above another is correct. Conversely, a **backward edge**, or **backedge**, is an edge that disagrees with the ranking; its existence indicates that the ordering of two nodes may be incorrect.

Following Figure 2, two example rankings can be generated: A>B>C and A>C>B. Examining the first ranking, A>B>C, shows that two forward edges occur on the graph: (A, B) and (B, C), because these edges agree with the ranking provided. However, the edge (C, A) is a backedge, because its existence states that node C should be ranked higher than node A, but it is not. The second ranking, A>C>B, has only one forward edge on the graph, (A, B). This ranking has two backedges (C, A) and (B, C). Both of these rankings are valid, but consideration must be given to the number of backedges for a ranking when applied to a graph to determine how accurate the ranking is for that graph.

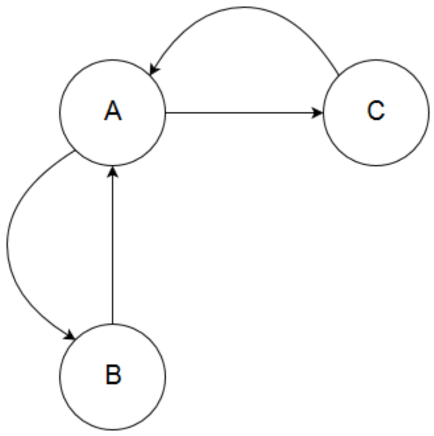


Figure 4: A strongly connected component with several cycles

The final property of graphs relevant for this report is connectedness. A directed graph is a **strongly connected** if there is a path from A to B and from B to A whenever A and B are vertices in the graph. Specifically, for a directed graph to be strongly connected, there must be a sequence of directed edges from any vertex in the graph to any other vertex. Following this concept, **strongly**

connected components are subgraphs of a directed graph G that are strongly connected but not contained in larger strongly connected subgraphs, that is, the maximal strongly connected subgraphs. In sum, a strongly connected component requires that any node in the component be reachable from any other node in the component by a path between them. Following Figure 4 above, nodes A, B, and C are all reachable from each other, and they are the maximal strongly connected subgraph, thus they form a strongly connected component. A strongly connected component contains several cycles, also demonstrated in Figure 4. Two cycles exist in this figure: a cycle containing nodes A and B, and a cycle containing nodes A and C.

2.1.1: Sports Data Representation With a Graph

When representing sports data with a graph structure, we assigned each sports team, or player if analyzing a single-person sport, a node, and each game played an edge between the two nodes to signify the relation between them. These edges were directed, where the winning team was the source node of the edge and the losing team was the destination node. However, this presented a problem if a tie occurred and no clear winner was determined. To address this, a tie was represented with two directed edges pointing between both teams. Although this resulted in more edges within the graph than total games played, it allowed the graph to maintain information about the tie, and signified that, based on that tie alone, neither team could be determined as better than the other.

2.2: The Minimum Feedback Arc Set Problem

The core of this project relates to the Minimum Feedback Arc Set problem. This problem is based upon one of Richard Karp's 21 original NP-Complete problems [12], the Feedback Arc Set problem. The goal of the Feedback Arc Set problem is to find a set of edges within an input directed graph that, when removed, results in a directed acyclic graph [30]. Further, a feedback arc set is minimum if there exist no smaller feedback arc sets for a given graph [30]. By removing the fewest edges possible, the Minimum Feedback Arc Set removes the cyclic property of the graph while maintaining more information than other feedback arc sets, as any unnecessarily removed edges result in information loss of the graph. The Feedback Arc Set problem has applications in varying fields of computer science, ranging from operating systems and process scheduling to database systems [7].

In order to understand NP-Completeness and how it relates to the Minimum Feedback Arc Set, we must define the P versus NP Problem in theoretical computer science. The class P, which stands for "polynomial time," describes a set of problems where solutions can be generated in a time value related to a polynomial of the size of the input to the problem. The class NP, which stands for "non-deterministic polynomial time," describes a set of problems where solutions can be verified efficiently in polynomial time [8]. If $P = NP$, we could demonstrate that every problem where a solution can be verified efficiently can also have a solution generated efficiently. However, most computer scientists believe that $P \neq NP$ and that no polynomial time algorithms exist to solve NP-Complete problems. Thus, the NP-Complete nature of the Minimum Feedback Arc Set problem means that we cannot efficiently generate a solution, so we must utilize approximations instead.

The descriptions of the Feedback Arc Set and Minimum Feedback Arc Set problems above apply in the cases where the input graph does not utilize edge weights. However, in the case of input graphs which contain edge weights, the approach must be adapted. Unweighted graphs can be considered as weighted in this adaptation, where each edge has a weight of 1. In this context, when searching for the minimum feedback arc set, we are searching for the set of edges with the smallest total edge weight that remove all cycles from the graph, effectively maintaining the greatest amount of edge weight within the graph rather than maintaining the greatest number of edges.

2.3: Graph Correctness Metrics

For the purpose of evaluating sports rankings in this project, we needed metrics to determine how accurate these rankings were. Two options are presented below, both of which were applied in this project. The first option is total backedge weight, which is centered in the weighted Minimum Feedback Arc Set problem; the second option is rank differential, a more comparative approach.

2.3.1: Total Backedge Weight

The first metric we investigate for evaluating sports rankings is total backedge weight. This metric involves applying a given sports ranking to a graph of sports data, using the node and edge format described in Section 2.1.1, and summing the weights of backedges for that ranking to get the **total backedge weight**. The weighted Minimum Feedback Arc Set problem suggests that if one ranking has more total backedge weight than another ranking, then the first ranking is less accurate for this data set. Indeed, rankings which place teams out of order will end up generating more backedges, thus having a higher total backedge weight. Rankings which rearrange the teams to remove backedges with as much total backedge weight as possible will be ignoring games with the smallest possible impact in the sports season and will subsequently be more accurate.

One benefit to the total backedge weight metric is that is an absolute metric. This means that rankings can be evaluated individually with this metric and are not dependent on other rankings for comparison. Thus, all rankings for a given graph can be compared against each other with no modification. The downside to this approach is ambiguity with the total backedge weight. Unless the total weight of the graph is presented, it cannot be determined whether the total backedge weight for a ranking is large or small. Otherwise, an additional ranking can be evaluated simultaneously to determine which ranking results in a better total backedge weight, but this approach removes the absoluteness of the metric.

2.3.2: Rank Differential

The second metric for evaluating sports rankings we explore is rank differential. **Rank differential**, also referred to as rank comparison, is a comparative metric, where its basis depends on the correctness of a **control ranking**. This metric is primarily used to establish relative correctness between two rankings for the same graph. To calculate the rank differential, the differences in rank number of teams between the control ranking and a **test ranking** are computed and averaged over all the teams. The resulting value indicates that, on average, the test ranking can be expected to place teams the given number of positions out of order when compared to the control ranking. Rank differential has foundations within the total backedge weight metric as well: if two rankings have a rank differential of 0, that means that both rankings are identical, so they must have the same backedges and subsequently the same total backedge weight.

An advantage to this approach for analyzing sports rankings is its simplicity. At a glance, it is easier to understand and present rank differential compared to total backedge weight. Additionally, no information about the graph or the specific ordering of teams in each ranking is necessary to convey the rank differential metric. The main disadvantage of this approach is its relative scoring and reliance on a control ranking. If looking to generate the best ranking for a graph, the rank differential metric is dependent on having the best ranking as the control to compare against. However, this approach is sufficient if the goal is to better understand what factors influence the control ranking by testing which approaches bring the test ranking closer to the control.

2.4: Minimum Feedback Arc Set Solution Algorithms

When considering sports rankings from a graph-based approach, we needed to implement algorithms to determine or approximate the Minimum Feedback Arc Set to find which edges from each sports data set could be removed to generate the best ranking. As mentioned in the Introduction, we implemented four different algorithms for use in our sports data sets: a brute force approach, the Berger/Shor New 2-Approximation algorithm, a Hill Climbing algorithm, and a variation of Hill Climb using the iterative metric of variance in rankings.

2.4.1: Brute Force

Brute force is defined in computer science as the approach of trying all possible solutions for a problem until the best one is found [5]. Because brute force checks all possible solutions to a problem, it is guaranteed to find the best solution. Brute force approaches have practical applications in computer science, such as generating solutions to string matching and comparison [5]. However, the major drawback to brute force is that, because all possible solutions are checked, completion of the algorithm often does not occur within reasonable time. In the string comparison application, if brute force is searching for the occurence of one set of non-contiguous characters in order within a string, the algorithm will take a factorial amount of time to complete, increasing very quickly as the size of the problem increases. Thus, larger problems are often not computationally feasible to solve with a brute force solution, unless the problem space can be reduced.

2.4.2: Berger/Shor New 2-Approximation Algorithm

The Berger/Shor New 2-Approximation algorithm was developed by Bonnie Berger and Peter Shor as an approximation to the Maximum Acyclic Subgraph problem [1]. The Maximum Acyclic Subgraph problem accomplishes the same goal as the Minimum Feedback Arc Set problem, except rather than returning the minimum feedback arc set, it returns the graph without those arcs [1]. Given the NP-Complete nature of these problems, there was a need to develop approximations that could terminate in polynomial time but would yield results close to the actual solution.

This algorithm guarantees that any cycle that exists in the input graph will be broken in the output graph: all nodes in the input graph are evaluated once, and during evaluation of each node, either its incoming or outgoing edges are discarded from the output graph, such that at least one edge of every cycle will be discarded, and the cycle broken [1]. Additionally, because it removes the larger of the two sets of edges for each node, the Berger/Shor algorithm always retains at least half of the edges within the graph. Finally, because each node is only visited once in this approximation, this algorithm completes in polynomial time, which makes approximating solutions to the Minimum Feedback Arc Set computationally feasible.

2.4.3: Hill Climb

Hill climbing is a greedy local search algorithm commonly used in artificial intelligence and is useful for solving optimization problems. A local search algorithm is an algorithm that looks at a single state when searching rather than a tree of paths. Changes in local search algorithms are typically to neighboring states. Greedy algorithms make decisions by choosing the state that is best in any given instance. The hill climbing search algorithm is a looping procedure which continuously moves in the direction of an increasing value of some success heuristic. During each loop, neighboring states of the graph are observed and the best possible neighbor state is chosen. This process continues until there is no longer any improvement in the heuristic value for any neighbor state [23].

Though hill climbing is quick to solve problems, it can get caught in a local maxima or local minima. A local maxima is a point for which the goal heuristic value reaches the maximum for neighboring states. The goal for the hill climbing search is to reach the global maxima, but it is impossible for a hill climbing search to know if a local maxima is also the global maxima. The local maxima problem can be somewhat mitigated through random restarts. Random restart hill climbing requires restarting the hill climbing loop multiple times with random initial states in the hope that one of these states will reach a better local maxima, or even reach the global maxima [23].

The hill climbing search can also struggle with plateaus in the state space. A plateau is a scenario where all of the neighboring states have the same heuristic value. Hill climbing can get stuck on a plateau since the neighboring states are not showing improvement for the hill climb search to choose. Plateaus can be dealt with using sideways moves, which are an allotted amount of occasions where the hill climbing search will settle for equivalent values of the heuristic in the hope that there will be improvement further in the state space [23]. Using these improvements, hill climbing is a suitable approximation approach for optimizing an ordering of nodes.

2.5: Traditional Ranking Algorithms

Many methodologies have been formulated to develop sports rankings without the use of graph theory. Some of the rankings produced by these methodologies are earned rankings while others are predictive. Earned rankings are based on prior feats whereas predictive rankings are ones that most accurately predict the outcome of a future game [24]. Earned rankings can be used to verify a given ranking and provide a basis for predictive rankings. A ranking can be both earned and predictive if, for example, the same ranking that is created after a season ends is then used to predict a ranking for the next season. The ranking is based upon the team's record over the most recent season and therefore earned. Because that ranking is then used to predict the next season's ranking, it is also predictive.

Some methodologies, like win-loss systems, have features that lend themselves well to being an earned ranking while a methodology such as Elo is useful for the purposes of predicting future game outcomes.

2.5.1: Win-Loss Systems

Win-loss systems are useful when producing a ranking as they do not require much data to compute. For each team, only two numbers need to be recorded, which correspond to the team's number of wins and losses. If needed, a third value can also be kept to represent the number of matches that ended in a draw, tie, or were otherwise canceled. This is not always needed, however, as some sports do not allow for a game to end with a tie.

An example of a win-loss system is shown in (1) [24], where the only influence in a ranking is whether or not a team wins. In (1), R represents the outcome between a home team and an away team, S_h represents the score of the home team, and S_a represents the score of the away team. If both teams are playing at a neutral venue, then assigning the variables is arbitrary.

$$R_g^{WL}(S_h, S_a) = \begin{cases} 1 \text{ if } S_h > S_a \\ 0 \text{ if } S_h = S_a \\ -1 \text{ if } S_h < S_a \end{cases} \quad (1)$$

Win-loss systems are useful when only a limited amount of data is available, since the system only considers victories and losses and not other factors in a game like score differential or penalties accrued by one team. However, because these systems only take into account the winner of the game, additional information about the match that could be accounted for is lost, as there is no distinction between a team that won a game by a blowout and a team that won a game by one score. Given that score differentials can vary widely across sports, a win-loss system can be easy to apply when looking at a variety of sports.

2.5.2: Elo

Elo is a particular team's ranking in relation to other teams using a numerical score [28]. The equation used for calculating Elo for world football teams is shown in (2) [29].

$$R_n = R_O + K \bullet (W - W_e) \qquad (2)$$

Updating the Elo for world football teams can be computed using the formula, where R_O represents the original rating of the team before the match and R_n represents the updating rating after the match. K is the weight constant for the tournament being played which varies in value from 60 for World Cup finals to 20 for friendly matches [29]. W is the result of the game where W is 1 if the outcome was a victory, 0.5 if the outcome was a draw, or 0 if the outcome was a loss. W_e represents the expected result of the game and is calculated using the formula
$W_e = 1/(10^{(-dr/400)} + 1)$ where dr is the difference in ratings plus 100 points for a team playing at home [29].

When teams play a match, Elo is gained by the victor and lost by the loser. The amount of Elo gained or lost by a team is dependent upon the Elo of the competitors. If the difference in Elo is large between teams, then the lower Elo team has more to win than it has to lose, while the higher Elo team has more to lose and not as much to win. At the start of a season, teams all begin with the same amount of Elo, and as the season progresses, teams slowly gravitate towards their final skill rating, thereby providing a numerical ranking of those teams. This is especially useful in situations with large numbers of competitors, where trying to compute an ordering would take significant time. One downside to Elo is that one bad game for a highly ranked team can have a large, negative impact on their rating. In the case of an upset, where a lower-ranked player beats a higher ranked player, their Elo is adjusted, allowing for simplistic self-correcting. Because of this feature, low-ranking teams who score an upset victory over a highly-ranked team will have their scores increased accordingly. In the future, this may be used to determine the following weeks' schedules and can add additional pressure to games where there are more points to be won.

The Elo system was adopted by the World Chess Federation in 1970 and in the rising popularity of e-sports, Elo has also become one of the predominant ranking systems for various Leagues [9] [17].

2.6: Factors Within a Ranking Algorithm

With the wide variety of ranking methodologies, different factors are considered when each of those rankings are produced. Two different rankings may be produced from the same dataset depending on what criteria was analyzed. In the following sections, several key factors that ranking methodologies consider are addressed and expanded upon.

2.6.1: Point Differential

Some ranking systems place emphasis on the final score of the game and use the score differential to assign "points" to a team. If two or more teams end with the same win-loss record, a score-differential system can be used to determine which team has the overall highest ranking from their games and thereby produce an ordering of the teams. An example of this is shown in (3) [24].

$$R_g^{SD}(S_h , S_a) = S_h - S_a = \Delta S_{ha} \qquad (3)$$

In the formula shown in (3), a system like this would reward teams for running up the score. If a game is a blowout then the winning team can earn more points from a 50-7 end-game score than a 35-7 end-game score. Since ΔS_{ha} is the difference in score between the home team and the away team, there is no distinction between a 14-7 end-game score and a 7-0 end-game score [24]. Though not always relevant, score differential systems do give a large advantage to the better team in an unequal skill match-up. If an away team were to blow out the home team, then the result would be a very large, negative value which may inflate or deflate a team's overall score, making tracking of individual game outcomes much more important overall.

A system like this, however, is somewhat limited to sports where scoring is a simple comparison between two teams. Sports like chess, for example, would have harder times applying a ranking system such as this due to the lack of "scoring" mechanic.

2.6.2: Recency of Game

A potential factor taken into consideration by traditional ranking algorithms is how recently each game was played. Games played near the end of the season should, theoretically, provide better insight into a team's current performance than games played at the start of the season, as many factors can change a team's performance throughout a season. For the purposes of our project, we focused on the date of the game. When looking at data across several years, we determined that more recent data can be used to identify trends in a team's performance and affect future predictive rankings.

2.6.3: Strength of Schedule

Each game played within a sports season represents a skill matchup between two teams. The skill of the opponents a team will face differs by team and can impact the win-loss ratio of that team over the course of a season. Weekly rankings of teams can be vastly skewed depending on which teams played each other recently. Low-ranking teams who play against high-ranking teams will most likely end the week with losses whereas the highly ranked teams should end the week with victories. In large leagues such as the NCAA Division I Football Bowl Subdivision, there are over 120 teams and not all teams will play against each until the post-season games [15]. Using divisions to divide up the teams makes it less likely that two teams will face each other until after the regular season. Predictive outcomes for these games can vary widely depending on what a team’s schedule looks like throughout the season. If one team consistently plays teams that are ranked lower, it is natural to assume that the first team will be very highly-ranked. However, this may be falsely representing their actual skill level when compared to others in the sports league. With worse opponents, basic win-loss systems begin to show their flaws as which teams are in a match is sometimes just as important as the outcome of the game.

2.7: Summary

In this chapter, we reviewed graph theory and relevant concepts to the Minimum Feedback Arc Set problem for this project. We explored metrics to analyze rankings and introduced the background behind our selected ranking algorithms. We then discussed the theory behind formulaic ranking algorithms and the potential factors they consider.

Chapter 3: Base Design

In this chapter, we introduce decisions made relating to the overarching design of our program, data collection, and data representation within the graph structure. We open by discussing considerations for obtaining and formatting relevant sports data for use with our algorithms and how to apply this data to our graph as edge weights. We then discuss each of our algorithms and their applications to graph theory, their runtime complexities, and our contributions to them. Finally, we plan how to efficiently add testing support to our program for future test cases.

3.1: Sports Data Format

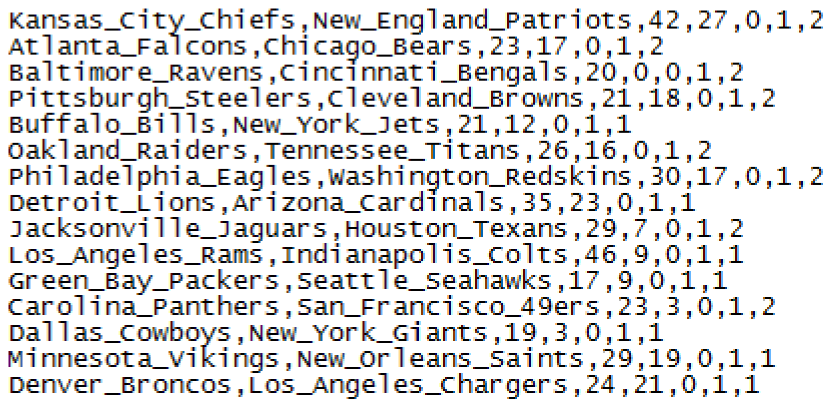
In order to best apply our project and its resources to practical applications, we developed a common format for all sports input data we wanted to collect. Within a given sport, there are many metrics that can be considered when generating a ranking, such as offensive or defensive performance. However, many of these metrics are specific enough that they cannot be applied to more than a few sports. To keep our program as flexible as possible, we needed to isolate factors important in determining the rank that are present in the majority of popular sports. Additionally, we wanted the presentation of our datasets to remain human-readable in case further analysis was necessary in the future. The resulting format we designed required the following, in order:

- The name of the winning team or player
- The name of the losing team or player
- The winner's score
- The loser's score

Additionally, we designed our format to accept the following data, though not require it:

- Whether a tie occurred
- When the game was played
- Where the game was played, whether at one team's home field, or a neutral venue

This format allowed us to capture a substantial amount of information applicable across all sports relevant to this project and apply it to the node weights in our graph. Once the format was agreed upon, sports data could be collected and processed for usage in testing against our rankings. Figure 5 below shows our data format applied to the first week of the 2017-18 NFL season, containing the two team names, their scores, the tie flag, the date, and the location flag.



```
Kansas_City_Chiefs,New_England_Patriots,42,27,0,1,2
Atlanta_Falcons,Chicago_Bears,23,17,0,1,2
Baltimore_Ravens,Cincinnati_Bengals,20,0,0,1,2
Pittsburgh_Steelers,Cleveland_Browns,21,18,0,1,2
Buffalo_Bills,New_York_Jets,21,12,0,1,1
Oakland_Raiders,Tennessee_Titans,26,16,0,1,2
Philadelphia_Eagles,Washington_Redskins,30,17,0,1,2
Detroit_Lions,Arizona_Cardinals,35,23,0,1,1
Jacksonville_Jaguars,Houston_Texans,29,7,0,1,2
Los_Angeles_Rams,Indianapolis_Colts,46,9,0,1,1
Green_Bay_Packers,Seattle_Seahawks,17,9,0,1,1
Carolina_Panthers,San_Francisco_49ers,23,3,0,1,2
Dallas_Cowboys,New_York_Giants,19,3,0,1,1
Minnesota_Vikings,New_Orleans_Saints,29,19,0,1,1
Denver_Broncos,Los_Angeles_Chargers,24,21,0,1,1
```

Figure 5: 2017-18 NFL example data format

In the case of the NFL, where games are measured on a weekly basis, the week number is substituted in place of the date as shown above; otherwise, the date follows a YYYY/MM/DD format. The location flag allows for home-field advantage to be applied when assigning edge weights, where 0 signifies a neutral venue, 1 signifies that the game was played at the winning team's venue, and 2 signifies that the game was played at the losing team's venue.

3.2: Sports Data Collection

In order to acquire various seasons of sports data, we saw the need to design a data collection tool that could pull the information from web sources. One such website, sports-reference.com, offered data for essentially every season of several major sports, all in similar formats. Unfortunately, the data was not offered in a format beyond bare HTML that could be automatically collected or could be directly downloaded for free. Node.js was chosen for our collection tool, as npm contains several virtual DOM implementations and the nature of the language allows the processing of a page to be exactly the same as if processing with JavaScript in a browser. Speed and efficiency of the collection script was not a concern, due to how small the data of an individual season is and how the server we collect from ultimately determines how quickly data can be collected, making Node.js a reasonable choice.

3.3: Basic Program Structure

Development of our rank generation program in this project first began with the implementation of a graph to maintain our sports data and the framework to store, manipulate, and output sample graphs. We initially developed a proof of concept for this program in Python, which read in a graph from a file and used a collection of Node and Edge objects to maintain it. However, after discussion, we migrated this proof of concept program to C++ and adopted an adjacency matrix for our graph's data structure. An **adjacency matrix** A of graph G is defined as "the n x n zero-one matrix with 1 as its (i,j)th entry when v_i and v_j are adjacent, and 0 as its (i,j)th entry when they are not adjacent" [22]. The migration to C++ offered us greater control over memory management, while the adjacency matrix greatly reduced memory usage and computation time.

Using an adjacency matrix improved the ease of node and edge mutation, and simplified the generation of adjacency lists, or lists of neighboring nodes from a given node.

We planned for the rank generation program to have two main parts. The first main part of the program was focused on handling user inputs and storing data that would need to be reused between algorithm runs. This functioned as a wrapper for the second main part of the program, which conducted the execution of the algorithms themselves. The general execution flow of the program is as follows: first, the program parses the input command line arguments and enables or disables the corresponding features as specified. The only required command line argument is a text file containing the sports data to be parsed into a directed graph. Second, the program reads the graph input file and generates an adjacency matrix according to the specified configurations for weighting edges. Next, the program acts as a wrapper and executes one or multiple ranking algorithms on the imported sports data. Finally, the program executes any post-processing or testing on the data before outputting results and exiting.

3.3.1: Data Retention

One of the first obstacles to working with the Minimum Feedback Arc Set problem was importing sports data as a graph that can be read and modified. This data needed to be first translated into nodes and edges, at which time it would be stored until needed by a rank generation algorithm. One of the main motivations for the switch from the initial Python implementation to C++ was for greater performance in the face of the NP-Complete nature of the Minimum Feedback Arc Set problem. The input graph would need to be accessed hundreds of thousands of times in the rank generation process, so performance of graph access was of great concern. Retention of graph data was redesigned into an adjacency matrix, rather than node and edge objects. Inside the adjacency matrix, each cell would hold the weight of the directed edge from the source node represented in the row to the destination node represented in the column.

As edges in the input graph were retained in an adjacency matrix, the information for nodes was retained as integer indices of the associated rows and columns in the adjacency matrix that each node referred to. The indices in the adjacency matrix could then be referenced in an index in a vector which contained the names of the nodes in the input graph. In the context of this project, these node names would refer to all unique team names in the input dataset. The decision to regard nodes as integers during computation greatly simplified the process of handling nodes in the program. From arbitrarily sized names of teams, node information was condensed down into a small, statically sized amount of memory that could be used both to retrieve the node’s name as well as access relevant edge weights in the adjacency matrix. Looping through a list of nodes was as simple as looping through an array or a vector of integers. The maximum number of nodes was also held globally in the program to easily know how many indices to loop through if all nodes needed to be iterated through. The maximum number of nodes was also used to signify which parts of the adjacency matrix were valid. Through referencing nodes as integers, the creation and manipulation of orderings was a simple and consistent process that proved to simplify the more complex task of generating rankings.

3.3.2: Edge Weights

Another important consideration when importing sports data into a graph structure was how to assign weights to each edge. The first implementations of our program used unweighted directed edges, where the weights for edges could only be 1's. Essentially, each edge only signified a win or loss, similar to how sports data would operate in an unweighted graph. Unweighted graphs led to greatly simplified data input and computation as only the winning team name and losing team name were required for input, but this resulted in the loss of crucial information about the game during rank calculation. Thus, a binary wins and losses system for developing our rankings was not sufficient.

Discussion on other relevant information to use in our rankings led us to the conclusion that point differential, strength of schedule, home-field advantage, and recency of game all impacted the importance of a win of one team over another. All of these factors except strength of schedule were represented explicitly within our data format and could easily be computed and applied as needed after being read in from the file. For simple testing in our first few program revisions, we began weighing each edge as the point differential of the game. Point differential appeared to be the most reasonable gauge of the magnitude of a victory and became the main basis of edge weights. In addition to point differential, home-field advantage and recency appeared to be good modifiers of the significance of a game result. We continued experimenting with different edge weight factors in the interest of better modeling sports data within a graph throughout this project, where further development is discussed in Chapter 6.

3.3.3: Edge Weight Customization

As part of the process of importing the sports data into a directed graph, the weights of the directed edges needed to be calculated from each game. This warranted the need for a system to condense sports data into edge weights. We designed a customizable framework for importing edges from columns within our sports data files. Utilizing a system of interactive prompts, the user could tell the program how to handle each of the additional data columns on a column-by-column basis. We designed the customization system in this fashion because some sports have different relevant data that may be unique to the sport. This configuration was also designed to be output to a readable text file generated through the program. The weight configuration file could then be provided as a command line argument and read into the program to skip the interactive prompts and automatically configure the weights as specified. This configuration file could then be shared between testers to replicate the weight configurations that produced a ranking. The weight configuration file also allowed the configuration variables to be edited outside of the program. Multiple configurations held in easily editable external files allowed comparison of results between different weight configurations.

To maintain flexibility for our weight configuration system, we designed several configurations for the columns of data within an input file. The two most basic kinds of configurations we created took the value in the column at face value and multiplied it either by one or by a configurable ratio. Another couple configurations also allowed the user to specify that a column contains the week or date of the match, which is information that could be used in the recency of game process. Columns could also specify which team was the home team. Finally, we allowed columns of information to be ignored altogether in the configuration system. This powerful tool allowed us to filter out irrelevant data from our input files to suit the changing testing requirements as our project developed. All of these configurations could be specified in any order and allowed us great flexibility in handling variable amounts of data that could differ between sports.

After reading each game from the input sports data file, all of the factors of the edge weights were consolidated into a single edge weight to indicate the significance of that edge. In the consolidation process, any post-processing on the edge weights would be applied. Most notably, this stage of the edge reading process would apply any decay or normalization in a process outlined in Chapter 6. Once the edge weight calculation had completed, the edge weight would be applied to the appropriate cell in the adjacency matrix. In the case where two or more edges shared origin and destination teams, we decided to sum the edge weights with the reasoning that an edge becomes even more significant if there are more than one games that support it. However, in order to not lose any information in the process of summing edge weights between the same two nodes, we maintain a separate list of extra indegrees and extra outdegrees. Finally, in the case of a tie, we apply the edge weight in both directions of the edges between the nodes in order to preserve the existence of the matchup.

3.4: Algorithms

Using our program and graph data structure as a platform for implementation, we explored different algorithms to solve or approximate the Minimum Feedback Arc Set problem. When selecting algorithms to implement, we had to be mindful of both the computation time and the accuracy of the output ranking, especially when considering the size of the different sports datasets we wanted to test with. We decided to implement four algorithms: a Brute Force approach, the Berger/Shor New 2-Approximation algorithm, an adaptation of the Hill Climbing approach used in artificial intelligence, and a similar Hill Climbing approach that utilized a comparison metric of our own design.

3.4.1: Brute Force

The first consideration for a ranking algorithm in our program was a brute force approach. Based on the notion of backedge weight discussed in Section 2.3.1, we designed an algorithm using the brute force methodology that iterated through every possible ranking of the nodes in a given graph and returned a ranking with the lowest total backedge weight. This was not a direct solution to the Minimum Feedback Arc Set problem because it returned the ranking for the graph from which the minimum feedback arc set could be determined instead of returning the set itself. The benefit of this algorithm was that we could find the best ranking for the input graph because all possible permutations were considered. The downside to this algorithm was its time complexity: because it evaluated all possible permutations, the computation time was factorial by number of nodes or teams, or O(V!). Therefore, this approach does not scale well to larger graphs and the rankings of such graphs would not be computable in a reasonable timeframe.

Our contribution to brute force was the exploration of optimizations with our algorithm to improve its efficiency. The first optimization we considered was to split the brute force evaluation process over several processor threads. Our brute force algorithm generated permutations in order to be evaluated, which allowed for this task to be divided among several threads. The second optimization we considered was to reduce the number of permutations evaluated by considering strongly connected components within the graph, as the presence of strongly connected components resulted in permutations that were not valid. For example, if a node within a strongly connected component had an incoming edge to it from outside the component, then the source node of that incoming edge should be ranked higher than any of the nodes where the root node is the source of an edge to.

3.4.1.1: Pseudocode

Our brute force algorithm, without optimizations, is explained at a high level using the pseudocode below. The algorithm takes a directed graph G as input, with vertex set V and edge set E, and outputs the best ranked ordering of vertices S. We iterate through each possible permutation of vertices in V and maintain the best total backedge weight and best permutation found so far. If the total backedge weight of the current permutation is less than the best found, then the current permutation and its total backedge weight are maintained. After all permutations have been evaluated, the best ranked ordering found is returned.

```
algorithm brute_force() is
      input: graph G = (V, E)
      output: best ranked ordering S

      best_weight <- weight(G)
      best_permutation <- {}
      for each permutation P of V do
            weight <- total_backedge_weight(P)
            if weight < best_weight do
                  best_weight <- weight
                  best_permutation <- P
            end if
      end for
      return best_permutation
end
```

3.4.1.2: Runtime Analysis

Because our brute force approach generates and calculates the total backedge weight of every possible ranking for a given graph, its time complexity is relatively poor. Our initial ranking is based on the order in which nodes are entered into our adjacency matrix, and is assembled in O(V) time. Each new permutation is generated in O(1), while the total backedge weight evaluation is completed for each permutation in O(V^2). This process is repeated for all V! permutations within the graph, resulting in a time complexity of O(V! * V^2 + V), or O(V!).

3.4.2: Berger/Shor New 2-Approximation Algorithm

As discussed in Section 2.4.2, the Berger/Shor New 2-Approximation algorithm was selected for this project to provide approximations of solutions for input graphs that brute force could not because of the input size. In general, approximation algorithms may not produce the best rankings for a graph, but they can produce acceptable solutions for cyclic graphs in polynomial time. This algorithm was selected because of its polynomial runtime and its factor of two correctness, which suggested comparable results to the brute force approach for larger graphs.

Our version of the Berger/Shor New 2-Approximation algorithm utilized the addition of topological sort to develop a total ordering from the acyclic output graph, and several preprocessors to improve the correctness of the output. The Berger/Shor algorithm was intended to solve the Maximum Acyclic Subgraph problem, so by design, it only returned an acyclic subgraph. Thus, topological sort could be applied to transform the output acyclic graph into a ranked total ordering, or ordering of all nodes in a graph.

Additionally, we designed our algorithm to preprocess the input for better results. Our first preprocessor was the strongly connected components preprocessor, which would determine the sets of nodes that were part of strongly connected components within the graph. These strongly connected components could then be evaluated by the Berger/Shor approximation, as any edges not included within a component are not part of cycles, thus they do not need to be removed. Our second preprocessor involved rearranging the order of nodes to be approximated within each strongly connected component. This ordering impacts which edges are removed during the approximation, as the removal of one set of edges earlier in the approximation may change whether another node has its incoming or outgoing edges removed, potentially resulting in a more accurate approximation.

3.4.2.1: Proofs

Lemma 1: The resulting graph is acyclic.

Proof: Suppose a cycle exists of vertices (v_1, v_2, … , v_n, v_1). After evaluation of vertex v_i in this cycle, where $1 \leq i \leq n$, either the edge <v_{i-1}, v_i> or <v_i, v_{i+1}> has been discarded by removing either the incoming or outgoing edges of v_i. Thus, a cycle can no longer be formed from v_1 to v_n [1].

Lemma 2: The output provided by this algorithm is always within a factor of two of the correct answer.

Proof: Each node v in graph G is evaluated once. The larger of the two sets of edges for each v is transferred to the output graph, while the smaller of the two sets is discarded. Therefore, at least half of all edges in the input edge set A are present in the output edge set A', meaning that $|A'| \geq (½)|A|$. Thus, the output graph is always within a factor of two of the potentially cyclic input, so it must be within a factor of two for the correct acyclic output [1].

Lemma 3: The algorithm terminates with a total ordering.
Proof: Following Lemma 1, the output graph from this algorithm is acyclic [1]. Therefore, a total ordering can be generated utilizing topological sort without entering an infinite loop.

3.4.2.2: Pseudocode

The following pseudocode provides a high-level description of how the Berger/Shor New 2-Approximation algorithm works. It begins with an input graph G with vertex set V and edge set E, and outputs an acyclic graph G' with vertex set V and modified edge set E'. The algorithm first determines the edges involved in each strongly connected component in the graph, removing them from the output graph. The remaining edges do not contribute to cycles and therefore can remain in the final graph. Then, the algorithm conducts the Berger/Shor approximation process on each strongly connected component, storing the resulting edges into the output graph.

Within the Berger/Shor approximation function itself, the strongly connected component input is received, while the acyclic component with edges removed is output. This function iterates through each vertex v in the input vertex set V and determines v's indegree and outdegree. If the indegree is larger than the outdegree for v, v's outgoing edges are removed from E, and v's incoming edges are copied over to E' before being removed from E. Otherwise, v's incoming edges are removed from E, and v's outgoing edges are copied to E' before being removed from E. Once this completes, it returns the acyclic subgraph, which is then topologically sorted to return the total ordering T.

```
algorithm berger_shor() is
      input: graph G = (V, E)
      output: total ordering T

      G' <- G
      scc <- strong_connect(G)
      remove scc edges from G'
      for component in sccs do
            new_component <- approximate(component)
            G' <- G' ∪ new_component
      end for
      return topological_sort(G')
end

algorithm approximate is:
      input: graph G = (V, E)
      output: acyclic graph G' = (V, E')

      G' <- G
      order nodes in G based on preprocessing heuristic
      for each v in V do
            if v.indegree > v.outdegree do
                  A <- outgoing edges from v
                  remove A from E

                  B <- incoming edges to v
                  G'.E' <- G'.E' ∪ B
                  remove B from E
            else do
                  A <- incoming edges to v
                  remove A from E

                  B <- outgoing edges from v
                  G'.E' <- G'.E' ∪ B
                  remove B from E
            end if
      end for
      return G'
end
```

3.4.2.3: Runtime Analysis

The Berger/Shor New 2-Approximation algorithm completes in greatly reduced time compared to the other algorithms implemented during this project. This algorithm visits every node within a given graph and determines the indegree and outdegree of that node, both operations which take O(1) time each. The smaller of the two sets of edges is discarded from the input graph, then the larger of the two sets of edges is copied to the final graph before being discarded from the input. Because edges are removed from the input graph in both cases at each iteration of the algorithm, each edge is processed only once in the algorithm, resulting in O(E) for all edge computations. Thus, our overall runtime is O(2V) + O(E), or O(V + E) [1].

3.4.3: Hill Climb

The third algorithm we implemented for our program was similar to the artificial intelligence algorithm Hill Climb. The basis for this algorithm originated from the concept that we may not need to remove cycles from the graph to compute a valid approximation. Our focus had been on using the total backedge weight metric and minimizing the total backedge weight for a ranking, a computation we could conduct quickly. Our Hill Climb implementation began with the nodes sorted by decreasing net edge weight, and evaluated swaps of two nodes in the ranking to see if the total backedge weight decreased. This process was repeated for all neighboring nodes in the ranking, and the ranking with the lowest total backedge weight would be chosen for this process to repeat upon again.

If the algorithm plateaued, where no swaps within a ranking reduced the backedge weight, we allowed for a set number of "sideways moves." A sideways move allowed for an equivalently-weighted ranking to be selected, even though it was no better, to see if Hill Climb could improve its backedge weight. Additionally, we implemented random restarts to this algorithm. Similar to its implementation in artificial intelligence, our random restarts began the Hill Climbing process again with a randomly-generated ranking if evaluation of a prior ranking had plateaued.

3.4.3.1: Pseudocode

The following pseudocode describes our Hill Climbing algorithm. This algorithm takes a directed graph G as input, with vertex set V and edge set E, and outputs a total ordering T. We begin with a permutation of all vertices in V, sorted by decreasing net edge weight. We maintain the best permutation found so far and its total backedge weight, and the current permutation to make swaps with. After each iteration, the permutation with the lowest total backedge weight is utilized as the starting permutation for the next iteration. Once hill climb cannot do any better, it conducts random restarts to repeat the process with a randomly-generated permutation as input. After all permutations and restarts have been evaluated, the total ordering with the lowest total backedge weight is returned.

```
algorithm hill_climb() is
      input: graph G = (V, E)
      output: total ordering T

      best_weight <- weight(G)
      best_permutation <- {}
      base_p <- order vertices by decreasing net edge weight
      for v in V do
            base_p <- swap(v, v+1) if in bounds
            weight <- total_backedge_weight(base_p)
            if weight < best_weight do
                  best_weight <- weight
                  best_permutation <- base_p
            end if
            base_p <- swap(v, v+1) to revert

            base_p <- swap(v, v+2) if in bounds
            weight <- total_backedge_weight(base_p)
            if weight < best_weight do
                  best_weight <- weight
                  best_permutation <- base_p
            end if
            base_p <- swap(v, v+2) to revert
            base_p <- best_permutation to repeat the cycle
      end for
      best_permutation = random_restarts(G, best_permutation)
      return best_permutation
end
```

3.4.3.2: Runtime Analysis

With regards to time complexity, our Hill Climb implementation was fairly efficient. In its implementation, we conduct v+1 and v+2 swaps for nodes, resulting in 2V swaps considered per iteration. Because each swap can be completed in O(1), each iteration of Hill Climb completes in O(2V). After each swap, the total backedge weight is evaluated, which completes in $O(V^2)$ time. The number of iterations performed within Hill Climb varies by ranking, unrelated to the number of vertices in the ranking. As discussed in [18], the resulting time complexity for hill climbing approaches is limited by the number of iterations completed d, expressed as O(d). Thus, our overall time complexity is $O(2V^3 \cdot d)$. However, our execution time limit and maximum number of iterations constant halts the algorithm far before it reaches a similar runtime to brute force.

3.4.4: Range of Correctness Search

The fourth algorithm we implemented was an extension of our Hill Climb approach discussed in the previous section. The basis behind this variation was that, unless given an input graph with the tournament constraint where every team has played every other team, there exists flexibility of the rank for some teams within the ranking generated, which we call the Range of

Correctness. The flexibility of rankings is accounted for in our evaluation heuristic for this algorithm, which tries to rearrange the nodes in a given ranking based on their net edge weights without generating any new backedges. Following hill climbing methodology, this heuristic is applied until the total backedge weight of the modified ranking cannot be improved.

3.5: Testing Considerations

In order to improve the efficiency of conducting test cases on our algorithms and rankings, we saw the need to automate parts of the testing process. We initially designed our program to support running multiple trials of our algorithms in succession, but this was restricted to one set of sports data, one configuration file, output only via the terminal console, and did not support external ranking evaluation. To address these restrictions, we designed a wrapper script that could run our program in a batch setup, and implemented file output support within the program. The wrapper script needed an input system to determine how to run the main program: which sports data to use, which configuration files to apply, and which ranking algorithms to use if generating a ranking, or which external ranking files to evaluate with. Based on our design, we proceeded with an input system for the script that read in a text file with the different trial instructions, which was then forwarded to the main program. Once the trials were completed, the output results could be read in by the script and organized as needed.

Alongside the rank generation script, we also wanted to explore rank comparison. Rank comparison, also known as rank differential, compares the differences in placement of teams between two rankings. We planned to use rank comparison testing to explore the differences between rankings generated by our program and rankings from external sources. In order to condense the comparison of rankings into a single metric, we decided to use the average difference in the placements of teams as our heuristic. Though the process of comparing rankings was originally completed by hand, we determined that the process was too time consuming for manual comparison to be viable for comprehensive rank comparison testing. Therefore, we designed a testing module in the program to automate the process. We required that this module execute the comparison of rankings, generate the average difference between team placements in the ranking, and output into a human-readable format. We could then use this module to explore what factors in our rankings produced the most similar rankings to external rankings, giving us insight on what factors were most highly considered in external rankings.

3.6: Summary

This chapter discussed the design decisions we faced during the introductory phase of the project and how we addressed them. We introduced our considerations for sports data formatting, storage, and representation within the graph data structure. Our algorithms were discussed in detail, with our pseudocode and adaptations showing the application of these algorithms to our project. We concluded this chapter with an introduction on how we planned to handle test cases and automation within our program.

Chapter 4: Base Implementation

After careful consideration about how we would design the base functionality of our program, we began implementation. This chapter focuses on the transition from the concepts discussed in our base design in Chapter 3 to functionality within our program. Each major feature, from our graph structure to our algorithms, is discussed in detail alongside justification for any design modifications made during implementation.

4.1: General Program Considerations

An implementation goal for the program was to maintain compatibility between Windows, Mac OS, and Linux operating systems. Individual Makefile recipes were created for each operating system configuration, so the user only needed to apply the correct recipe on any compatible compiler. In order to achieve this goal, the implementation of the program could not use any operating system specific libraries or functions. The only operating system specific functionality that was implemented was the use of multithreading using POSIX threads within the brute force algorithm, discussed in Section 4.2.1.1. Conditional compilation was used to disable the POSIX threads in Makefile recipes for Windows operating systems. The second issue with intercompatibility was a discrepancy in the format of program-generated files between Windows and Linux. Files generated by the program have operating system specific line endings due to how each operating system writes to the filesystem. Any files generated by one operating system could not be read by a different operating system if the line endings were different. This issue is most notable when exchanging the weight configuration files mentioned in Section 3.3.3.

4.1.1: Minimum Feedback Arc Set Evaluation Process

Once the edge weights were configured in the adjacency matrix, the program could refer back to the weights in the graph to calculate the total backedge weight. As the main heuristic for success in the Minimum Feedback Arc Set problem, the total backedge weight of a given ordering needed to be efficiently calculated using the weights stored in the adjacency matrix. Our evaluation process takes a ranking and steps through it in two nested loops. The outer loop iterates through the ranking itself and processes every node in the ranking. The inner loop examines every node that has been visited by the outer loop so far and checks if there has been an instance where a node later in the ordering has beaten a node earlier in the ordering, signifying a backedge. After a backedge has been detected, it is added to a running sum totaling the backedge weight of a given ordering.

In addition to the internal evaluation process for total backedge weight, we implemented the functionality for our program to generate the total backedge weight for an external ranking. External rank evaluation was implemented as a module separate from our rank generation process in our program. External rank evaluation allowed us to compare external rankings on the same basis as our internal rankings, such as with the total backedge weight metric.

4.2: Algorithms

This section details the transition of our four algorithms from the theoretical perspective described in the base design in Chapter 3 to the technical perspective within our program. The specific methodologies behind the implementation of each algorithm are explored, along with any challenges we faced and their resolutions, as well as additional functionality added.

4.2.1: Brute Force

Our initial brute force implementation followed the logic detailed in Section 3.4.1. Brute force only utilized the global adjacency matrix as input and output a vector of nodes indicating the best ranking found. Inside the function, a vector of nodes representing the first permutation was assembled using the number of nodes defined in the matrix. A vector for the best permutation found and its total backedge weight were initialized and updated throughout the process. Our brute force function utilized the C++ standard library next_permutation() function, which would return the next lexicographic permutation of a given input vector if one existed. We used a while loop to iterate through all available permutations from next_permuation() and evaluate them, updating our reference variables if better orderings were found. Once all permutations had been evaluated, the vector with the best ranking was returned.

Testing our brute force algorithm on sample graphs resulted in several cases where more than one ranking had the same total backedge weight, indicating equivalency in correctness according to our metric. We expanded upon the brute force algorithm to account for this by implementing a variation that maintained and returned all rankings with the same lowest backedge weight. This functionality was also extended to create a feature we called "brute force authentication," which was a method hook for other algorithms to pass in their generated rankings to and determine whether these rankings were equivalent to what brute force would generate as a solution. Adding brute force authentication was crucial in early testing where the correctness of other algorithms had not yet been determined.

4.2.1.1: Brute Force Modifications and Optimizations

During the implementation of brute force, we made modifications and optimizations to our evaluation heuristics in the interest of reducing computation time to make using the brute force algorithm more feasible. The first optimization we considered was multithreading support. Because brute force was evaluating all permutations of rankings for a graph, there was no need for a specific order of evaluation. Thus, the evaluation task could be split into separate threads and evaluated separately as the processor allowed. To divide the task, each thread would generate permutations as if the highest-ranked node, or head node, had been removed. The permutation was evaluated as if the head node was still in place, where each thread would have a list of nodes to use as the head. For example, using two threads and the nodes A, B, C, and D, Thread 0 would evaluate all permutations where A and C were the head, while Thread 1 would evaluate where B and D were the head. This optimization decreased computation time by a linear factor of the number of threads; sufficient for running on slightly larger graphs than before, but not computationally feasible for full-sized sports season graphs.

The second optimization implemented for brute force was a strongly connected components preprocessor. Using a strongly connected components preprocessor allowed us to discard certain permutations; for example, if node A was the only node that had an edge to node B, then any nodes with an edge to node A should be ranked higher than any edges that node B points to. However, this optimization would only improve performance in cases where the graph contained more than one strongly connected component. This strongly connected components optimization decreased runtime by reducing the number of nodes considered in the original O(V!) runtime within brute force. Runtime then became the summation of O(V'!) for the set V' of nodes in each component. However, runtime would increase by O(2V+E) due to the computation of the strongly connected components within the graph.

4.2.2: Berger/Shor New 2-Approximation and Preprocessors

As mentioned in Section 3.4.2, we implemented and modified the Berger/Shor New 2-Approximation algorithm to generate rankings. We followed the general program logic outlined in [1] to develop our base implementation. To begin, the algorithm required input of the adjacency matrix representing the input directed graph. This matrix was this copied into an equivalent adjacency matrix because this algorithm mutates the graph by nature, and we wanted to preserve the original graph if other algorithm runs needed to be performed afterwards. Once the matrix was duplicated, a second matrix named "a_hat" was initialized to the number of nodes in the graph, as this would hold the approximated output matrix. The duplicated matrix was then iterated on by column, representing the winner, where the indegree and outdegree for that node were summed and compared. The larger of the two values resulted in copying their adjacency matrix wins or losses to a_hat before zeroing out all edges connected to that node from the duplicates matrix. This process was repeated until all nodes had been processed, such that the duplicated matrix was emptied and a_hat contained the approximated adjacency matrix, which was then returned.

Due to the edge-removing nature of the Berger/Shor New 2-Approximation algorithm to break cycles, it removed edges that were not involved in any cycles, resulting in a sparser graph and worse rankings. These rankings were still correct approximations to the Maximum Acyclic Subgraph problem, but we saw the need to address this issue in the interest of potentially improving our approximations. As discussed in Section 3.4.2, we decided to use a strongly connected components preprocessor to address this, so we implemented Tarjan's Strongly Connected Components algorithm. Tarjan's algorithm is based upon Depth First Search and utilizes a stack to maintain when each node in the graph is first visited and when each is found again by edges from other nodes. Our implementation of this preprocessor has a time complexity of O(2V+E), as it visits every edge once to go to every node once when processing cycles, and iterates through all nodes again to place them in component form [26].

Our implementation of the strongly connected components preprocessor returned a vector of strongly connected components for the input graph, where each component was represented as a vector of the nodes it contained. Each component was used to construct a new adjacency matrix with these nodes and the edges between them to simulate a subgraph. The starting adjacency matrix for each component was subtracted out from the output adjacency matrix and passed into the Berger/Shor algorithm to be approximated, after which the approximation matrix was added back into the output adjacency matrix. By implementing the strongly connected components algorithm, we only applied the Berger/Shor approximation algorithm on collections of nodes that contained cycles, allowing edges outside of cycles to remain in the graph and improving the results of the algorithm with minimal overhead.

Additionally, we investigated preprocessing the ordering in which nodes were evaluated for edge removal in the Berger/Shor algorithm. We implemented this preprocessor to be applied to each strongly connected component before being processed by the Berger/Shor algorithm. The algorithm selection for Berger/Shor within the program was split to allow for an integer flag to indicate which preprocessor ordering to evaluate with. In addition to offering the user no preprocessing, several sorting options were added to the preprocessor: number of outgoing edges descending, number of outgoing edges ascending, ratio of outgoing to incoming edges descending, outgoing edge weight descending, and randomized. Within each option, a vector is maintained to indicate the ordering of nodes to be evaluated, which is modified depending on the option selected.

4.2.3: Hill Climb

Given the variance in hill climb's execution time, we implemented our Hill Climbing algorithm to be configurable in its minimum and maximum runtime. We accomplished this flexibility by allowing the number of random restarts and sideways moves to be configurable, where the minimum and maximum number of rankings evaluated could be hard-coded, so that evaluation time could be capped if needed. As discussed in Section 3.4.3, we decided to expand the evaluation portion to contain the n+2 neighboring node in order to potentially improve search results. We found that the runtime was low enough that examining the n+2 neighboring nodes would not increase computation time greatly, but would allow for consideration of additional swaps that could result in better rankings.

The basic implementation of the Hill Climbing algorithm for the Minimum Feedback Arc Set in our program followed the original design outlined in Section 3.4.3. The algorithm first created a preordering of teams based upon their net edge weights in the input graph. Then, a series of temporary swaps were performed upon this preordering, and the resulting ranking with the best swap with the lowest total backedge weight was saved. This continued until no further improvement in total backedge weight is seen. Sideways moves, as outlined in Section 3.4.3, were implemented with an easily configurable preprocessor-defined variable. If the best swap in one round of Hill Climbing resulted in an equivalent score, a sideways move would be expended. This process would continue until a better solution is found or until there are no more allowable sideways moves. As mentioned in Section 3.4.3, sideways moves were implemented to reduce the probability of getting stuck in a "plateau." The total amount of available sideways moves was set to 500 because they were seen to be relatively inexpensive in their total added computation time after some testing.

Beyond the implementation of sideways moves, we also implemented random restarts. As outlined in Section 3.4.3, once Hill Climbing completes evaluation of the initial preordering, the program will generate multiple randomized preorderings that the Hill Climbing algorithm will process in the hopes that one of the resulting permutations will result in a reduced total backedge weight. After the implementation of the random restarts was completed, we noticed that it substantially increased the computation time of the Hill Climbing process as expected. One iteration of Hill Climbing was relatively quick to complete on a set of 130 teams or less, as shown in the base results in Section 5.2.3. The time for execution of one iteration of Hill Climbing also grew with the number of nodes in a permutation because of the increased number of swaps necessary to complete per iteration. Due to the large variance in the execution time for Hill Climbing, dependent on the number of teams in the dataset, we decided that there was not a "one size fits all" number of restarts that should be allowed.

In order to manage the variable time requirement for Hill Climbing depending on the number of restarts, we implemented a hybrid system of timing and hardcoded values. The new system for restarts in Hill Climbing used a hardcoded lower bound and upper bound for the number of restarts. For our testing, we set the lower bound for the number of restarts to 20 and the upper bound to 300. We reasoned that an adequate minimum amount of restarts was 20 because the difference in time required between 1 and 20 restarts did not seem significant enough to sacrifice the potential for a better random preordering. We also determined that any more than 300 restarts was unnecessary for the amount of time that would be required. Therefore, the Hill Climbing process would always run with at least 20 random restarts and at most 300 random restarts. For the restarts occurring in between these bounds, we implemented a configurable timing system that would check how long the Hill Climbing process had been executing and would determine if another random restart was allowed, where the execution time was measured from the start of the first preordering Hill Climbing evaluates. For example, before allowing restart number 21, the system would first check if the hill climb process had exceeded its configurable maximum amount of execution time before proceeding. If Hill Climbing had exceeded its allotted time after restart number 100, the system would stop computing random restarts. If the system reached the upper bound of 300 random restarts without surpassing the configured maximum amount of execution time, the system would still terminate Hill Climbing because the upper bound had been reached. The maximum execution time could also be configured to the needs of the user with the command line argument "--hctime", which allowed testing using the Hill Climbing algorithm to be configured to the needs of the tester.

The final improvement made in the implementation of Hill Climbing was to extend the number of nodes checked in each swap to also check the n+2 neighbor as well as the n+1 neighbor. Essentially, each node would not only check a swap with its nearest neighbor but would also check the swap with the neighbor 2 positions away.

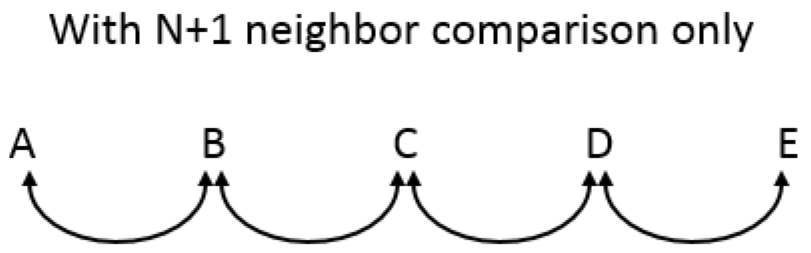


Figure 6: Initial Hill Climb implementation with N+1 neighbor comparison

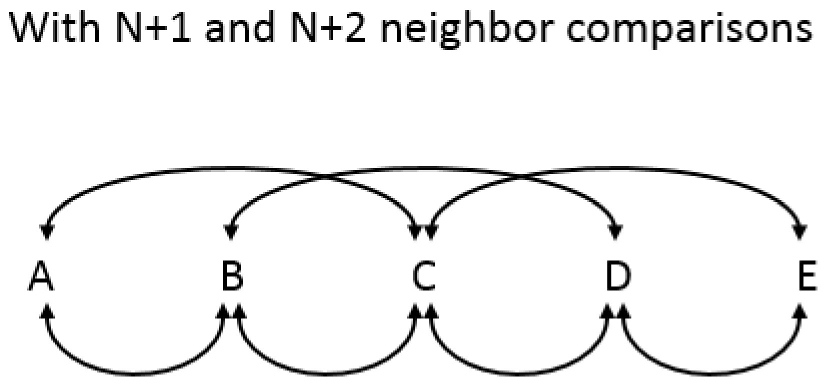


Figure 7: Revised Hill Climb implementation with N+1 and N+2 comparisons

This process is shown in Figures 6 and 7 above. Note that the arrows on the top of the nodes delineate the position n+2 swaps and the arrows below the nodes delineate the position n+1 swaps. As shown in Figure 7, this process results in (n-1) • (n-2) swaps compared to (n-1) swaps as originally implemented. We decided that the decrease in runtime performance was worth the possibility of finding a better permutation by exploring more swaps.

4.3: Data Collection Script

To gather relevant sports data, several scripts were written in Node.js utilizing the jsdom package from npm. The main collection script can gather and process NBA, NFL, NHL, MLB, and college football data from sports-reference.com. The data were mostly formatted inside similar HTML tables which allowed for similar processing. The data pages were pulled through simple HTTP GET requests and used jsdom so that the data tables could be easily found and processed row by row. Each row contained information about a single game, with some varying columns, and was stored inside its own object. The list of game objects could then be written out in our input file format as well as in JSON. JSON output allowed data to be easy reprocessed, whether to remove games we did not want or for conversion into a different format, without the need to pull information from the page again.

The script featured various options, including: cumulative output into one file, the ability to include only games through a given week and date in the season, our output format, and JSON output. The cumulative output option was necessary to combine months of NBA data due to the way sports-reference.com formatted the data. The options to include games through a week or date were specific to each sport. NFL and college football games are organized by week, but MLB, NBA and NHL games are only identified by date. Another script was also written to prune college football games involving at least one team with less than some number of games in the season dataset. Since Division I teams did not always exclusively play Division I teams, teams that were not relevant to our ranking inflated node counts of our college football datasets from 128 to 216.

Table 1: Sports Data and Rankings Collected For Each League

League	Data collected	Rankings collected
College football	2015-17	2015-17
MLB	2011-15, 2017	2011-15
NBA	2016-18	-
NFL	2012-18	2012-18
NHL	2013-18	2013-17

4.4: Bash Testing Script

With all of the aforementioned features implemented over the course of this project, we had many opportunities to test our data and algorithms to draw conclusions. All of this testing, however, led to increased burden with conducting test cases. All trials with our program had to be entered via the command line with an additional input layer to choose the algorithm for processing. Given the inefficiencies of running test cases from the command line, we saw the need to develop a script to automate portions of testing. As mentioned in Section 3.5, we turned towards a wrapper script to aid in testing. We chose to write this script in Bash because it needed to launch our main program, but we did not want to have to manage process forking and permissions, especially with the differences in process handling between Linux and Windows. This decision sacrificed some performance for the ease of development and portability.

At first, the Bash script was implemented to take a file with a predefined set of arguments as input and forward these inputs to individual launches of the program. This first implementation was a large improvement as trials could be saved and repeated easily with the input file format, removing the need to utilize the command line to run the program in batch. One drawback to this script was that all program trials were only output to the console, so it was harder to analyze output logs, especially with limited console window space. An additional drawback was that the most frequent command line arguments and options were hard-coded into the Bash script; although this provided some simplicity with inputting the trial information into the input file, it required additional updates anytime a feature or argument was added or changed in the program.

To address the first drawback, the program received file logging support so that testers would no longer need to look for data in the console after running test cases. Further enhancing this feature, we extended the Bash script to read in log files that were generated during the trials which allowed us to easily manipulate and view data from the test cases. The script output this data to CSV files, where different rankings, total backedge weights, computation times, and comparisons of pre- and post-processors could be compared side-by-side with ease. To address the second drawback of command line limitations, the Bash script input format was modified to take the program arguments verbatim, with the algorithm choice separate. This greatly increased the flexibility of the script, as no changes were necessary internally if any new functionality was added to the program, as these new arguments could be placed into the input file and run immediately.

4.5: Rank Comparison Testing Module

The rank comparison testing module was designed to facilitate the comparison of rankings in an automated module. This module was built as an add-on to the rank generation process of the program and could be used without generating any rankings at all. The implementation for the rank comparison module was separated into two parts. The first part of the rank comparison testing module compared the two rankings and calculated the average difference in placements, and then generated the output of the comparison process. The second part of the rank comparison script module automated the generation of comparisons to test the ability for the rankings generated by our algorithms under different edge weight configurations to match an external ranking. The rank generation process was separated in this way to service the needs of testing specific rankings while also automating the large tests of multiple configurations.

4.5.1: Generating Rank Comparisons

The first part of the rank comparison testing module, designed to generate and output comparisons, was implemented to compare two ranking files. We reasoned that a rank comparison is ultimately a calculation on the different positions of teams in two separate rankings. For the purposes of calculating the comparison, we defined these two rankings as the control ranking and the test ranking. This process measured the positions of teams in the test ranking compared to their positions in the control ranking. The difference in position for each team was calculated and used to determine the overall average difference in position for the ranking. This average difference in position could then be utilized as the heuristic to determine how similar one ranking was to another, where a lower average difference would signify greater similarity in the rankings.

To complete the rank comparison calculation, the rank comparison testing module first imported the control ranking. The required format for the control ranking was implemented as a list of the teams in their ranked order. The list format was used because it corresponded with the format of our external rankings and the format of rankings generated by our program. During the import for the control ordering, the size of the ranking was captured and used as the size for which to hold all test rankings to. We reasoned that it did not make sense to compare rankings that were not the same size. This value was also useful for allocating the appropriate amount of memory to contain the one or more test orderings.

After importing the control ranking, the rank comparison testing module imported the test ranking. The test ranking input could be one of two types. The first type of input was a list input which, for the same reasons as using a list input for control ranking, allowed a flexible input of any external ranking. The second input type for a control ordering was a program-generated results file. Results files were already in use by the program and the automated test script described in Section 4.4. Importing these results files allowed us to quickly generate multiple rankings with different configurations and use the generated rankings in rank comparisons. Furthermore, a results file allowed us to import more than one ranking as test rankings for the rank comparison testing module. Since an arbitrary number of rankings could be recorded on a results file, we simply parsed through each of the rankings on the results file and imported them for testing. This process required the implementation of a small preprocessor to count the number of rankings on the results file to allocate the appropriate amount of memory for the test rankings. Regardless of the input type of the test ranking file, the differences in ordering between the control ranking and each test ranking was then computed. The computation process for a rank comparison involved iterating through each team in the control ordering and finding the corresponding position in the test ordering. Using the two positions, the difference in position was calculated, and ultimately all values in the set of differences in position were averaged together to result in the final heuristic for a rank comparison. If a results file was used, the average difference in rank position was computed in this fashion for all test rankings.

We implemented the automatic output of the results of a rank comparison into a CSV file for easy viewing. An example output of the automated process is listed Table 2.

Table 2: Sample Rank Comparison Output

Rank	2017nfl_usatoday_top10.txt	sample_ordering.txt	Net Difference
1	New_England_Patriots	New_England_Patriots	0
2	Pittsburgh_Steelers	Pittsburgh_Steelers	0
3	Minnesota_Vikings	Minnesota_Vikings	0
4	Los_Angeles_Rams	Los_Angeles_Rams	0
5	New_Orleans_Saints	New_Orleans_Saints	0
6	Kansas_City_Chiefs	Jacksonville_Jaguars	1
7	Carolina_Panthers	Kansas_City_Chiefs	3
8	Jacksonville_Jaguars	Atlanta_Falcons	2
9	Atlanta_Falcons	Los_Angeles_Chargers	1
10	Los_Angeles_Chargers	Carolina_Panthers	1
			Avg: 0.8
	Score: 1.5625		Score: 2.42188
Minimum Average Diff: 0.8	in: sample_ordering.txt		
Control Score: 1.5625	Min Avg Diff Test's Score: 2.42188		
Using Quartile Evaluation: True			

As shown in the output in Table 2, the rank comparison testing put the sample orderings into a table and displayed the differences in the rankings. The total backedge weight, listed as "score" above, was also displayed, allowing for an easy comparison between the relative strengths of each ordering. The edge weights in this test case were normalized, leading to the non-integer values of the total backedge weight. The rank comparison output was also implemented to display which test ordering had the smallest average difference in ordering. This was helpful in situations where there were multiple test rankings being compared.

Though this approach to rank comparison testing was adequate for comparing most rankings, it was not suitable for truncated rankings. Truncated rankings, explained further in Chapter 7, are rankings only contain the top-25, top-10, or top-n teams out of a superset of teams. In order to handle truncated rankings, we adopted the notion of an "N+1" node in each truncated ranking that would act as a placeholder for the rest of the teams outside the top-n superset. In cases where a team in a test ranking was not also in the corresponding control ranking, we assumed that the placement of the team in the control ranking was in the N+1 position in the control ranking. For example, in two top-10 rankings, if Team A placed in 9th place in the test ranking but was not present in the control ranking, the testing system would consider Team A's place in the control ranking to be at the 10+1 position in the control ranking, rank 11. Therefore, the difference in rank for Team A between both rankings would be 2.

4.5.2: Automated Rank Comparison Testing

The second part of the rank comparison testing module dealt with the automation of our rank comparison testing. Using the first part of the rank comparison testing module, we could efficiently calculate and output the comparison of team positions in rankings. In order to utilize rank comparisons to determine what configurations of our ranking algorithms led to similar rankings, we needed to run a large quantity of tests. We wanted to test different configurations of our edge weight factors in rank generation, which are further explained in Section 6.1.4. The total number of rank comparisons required extensive time when manually exporting each output ranking and rerunning the program to generate the rank comparison. It was clear that automation was necessary in order to cover the breadth of rank comparisons we required, so we needed to consider to what degree we should devote resources into automating the rank comparison process.

We considered computing all of the rankings and comparisons internally in its own module. This solution would have led to the cleanest implementation and most customizable output. The first of two drawbacks to this idea was that it would require new versions of every ranking algorithm so that they could return the ranking to the system rather than display the ranking and exit. The second issue was that the time required to implement this approach to automation would not have been worth the benefits of this feature.

We decided to implement the automation for the rank comparison testing in a way that utilized as much of the existing functionality in the program as possible. We recalled that the existing rank comparison computation and output could already handle an arbitrary number of rank comparisons, as long as the test orderings were consolidated in a results file. We concluded that we could automate the testing by automating the generation of one large results file with all of the testing rankings present and use the existing rank comparison functionality to accomplish this task.

The ranking algorithms were already programmed to output to results files and the existing rank comparison system could handle these results files. We simply needed to automate the generation of a results file with the correct orderings to be used in the rank comparison. At the time of implementation, we already had the functionality to reset edge weight configurations for the rank generation at runtime. We designed the process of creating ranking data to utilize a loop to iterate through the different configurations and trigger the running of the chosen ranking algorithms. As the different rankings were generated, the results would be recorded in the growing results file. Once each edge weight configuration had been tested, a rank comparison between a specified control file and the algorithm ranking results in the results file was automatically initiated, creating a full rank comparison output for the rankings generated with the different configurations. This satisfied our need for automated testing of rank comparisons.

4.6: Summary

In this chapter, we discussed all of the features and functionality created during the base implementation of this project. We began by explaining decisions made at the program level regarding changes or complications during implementation. We discussed the implementation of our algorithms and the optimizations utilized. Then, we introduced how our data collection script and bash testing script were implemented, and how we added functionality for rank comparison testing to the main program.

Chapter 5: Base Results

Upon implementing the base functionality of our graph structure, ranking algorithms, and edge weight algorithms, we ran test cases on our algorithms to better understand how they perform. This chapter demonstrates two types of test cases: correctness testing, which is concerned with minimizing the total backedge weight within rankings; and performance testing, which is concerned with minimizing the runtime of our algorithms. All total backedge weight values have been normalized to the total weight of the graph, and are displayed and analyzed as such. We test the changes in performance and correctness with our optimizations to our algorithms on sample graphs and on a sports dataset, and include and discuss the results for each trial.

5.1: Correctness Testing

In order to generate accurate rankings with our full sports data sets, we first needed to conduct test cases on smaller sample graphs to determine the correctness of our algorithms in certain scenarios. We generated several small sample graphs so that our brute force implementation could complete in reasonable time and serve as a baseline for the correctness of our Berger/Shor and Hill Climb approximation algorithms. Correctness testing our approximation algorithms on small graphs would allow us to anticipate how well they would perform on larger graphs that would not be computationally feasible for brute force.

In this chapter, we refer to several sample graphs that we generated for testing, which are listed in Table 3 alongside their properties such as number of nodes, number of edges, whether or not they are weighted or cyclic, and how many strongly connected components they contain.

Table 3: Sample Graphs Used For Base Results Testing

Graph name	Nodes	Edges	Weighted?	Cyclic?	SCCs
Input	4	4	No	No	4
Input3	6	6	No	Yes	4
Input_test3	3	5	No	Yes	1
Input_hard	10	18	Yes	Yes	3
9nodes	9	15	Yes	Yes	1
10nodes	10	13	Yes	Yes	7
11nodes	11	18	Yes	Yes	6

5.1.1: Brute Force

Running our brute force algorithm on several of our sample graphs allowed us to determine the rankings with the lowest total backedge weight for each graph, an important point of reference when testing our other algorithms. The first two tests we ran were to analyze the ranking correctness and runtime performance of weighted and unweighted graphs as a starting point for our brute force implementation with no optimizations. Input, Input3, and Input_test3 were used as the unweighted test graphs in these tests, and 9nodes, 10nodes, 11nodes, and input_hard were used as the weighted test graphs. The total backedge weights for the best rankings for each graph are shown in Table 4.

Table 4: Brute Force Weighted and Unweighted Correctness

	Backedge Weight / Graph Weight	Runtime
Input_test3	2 / 5 (40%)	0s 56µs
Input	0 / 4 (0%)	0s 227µs
Input3	1 / 6 (16.66%)	0s 1376µs
9nodes	6 / 86 (6.976%)	0s 895467µs
10nodes	4 / 78 (5.128%)	9s 602567µs
Input_hard	5 / 31 (16.12%)	9s 514241µs
11nodes	8 / 96 (8.333%)	113s 726752µs

The runtime for these tests show that our brute force approach was almost instantaneous for graphs with fewer than nine nodes and set the benchmark for any optimizations we added, but that the runtime increases greatly when graphs with more nodes were introduced. Because our brute force approach guaranteed correctness in its rankings for all input graphs, the only optimizations we could make improved the runtime of brute force.

The second set of tests we conducted were to determine how significantly our strongly connected components preprocessor reduced runtime for brute force. We expected that this preprocessor would greatly reduce runtime in cases where several strongly connected components were present in the input graph, but would slightly increase runtime if only one strongly connected component existed due to the additional runtime to determine the strongly connected components.

Table 5: Brute Force Comparison With and Without Strongly Connected Components (SCCs)

	Without SCC	With SCC
9nodes Backedge Weight Total Runtime	6 / 86 (6.976%)	6 / 86 (6.976%)
	0s 895467µs	**0s 68951µs**
10nodes Backedge Weight Total Runtime	4 / 78 (5.128%)	4 / 78 (5.128%)
	9s 602567µs	**0s 289µs**
Input_hard Backedge Weight Total Runtime	5 / 31 (16.12%)	5 / 31 (16.12%)
	9s 514241µs	**0s 15443µs**
11nodes Backedge Weight Total Runtime	8 / 96 (8.333%)	8 / 96 (8.333%)
	113s 726752µs	**0s 536µs**

The tests in Table 5 indicate that the strongly connected components preprocessor substantially reduces brute force runtime in graphs with several strongly connected components, with performance gains by a factor of over one million in the case of 11nodes. This optimization demonstrates that larger graphs, which would have been computationally infeasible before, can now be processed in reasonable time if they contain several strongly connected components. However, this places dependence on the strongly connected components of the input graph, rather than just on the number of nodes. For example, because Input_hard only had three strongly connected components compared to 10nodes with seven, Input_hard's runtime could not be reduced as substantially.

The third set of tests we conducted applied our multithreading optimization to the strongly connected components variation of our brute force algorithm. We conducted these tests on the same four graphs as above, where each graph was evaluated with one thread, two threads, and four threads.

Table 6: Brute Force Threading Evaluation Times

	1 Thread	2 Threads	4 Threads
9nodes Backedge Weight	6 / 86 (6.976%)	6 / 86 (6.976%)	6 / 86 (6.976%)
Total Runtime	0s 58284μs	0s 37683μs	**0s 23136μs**
10nodes Backedge Weight	4 / 78 (5.128%)	4 / 78 (5.128%)	4 / 78 (5.128%)
Total Runtime	0s 853μs	**0s 840μs**	0s 1014μs
Input_hard Backedge Weight	5 / 31 (16.12%)	5 / 31 (16.12%)	5 / 31 (16.12%)
Total Runtime	0s 8289μs	0s 4415μs	**0s 4202μs**
11nodes Backedge Weight	8 / 96 (8.333%)	8 / 96 (8.333%)	8 / 96 (8.333%)
Total Runtime	**0s 1056μs**	0s 1101μs	0s 1096μs

Our tests with multithreading were not completely as expected: we suspected that there would always be a linear decrease in runtime if more threads were applied, provided that the CPU could support them. While 9nodes and Input_hard both saw noticeable improvements in runtime from one thread to two, 10nodes and 11nodes both saw negligible changes. Further, only 9nodes had legitimate improvement when running with four threads compared to two. We suspect that this is because the runtimes for the other graphs were already optimized substantially from the strongly connected components processor, where the benefit of having additional threads for evaluation was almost negated by the overhead for each thread. However, we believe that multithreading is a substantial improvement, especially for input graphs with only one strongly connected component, which still have room for optimization.

5.1.2: Berger/Shor New 2-Approximation Algorithm

Using our brute force results as baselines, we conducted tests on our Berger/Shor approximation algorithm implementation. Because this algorithm is an approximation, we were expecting performance gains compared to brute force at the expense of correctness. The first set of tests we conducted compared our initial implementation of the Berger/Shor algorithm to our variation with Tarjan's Strongly Connected Components preprocessor. Our expectation was that this preprocessor would not have any benefit where the graph has one strongly connected component, but that it could increase correctness when several components are present. We conducted tests on the following four sample graphs: input_test3 and input, both unweighted; input_hard and 9nodes, both weighted.

Table 7: Berger/Shor Strongly Connected Components Comparison Results

	Backedge Weight / Graph Weight (Without SCCs)	Backedge Weight / Graph Weight (With SCCs)
Input_test3	2 / 5 (40%)	2 / 5 (40%)
Input	0 / 4 (0%)	0 / 4 (0%)
Input_hard	12 / 31 (38.70%)	12 / 31 (38.70%)
9nodes	12 / 86 (13.95%)	12 / 86 (13.95%)

In terms of correctness, the baseline Berger/Shor algorithm performed well on Input_test3 and Input, each generating a ranking equivalent in total backedge weight to brute force. However, performance on Input_hard and 9nodes was not as good, where Berger/Shor received 38.70% and 13.95% compared to brute force's 16.12% and and 6.97% respectively. We suspect that this is because our baseline Berger/Shor implementation was designed for unweighted graphs and only considered number of edges rather than edge weights.

The results in Table 7 indicate no improvement in total backedge weight for rankings generated with the strongly connected components preprocessor on our sample graphs. In each test case, the rankings produced were identical. We expect that maintaining the edges between the strongly connected components within our sample graphs may not have been significant enough to impact the output ranking to maintain more edges and develop a ranking with lower backedge weight. However, we suspect that on larger graphs with more strongly connected components, improvements would be noticeable.

The second set of tests we conducted with our Berger/Shor algorithm was with our node-sorting preprocessor. We implemented several sorting arrangements for this algorithm: number of outgoing edges descending, number of outgoing edges ascending, ratio of outgoing to incoming edges descending, outgoing edge weight descending, and randomized. We expected that the sorting process would add minimal overhead to the Berger/Shor algorithm and potentially improve the correctness of the resulting rankings. Our tests showing the total backedge weight using our two sample graphs, input_hard and 9nodes, are shown in Table 8.

Table 8: Berger/Shor Node Sorting Preprocessor Results

	Input_hard	9nodes
Y1 (descending number of outgoing edges)	10 / 31 (32.25%)	8 / 86 (9.30%)
Y2 (ascending number of outgoing edges)	8 / 31 (25.80%)	12 / 86 (13.95%)
Y3 (descending win-loss ratio)	11 / 31 (35.48%)	8 / 86 (9.30%)
Y4 (descending outgoing edge weight)	**6 / 31 (19.35%)**	8 / 86 (9.30%)
Y5 (no sorting)	12 / 31 (38.70%)	12 / 86 (13.95%)
Y6 (randomized)	7 / 31 (22.58%)	**6 / 86 (6.97%)**

In both sample graphs, the node sorting preprocessor shows improvements of the total backedge weight for the ranking generated, depending on which sorting methodology was selected. In both cases, no sorting (Y5) resulted in rankings that were tied for or had the highest total backedge weight. Sorting by outgoing edge weight (Y4), or weight of all wins by each team, yielded consistently positive results. Additionally, randomized sorting (Y6) yielded good results, though we are skeptical of consistently good rankings in repeated tests. As expected, sorting by number of outgoing edges (Y1 and Y2) did not yield good results because these tests were conducted on weighted graphs, and these cases do not consider edge weight when sorting the nodes.

To apply the node sorting preprocessor to practice, we conducted test cases on our 2016-17 NFL dataset. Each test had the strongly connected components preprocessor enabled and utilized node sorting methodologies Y1-Y6, similar to the above tests. Only the top-10 teams in each ranking are displayed, but the total backedge weight is representative of the full ranking.

Table 9: Berger/Shor Results on 2016-17 NFL Data

Y1	Y2	Y3	Y4	Y5	Y6
Patriots	Cardinals	Patriots	Patriots	Redskins	Steelers
Cowboys	Buccaneers	Cowboys	Steelers	Steelers	Cowboys
Steelers	Falcons	Steelers	Cowboys	Patriots	Redskins
Giants	Texans	Seahawks	Falcons	Colts	Giants
Seahawks	Titans	Falcons	Cardinals	Cowboys	Broncos
Dolphins	Dolphins	Cardinals	Seahawks	Giants	Colts
Raiders	Steelers	Chiefs	Chiefs	Dolphins	Titans
Chiefs	Chiefs	Raiders	Bills	Seahawks	Packers
Falcons	Raiders	Packers	Packers	Raiders	Buccaneers
Cardinals	Packers	Giants	Eagles	Titans	Texans
647 / 2623 (24.66%)	714 / 2623 (27.22%)	**521 / 2623 (19.86%)**	594 / 2623 (22.64%)	758 / 2623 (28.89%)	747 / 2623 (28.47%)

These results demonstrate significant change in rankings generated with and without the node sorting preprocessor. Our base implementation with no sorting (Y5) generated a ranking with more than 10% additional backedge weight compared to our best sorted ranking (Y3). Our unweighted edge weight preprocessors (Y1 and Y2) performed well without considering edge weights, but our net weight sorting (Y4) produced the second best results.

5.1.3: Hill Climb

We performed correctness testing on our Hill Climbing algorithm to understand how it performed in general and to determine if there were any cases where it performed better or worse overall. In the first set of tests, we wanted to compare the correctness and execution time on two graphs using our initial n+1 comparison approach and our revised n+2 comparison approach. The two graphs we used for these tests were Input3 and Input_test3, and we have included the results in Table 10.

Table 10: Hill Climb N+1 vs N+2 Correctness

	Input_test3	Input3
N+1 Comparisons Only Backedge Weight Total Runtime	2 / 5 (40%)	1 / 6 (16.66%)
	0s 57µs	0s 225µs
N+1 and N+2 Comparisons Backedge Weight Total Runtime	2 / 5 (40%)	1 / 6 (16.66%)
	0s 194µs	0s 243µs

In terms of correctness, our baseline Hill Climbing algorithm generated rankings with the same total backedge weight as brute force. However, these tests do not show any difference between ranking produced and its total backedge weight when using n+1 comparisons versus n+1 and n+2 comparisons. We suspect that this is due to the small size of the sample graphs tested and expect better results in graphs with more nodes and a higher likelihood of plateauing. In the case of runtime, a small increase appeared with n+2 comparisons, however we suspect that this change will not make hill climb computationally infeasible with our larger sports season graphs.

The second set of correctness tests we conducted on our Hill Climbing approach tested our random restart functionality. For these tests, we limited each trial to 300 random restarts and utilized the n+2 comparison functionality tested above. The results of our tests are shown in Table 11.

Table 11: Hill Climb Random Restart Correctness

	Input_hard	9nodes
No Random Restarts Backedge Weight Total Runtime	6 / 31 (19.35%)	10 / 86 (11.62%)
	0s 708µs	0s 432µs
Random Restarts Backedge Weight Total Runtime	**5 / 31 (16.12%)**	**6 / 86 (6.97%)**
	0s 57467µs	0s 47070µs

These tests demonstrate that random restarting offers an improvement in correctness when dealing with our sample graphs, indicating that even on small graphs, plateauing is possible and can hinder the correctness of hill climb. We expect that random restarts can greatly improve the correctness with sports season graphs, where more nodes and edges increase the likelihood for more local minima. However, random restarts did increase computation time for Hill Climb substantially, even for our sample graphs. We suspect that 300 random restarts may be too many for larger graphs where each restart is likely to take longer, but that implementing a smaller number of restarts can still improve correctness without significantly compromising runtime.

The final modification made to Hill Climb was the implementation of "sideways moves." This modification was made to combat plateauing when searching for a solution by allowing hill climb to consider equally-weighted rankings for the next iteration if no improvements in rankings occurred during the current iteration. The following tests on sample graphs utilize n+2 comparisons and allow for 300 random restarts. In the cases where sideways moves are enabled, 500 sideways moves are allowed before Hill Climbing completes.

Table 12: Hill Climb Sideways Moves Correctness

	Input_hard	9nodes
No Sideways Moves Backedge Weight	5 / 31 (16.12%)	6 / 86 (6.97%)
Total Runtime	0s 57467µs	0s 47070µs
500 Sideways Moves Backedge Weight	5 / 31 (16.12%)	6 / 86 (6.97%)
Total Runtime	5s 513326µs	4s 495037µs

In the above test cases, sideways moves appear to make no improvement towards the total backedge weight of the rankings generated. We hypothesize that this is because the local minima of our sample graphs are well-defined and are not close to other local minima. In practice, this means that permutations that result in total backedge weights that are close to a local minima are not similar to permutations that result in backedge weights close to other local minima. The addition of sideways moves also reduces the performance of Hill Climb substantially, as each permutation utilized is allowed 500 moves within 300 random restarts. We suspect that this may result in a significant increase in runtime if evaluating larger graphs, where the additional time of random restarts and sideways moves is magnified by the base computation time for each permutation.

Finally, to demonstrate the improvements of our Hill Climbing algorithm in practice, we conducted a comparison test on our 2016-17 regular-season NFL data set. These tests depict the differences between our initial Hill Climb approach with no modifications and our finalized Hill Climb approach with n+2 comparisons, 300 random restarts, and 500 sideways moves. Only the top-10 teams are displayed below, but the total backedge weights for each ranking are indicative of the full ranking.

Table 13: Hill Climb Results on 2016-17 NFL Data

Base Hill Climb	Modified Hill Climb
New England Patriots	New England Patriots
Atlanta Falcons	Atlanta Falcons
Dallas Cowboys	Dallas Cowboys
Pittsburgh Steelers	Green Bay Packers
Kansas City Chiefs	Pittsburgh Steelers
Arizona Cardinals	Kansas City Chiefs
Green Bay Packers	Arizona Cardinals
Seattle Seahawks	Seattle Seahawks
Denver Broncos	Denver Broncos
Philadelphia Eagles	Philadelphia Eagles
515 / 2623 (19.63%)	**509 / 2623 (19.40%)**
0s 5799µs	101s 547370µs

The above tests show a small improvement in the correctness of our Hill Climbing approach. We expect that this small improvement is due to the baseline Hill Climbing algorithm beginning with a ranking with small enough total backedge weight where there was little room for improvement, especially considering that the ranking from the baseline Hill Climbing algorithm has less backedge weight than the best Berger/Shor ranking. Both Hill Climb rankings generated contain the same ten teams, but have only four teams swapped within their rankings. This indicates that our data set has well-defined minima and maxima, as our modified Hill Climbing algorithm was able to find these variations in its ranking. However, this improvement in correctness comes at a large increase in runtime of the algorithm, which is now several orders of magnitude slower than our base Hill Climb approach.

5.2: Performance Testing

To assess the performance of our algorithms, timing tests were completed with each algorithm. Our approximation algorithms were evaluated using the same set of randomly generated graphs of varying densities and sizes. Three graphs were generated for each permutation of 10, 100, and 1000 nodes and densities 0.2, 0.4, 0.6, 0.8, and 1. All tests used a single thread on the same Intel i7-4790K processor running at 4.0 GHz. These tests results may have varying results if repeated, as the operating system and other background programs could have influenced the execution time of the algorithms. Due to its time complexity, brute force was evaluated as stated in the following section.

5.2.1: Brute Force

The performance of brute force was evaluated using graphs where every node was connected to two other nodes such that a single cycle was formed, e.g. A-B, B-C, C-A. All edges were given weights such that every possible permutation would be an optimal solution. However, these sample graph choices would not impact the performance of the algorithm. Brute force operates independent of graph density, number of edges, and how nodes are connected. The algorithm only depends on the number of nodes in the graph.

Table 14: Brute Force Threading Execution Times in Seconds

Nodes	1 thread	2 threads	3 threads	4 threads
8	0.064	**0.030**	0.039	0.034
9	0.674	0.320	0.215	**0.205**
10	7.375	3.438	2.819	**2.153**
11	88.527	44.763	30.042	**22.487**
12	1154.832	534.362	356.373	**269.512**

For any graph with 8 or fewer nodes, brute force ran instantaneously. As expected, each increase by one node increased execution time by slightly more than a factor of n!/(n-1)!, or n. Beyond 13 nodes, brute force takes an excessively long time to complete. Multithreading was able to decrease execution time by a factor of nearly the number of threads created. Unfortunately, this was not enough of an improvement to make a significant impact. Using the rate of permutations processed from the 12-node, two-threaded test, a single processor running at max usage on a 16 node graph would take 68 days to complete. A graph the size of an NFL dataset with 32 nodes would take $2.327*10^{21}$ years.

Table 15: Estimations of Brute Force Execution Time with 8 Threads (i7-4790K at 4.0 GHz)

Nodes	Predicted execution time
12	134 seconds
13	29 minutes
14	7 hours
15	4 days
16	68 days
20	21516 years
32	$2.327*10^{21}$ years

The expected runtime results in Table 15 were based on the rate of permutations processed per second from the 12-node, 2-threaded test, 448199.5351 permutations/second.

5.2.2: Berger/Shor New 2-Approximation Algorithm

As stated above, approximation algorithms were evaluated using 45 random graphs of different sizes and densities. Based on the averages in Table 16, the algorithm's performance appears to closely follow the expected O(V+E) time complexity from 10 to 100 nodes. The times for the 1000 node graphs, however, were an order of magnitude longer than expected. This could be due to a large number of memory operations caused by the high node count and maintenance of several adjacency matrices significantly impacting performance. Overall, the algorithm is quick.

Table 16: Average Berger/Shor Execution Time on Random Graphs in Seconds

	Density					
Nodes	0.2	0.4	0.6	0.8	1.0	Average
10	0.000318	0.000153	**0.000146**	0.000149	0.000164	0.000186
100	0.016147	0.017201	0.016495	0.016187	**0.014591**	0.016124
1000	11.716751	13.540069	14.652580	14.562917	**11.409153**	13.176294

Graph density appears to have an interesting effect on the runtime of the Berger/Shor algorithm. For both 100 and 1000 nodes, graphs with every possible edge had the shortest execution time, while graphs with mid-range density took longer. This effect was reversed for 10-

node graphs. For unknown reasons, the 10-node graphs of density 0.2 took over twice as long to execute than those of mid-range densities.

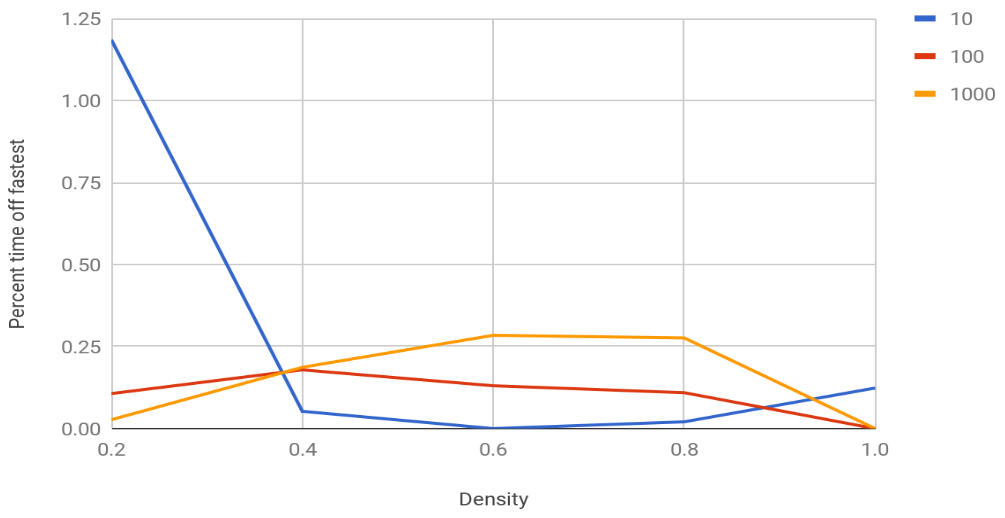


Figure 8: Berger/Shor percent time off fastest density for each node count

5.2.3: Hill Climb

Using the same set of graphs, a single iteration run to completion of the Hill Climbing approximation algorithm took significantly longer than the Berger/Shor algorithm. For graphs of 1000 nodes, execution took several hours to complete. This was expected, as we had calculated the time complexity of Hill Climbing as $O(V^3)$. The difference between the execution times of the 10 and 1000 node graphs demonstrated this. However, the 100 node graphs ran twice as fast as expected.

Table 17: Average Hill Climb Execution Time on Random Graphs in Seconds with Comparison

	Density						
Nodes	0.2	0.4	0.6	0.8	1.0	Average	B/S Avg
10	0.015727	0.016845	0.019644	**0.014929**	0.015030	0.016435	0.000186
100	**6.634456**	6.918450	7.073458	7.068419	7.368316	7.012620	0.016124
1000	**11296.777**	14201.902	15152.427	16458.629	16610.378	14744.023	13.176294

Graph density had a significant effect on execution time. For all three node counts, runs mostly took longer the denser the graph was. This is due to the fact that more edges mean there are more possible valid swaps Hill Climb could make. The 10-node graphs with densities 0.8 and 1.0 were the only exceptions to this, completing faster than the other 10-node graphs instead.

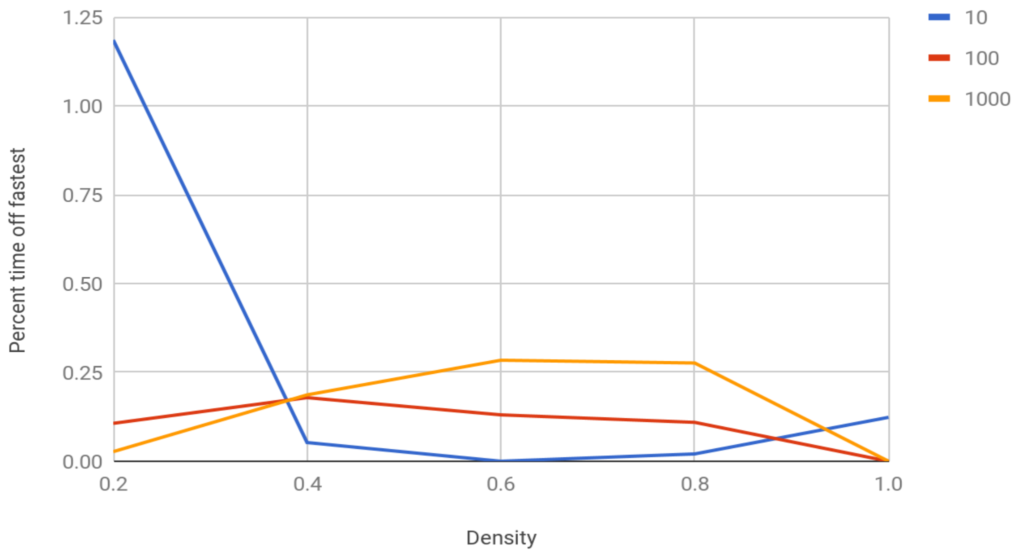


Figure 9: Hill Climb percent time off fastest density for each node count

5.3: Summary

In this chapter, we discussed the correctness and performance tests we conducted on the base implementation of our program and algorithms. We demonstrated how the optimizations we made to our algorithms affected the total backedge weights of their generated rankings and runtime, and provided explanations as to why different algorithms performed better or worse on certain graphs. Finally, we explored the relation of input graph properties, such as number of nodes and density, to the performance of each of our algorithms.

Chapter 6: Edge Weights

After initially experimenting with point differential as edge weights within graphs of sports data, we explored further factors to consider in edge weights to represent more information and create a more accurate ranking. This chapter details the considerations that we made when utilizing edge weights to represent sports data. Each element of our design, implementation, and testing methodology is detailed in following the sections.

6.1: Edge Weight Design

As discussed in previous sections, determining the proper edge weight for each game in our sports data is paramount to accurately capturing the data and producing correct rankings. Aside from reading and condensing sports data into edge weights, we also designed three modifiers for the edge weights that were used to modify the significance of edges based upon multiple factors: linear decay, score normalization, and strength of schedule. We explore these three modifiers in the following sections.

6.1.1: Linear Decay

Before we explored how to weigh our edges, our program did not account for any metric of recency of game when calculating edge weights. We suspected that external rankings applied the recency of game as some form of decay, where more recent games were given higher value than older games, so we explored different approaches to incorporate this in our rankings. We discovered two solutions: linear decay, where each prior game is weighed lower values by a consistent factor compared to the next most-recent game; and exponential decay, where two coefficients determine how highly to weigh a prior game. We expect that external rankings utilize exponential decay, where the graph of the two factors resembles a convex curve, decaying a game mildly if it was recent but decaying it much more heavily if older. After discussion, we decided to pursue linear decay because we felt we could develop a suitable approximation without having to handle the additional complexity of exponential decay.

6.1.2: Score Normalization (Beta Methodology)

Using the point differential as the weight for each edge within our graph presented drawbacks when modeling our sports data. The first drawback was how to handle equivalent point differentials. For example, we initially felt that a game with a final score of 3-0 should not be weighed the same as a game with a final score of 6-3. Even though the point differentials are equal, the first game resulted in a shutout of the second team. However, after discussion, we concluded that external rankings likely applied minimal consideration to this, and decided not to apply a base weight factor. The second drawback was seen in games where a team ran up the score. For example, a better team could continue scoring points within a game to increase their point differential, thus resulting in a much larger edge weight for that game.

Normalizing the edge weights within the graph proved to be a viable solution to the above drawbacks, which we completed through a process known as Beta Normalization. **Beta Normalization** divides each game's point differentials into ranges with an **interval** of one score for that sport. An example of the Beta normalization process and the points within each interval for American Football are displayed in Table 18. The first key point of this normalization process is that the highest Beta range contains no upper bound for score. Our discussion concluded that, after a four score point differential for our test football graphs, any additional points scored were likely meaningless in assigning the edge weights. The second key point for this process was the actual assignment of intervals. For example, we figured that a touchdown, or one score, can be significant enough to modify which range a game is placed into for football games.

Table 18: Beta Interval Points for American Football

Beta Interval 1	Beta Interval 2	Beta Interval 3	Beta Interval 4	Beta Interval 5
1-7 points	8-14 points	15-21 points	22-28 points	29+ points

6.1.3: Quartile of Losing Team (Strength of Schedule)

Reviewing the factors of ranking algorithms discussed in Section 2.6, we noted that we still had not introduced an explicit solution to the strength of schedule factor. The weighted Minimum Feedback Arc set problem implicitly factors in strength of schedule because better teams are more likely to have more forward edges that should cancel out any backedges from an upset match. Utilizing a graph-based approach made it more difficult to quantify strength of schedule compared to a ranking system such as Elo, so we needed to develop our own factor to account for it. The result of discussion was a quartile factor system, where we considered the relative placement of the losing team in the ordering of all teams. To apply this, we designed a multiplier for the quartile of the losing team to be applied to any edge.

To compute the quartile factor, the teams in any given ordering were divided into quartiles during evaluation, based on what rank each team was placed in. The quartile of the losing team was used as the basis to choose the quartile multiplier, which ranged from zero to one. This quartile multiplier was designed to be used in the evaluation process for a given ordering of teams, which changed depending on the specific ordering used with it. The quartile multiplier would be a modifier to any edge weights counted as backedges toward the total backedge weight. A value of zero as a multiplier would multiply the edge weight by zero, essentially removing the significance of the edge. For that reason, quartile multipliers have an inclusive range from 0.25 to 1. A value of one as a multiplier would multiply the edge weight by one, not diminishing the significance of the edge at all. The quartile multiplier was designed to diminish the reward of winning against a low ranked team and reward victories against strongly ranked teams.

Along with decay and normalization, the quartile multiplier in evaluation became the third modifier for edge weights.

6.1.4: Alpha/Beta Methodology

Reflection on different approaches to our ranking factors of decay, strength of schedule, and point differential influenced the redesign of our edge weight algorithm to apply these changes. The resulting formula is referred to as Alpha/Beta in the remainder of this report. Our algorithm applied the product of variables Alpha and Beta as each edge weight during processing of the adjacency matrix, where Alpha and Beta were each values between 0 and 1. **Alpha** represented the linear decay factor for the game, where a value of 0 disregarded the oldest games in a graph altogether and a value of 1 resulted in no decay. **Beta** represented the edge weights applied to each interval before decay, where a value of 0 resulted in games in the first interval receiving zero edge weight, while a value of 1 resulted in equivalent weight factors being applied to the point differentials, regardless of interval.

Our implementation of this Alpha/Beta system was designed to be adaptable for future testing needs. Both Alpha and Beta values were applied from the configuration file where they could be modified with ease. Additionally, the number of Beta intervals and Beta step size were also stored in the configuration file. The step size varies by sport where the point value of a score differs, so this value needed to be flexible. If we wanted to adjust the number of intervals for testing purposes, that functionality was also present.

6.2: Edge Weight Implementation

The implementation of edge weight normalization, also known as score normalization, in the program had some challenges. As described in Section 6.1.2, edge weight normalization is the process of normalizing the point differential of a game into a small number of categories. The actual edge weight values calculated from the point differential are fixed by a Beta value and are decimal values. The first complication presented itself when determining how to store decimal values for the normalized edge weights alongside the original edge matrix. The original adjacency matrix was implemented as a two-dimensional array of integers up to a maximum size of 250 by 250. At the time of the implementation of score normalization, all of the functions in the program expected the edges to be stored as integers; therefore, substantial refactoring would be necessary to introduce edges as double precision floating point numbers.

We decided to implement the adjacency matrix of normalized edges as a two-dimensional array of doubles, allocated to the exact number of cells required in order to reduce memory usage. In order to allocate the necessary amount of memory for the normalized adjacency matrix when the number of teams in the input file was unknown, we had to develop a workaround. First, the normalized adjacency matrix was allocated to the maximum size of the non-normalized adjacency matrix of 250 by 250. The normalized matrix was then filled simultaneously with the non-normalized adjacency matrix in the reading process of the input file. As each edge was read, the score differential had the normalized edge weight calculated depending on the Beta value, the size of Beta steps, and the number of Beta steps. When the reading process concludes, the total number of nodes and the maximum memory needed for an adjacency matrix is known. Finally, the oversized normalized adjacency matrix is reallocated into a smaller size and the values are copied over.

The decision to have an optional normalized adjacency matrix required that all ranking algorithms be duplicated as well to utilize the normalized matrix. This was necessary because there were so many accesses to the adjacency matrix in each algorithm that it was prohibitively difficult to implement variable access to the normalized adjacency matrix. Although this made the logic simpler within functions, this decision greatly increased the time required for code maintenance.

6.2.1: Alpha/Beta Heatmaps

Upon implementing the Alpha/Beta functionality within our program, we applied it to our evaluation of external rankings. In our discussion, we saw testing the performance of rankings with different Alpha and Beta values as a potential way to reverse-engineer external rankings and better understand what factors influenced them. We could evaluate external rankings several times with different combinations of Alpha and Beta and compare the resulting total backedge weights, where lower total backedge weights would indicate a better response to the Alpha and Beta values used.

To better compare our resulting total backedge weights, we added functionality to our program to generate a heatmap. For the purposes of this project, a **heatmap** is a matrix of total backedge weight values for a given ranking where the cells are colored in varying gradients based on their value, with the horizontal axis indicating changes in Alpha and the vertical axis indicating changes in Beta. Heatmaps allowed us to observe the total backedge weights at a glance to quickly identify any trends in increases or decreases of backedge weights with changes of Alpha and Beta.

We began implementation of heatmap functionality by replicating our external rank evaluation process. Instead of simply evaluating the total backedge weight of the ranking, we modified the external evaluation to disregard any Alpha and Beta values provided by the configuration file because the adjacency matrix for normalized edges is regenerated with multiple permutations of Alpha and Beta values. In the process of generating the new normalized adjacency matrix, the sports data input file is reread. Since normalized edges compress the information of an edge into a singular value, it is not possible to extract the lost information from the original sports data from the edge. Therefore, the entire input file needed to be reused to generate each new batch of edge weights. The evaluation function was rerun on the ordering with a given Alpha and Beta to evaluate the total backedge weight for that instance.

We chose to explore permutations of Alpha and Beta values ranging from 0 to 1, inclusive, in increments of 0.1. We chose such small increments as a way to increase granularity in testing, given that the external evaluation code had minimal computation time. The total backedge weights of the given external ranking would change as the values of Alpha and Beta altered how edges in the input file were weighed. These total backedge weights were all then normalized to their respective total edge weight in the graph at each instance of a given Alpha and Beta value pair. For example, if the total backedge weight of the external ranking was evaluated to 20.0 and the total edge weight in the entire graph, regardless of direction of the edges, was 80.0, the normalized total edge weight would be 0.25 as the backedge weight of the ranking was one fourth of the total edge weight in the graph.

All of the normalized total backedge weights of each permutation of Alpha and Beta for the ordering were gathered and displayed in a table that could be converted to a heatmap. Our heatmap output was chosen to be in CSV (comma separated values) format. CSVs are flexible for our purposes: they can be viewed in Microsoft Excel and most text editors. Additionally, when opened in Excel, they were presented in a clean table view, where it was simple to view the values for individual columns compared to manually printing out an ASCII table. Finally, in order to add color to the heatmap, we utilized Microsoft Excel's workbook functionality, as CSVs do not support formatting.

6.3: Edge Weight Results

In this section, we apply our Alpha/Beta edge weight and heatmap functionalities towards our algorithms and external rankings. These tests allow us to better understand the factors that external rankings account for by adjusting the Alpha and Beta values to give more or less priority to these factors. From this, we can determine using the total backedge weight from different Alpha/Beta combinations which rankings favor which values. The first tests we conducted with our heatmaps were to determine how our quartile and home-field advantage modifiers impact our rankings and external rankings. For these test cases, the home-field advantage modifier added three points, or one field goal, to a victory by an away team in football games. Afterwards, we tested our lowest backedge weight external rankings for the 2016-17 NFL dataset, 2016-17 CFB dataset, 2014-15 NHL dataset, and 2015 MLB dataset to better understand the factors they consider. Heatmaps from additional rankings for these datasets are included and discussed in Appendix B.

The first set of tests we conducted were Alpha/Beta heatmap tests comparing total normalized backedge weight values for different Alpha/Beta values while toggling our quartile and home-field advantage factors in our edge weight algorithms. As explained in Section 6.2.1, our heatmaps are adjacency matrices containing the normalized total backedge weights of each algorithmic or external ranking on a given dataset. Our heatmaps present changes in Alpha on the vertical axis from 0 to 1 in 0.1 increments and present changes in Beta on the horizontal axis from 0 to 1 in 0.1 increments. These heatmaps are colored on a red to green gradient, where shades of green represent smaller total backedge weights and shades of red represent larger total backedge weights.

	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
0	**<u>0.206206</u>**	0.214998	0.220393	0.224041	0.226673	0.22866	0.230215	0.231464	0.232489	0.233346	0.234073
0.1	0.21071	0.218929	0.223951	0.227337	0.229774	0.231613	0.233049	0.234202	0.235149	0.235939	0.236609
0.2	0.214528	0.222245	0.226942	0.230102	0.232373	0.234084	0.23542	0.236491	0.237369	0.238103	0.238724
0.3	0.217806	0.22508	0.229493	0.232456	0.234582	0.236183	0.237431	0.238432	0.239252	0.239936	0.240516
0.4	0.220651	0.227531	0.231693	0.234483	0.236483	0.237987	0.239159	0.240099	0.240868	0.24151	0.242054
0.5	0.223143	0.229671	0.233611	0.236248	0.238136	0.239555	0.240661	0.241546	0.242271	0.242875	0.243387
0.6	0.225345	0.231556	0.235298	0.237798	0.239587	0.240931	0.241977	0.242814	0.243499	0.244071	0.244555
0.7	0.227303	0.233229	0.236792	0.239171	0.240871	0.242147	0.24314	0.243934	0.244585	0.245127	0.245586
0.8	0.229057	0.234724	0.238126	0.240394	0.242014	0.24323	0.244175	0.244931	0.24555	0.246066	0.246502
0.9	0.230637	0.236068	0.239323	0.241491	0.24304	0.2442	0.245103	0.245824	0.246415	0.246907	0.247323
1	0.232068	0.237283	0.240404	0.242482	0.243964	0.245075	0.245938	0.246629	0.247194	0.247664	0.248062

Figure 10: 2017-18 NFL heatmap (Sports Illustrated, Quartiles Off)

	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
0	**<u>0.090165</u>**	0.109563	0.122772	0.132347	0.139606	0.145299	0.149882	0.153653	0.156809	0.159489	0.161793
0.1	0.090861	0.1104	0.123694	0.133323	0.14062	0.146341	0.150946	0.154733	0.157902	0.160593	0.162907
0.2	0.091445	0.111102	0.124465	0.134141	0.14147	0.147214	0.151836	0.155637	0.158817	0.161517	0.163838
0.3	0.091942	0.111699	0.125121	0.134835	0.142191	0.147954	0.152592	0.156404	0.159593	0.162301	0.164628
0.4	0.092370	0.112213	0.125686	0.135433	0.142811	0.148591	0.153241	0.157063	0.16026	0.162974	0.165307
0.5	0.092743	0.11266	0.126177	0.135952	0.143351	0.149145	0.153806	0.157636	0.16084	0.163559	0.165896
0.6	0.093070	0.113052	0.126608	0.136408	0.143823	0.14963	0.154301	0.158138	0.161348	0.164072	0.166413
0.7	0.093360	0.113399	0.126989	0.136811	0.144242	0.150059	0.154738	0.158582	0.161797	0.164525	0.16687
0.8	0.093618	0.113709	0.127328	0.13717	0.144614	0.150442	0.155128	0.158978	0.162197	0.164929	0.167276
0.9	0.093850	0.113986	0.127633	0.137492	0.144948	0.150784	0.155477	0.159332	0.162555	0.16529	0.167641
1	0.094059	0.114236	0.127907	0.137782	0.145249	0.151093	0.155791	0.159651	0.162878	0.165616	0.167969

Figure 11: 2017-18 NFL heatmap (Sports Illustrated, Quartiles On)

The heatmaps in Figures 10 and 11 show our Sports Illustrated ranking total backedge weights with and without our quartile factor applied. In Figure 10, where the quartile factor is disabled, increases in the value of Alpha and increases in the value of Beta contribute roughly equivalently towards the total backedge weight, which implies that Sports Illustrated applied equivalent preference towards recency of match and point differential. However, with quartiles applied as shown in Figure 11, increases in the value of Alpha are far more significant in increasing the total backedge weight than increases in Beta. We expect that this is because the quartile factor reduces the edge weight before decay substantially for most edges, such that applying less of a decay factor increases the total graph weight enough to result in increased total backedge weight.

	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
0	**0.222222**	0.228702	0.232678	0.235367	0.237306	0.238771	0.239917	0.240837	0.241593	0.242225	0.242761
0.1	0.231285	0.236792	0.240157	0.242426	0.244059	0.245291	0.246254	0.247027	0.24766	0.24819	0.248639
0.2	0.238968	0.243617	0.246446	0.24835	0.249718	0.250749	0.251553	0.252198	0.252728	0.253169	0.253544
0.3	0.245564	0.249451	0.251809	0.253392	0.254528	0.255383	0.25605	0.256585	0.257023	0.257389	0.257699
0.4	0.251289	0.254495	0.256435	0.257735	0.258667	0.259368	0.259914	0.260352	0.260711	0.26101	0.261263
0.5	0.256304	0.2589	0.260467	0.261516	0.262267	0.262831	0.263271	0.263623	0.263911	0.264151	0.264355
0.6	0.260734	0.26278	0.264013	0.264836	0.265426	0.265868	0.266213	0.266489	0.266714	0.266903	0.267062
0.7	0.264675	0.266224	0.267154	0.267776	0.26822	0.268554	0.268813	0.269021	0.269191	0.269332	0.269452
0.8	0.268205	0.2693	0.269958	0.270397	0.27071	0.270945	0.271128	0.271274	0.271394	0.271493	0.271578
0.9	0.271383	0.272066	0.272475	0.272748	0.272942	0.273088	0.273201	0.273292	0.273366	0.273428	0.273481
1	0.274262	0.274566	0.274747	0.274869	0.274955	0.27502	0.27507	0.27511	0.275143	0.275171	0.275194

Figure 12: 2016-17 NFL heatmap (Sports Illustrated, Quartiles Off, Home-Field Advantage Off)

	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
0	**0.206622**	0.216686	0.223419	0.228239	0.231861	0.234682	0.236942	0.238792	0.240334	0.24164	0.242761
0.1	0.217273	0.226058	0.231914	0.236096	0.239232	0.241671	0.243622	0.245219	0.246549	0.247674	0.248639
0.2	0.226292	0.233962	0.239058	0.24269	0.245409	0.247522	0.24921	0.25059	0.251739	0.252711	0.253544
0.3	0.234028	0.240718	0.24515	0.248304	0.250662	0.252491	0.253952	0.255146	0.25614	0.256979	0.257699
0.4	0.240736	0.246558	0.250407	0.25314	0.255182	0.256765	0.258028	0.259059	0.259917	0.260642	0.261263
0.5	0.246609	0.251658	0.254989	0.257351	0.259114	0.260479	0.261568	0.262457	0.263196	0.26382	0.264355
0.6	0.251793	0.256149	0.259018	0.26105	0.262564	0.263737	0.264672	0.265434	0.266068	0.266604	0.267062
0.7	0.256403	0.260135	0.262588	0.264324	0.265617	0.266618	0.267415	0.268065	0.268606	0.269062	0.269452
0.8	0.260529	0.263695	0.265774	0.267244	0.268338	0.269184	0.269857	0.270407	0.270863	0.271248	0.271578
0.9	0.264244	0.266896	0.268635	0.269863	0.270777	0.271483	0.272046	0.272504	0.272885	0.273206	0.273481
1	0.267606	0.269788	0.271218	0.272227	0.272977	0.273556	0.274017	0.274393	0.274705	0.274969	0.275194

Figure 13: 2016-17 NFL heatmap (Sports Illustrated, Quartiles Off, Home-Field Advantage On)

Additionally, as shown in Figures 12 and 13, we tested the impacts of our home-field advantage modifier on Sports Illustrated's power ranking for the 2016-17 NFL dataset. Overall, this modifier does not substantially impact the trends in Alpha and Beta values for the total backedge weight of the ranking. The best Alpha and Beta values are identical for both at (0,0). However, the total backedge weight with home-field advantage enabled decreases slightly, which we expect is because the forward edge weight increases more with the home-field advantage factor applied than the backedge weight does. Additionally, enabling quartiles reduces the significance by which increases in Beta values with a large Alpha value increase the total backedge weight, which we suspect is because considering home-field advantage modifies Beta's point differential, potentially making the games in a season closer together. Thus, changing the decay through Alpha would modify the edge weights more significantly.

The second set of tests we conducted with our Alpha/Beta heatmaps demonstrate the trends in total backedge weight based on differing Alpha and Beta values for each of our rankings in the 2016-17 NFL dataset, 2016-17 CFB dataset, 2014-2015 NHL dataset, and 2015 MLB dataset. The heatmap in Figure 14 shows how the NFL.com power ranking for the 2016-17 NFL season dataset responds to changes in Alpha and Beta values. Heatmaps and analysis for additional rankings from these seasons are included in Appendix B. The heatmaps discussed in this section were chosen because their best Alpha/Beta combination resulted in the lowest total backedge weights compared to other two power rankings.

	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
0	**<u>0.151276</u>**	0.15178	0.152088	0.152297	0.152448	0.152562	0.152651	0.152722	0.152781	0.15283	0.152872
0.1	0.152303	0.152775	0.153064	0.153259	0.153399	0.153504	0.153587	0.153653	0.153707	0.153753	0.153791
0.2	0.153174	0.153615	0.153884	0.154065	0.154195	0.154293	0.154369	0.154431	0.154481	0.154523	0.154559
0.3	0.153921	0.154334	0.154584	0.154752	0.154872	0.154963	0.155034	0.15509	0.155137	0.155176	0.155209
0.4	0.15457	0.154954	0.155187	0.155343	0.155455	0.155539	0.155605	0.155657	0.1557	0.155736	0.155766
0.5	0.155138	0.155497	0.155713	0.155858	0.155962	0.15604	0.1561	0.156149	0.156189	0.156222	0.15625
0.6	0.15564	0.155974	0.156176	0.15631	0.156406	0.156479	0.156535	0.15658	0.156617	0.156647	0.156674
0.7	0.156087	0.156398	0.156585	0.15671	0.1568	0.156867	0.156919	0.156961	0.156995	0.157023	0.157047
0.8	0.156487	0.156777	0.156951	0.157067	0.15715	0.157212	0.157261	0.1573	0.157331	0.157358	0.15738
0.9	0.156847	0.157117	0.157279	0.157387	0.157464	0.157522	0.157567	0.157603	0.157632	0.157657	0.157678
1.0	0.157173	0.157425	0.157576	0.157676	0.157748	0.157801	0.157843	0.157876	0.157904	0.157927	0.157946

Figure 14: 2016-17 NFL heatmap (NFL.com, Quartiles On)

From the heatmap in Figure 14, we can infer several factors about NFL.com's power ranking. The most apparent trend in this heatmap is how total backedge weight significantly increases as the value of Alpha increases. The total backedge weight does increase with an increase in Beta, as expected, but not to the magnitude of changes in Alpha. We suspect that this is because the ranking from NFL.com has the teams placed such that games or edges which would have less value with smaller Alpha values were backedges. Specifically, according to the heatmap and our total backedge weight metric, the NFL.com power ranking considers the recency of game significant. Additionally, the small increases in total backedge weight from increases in Beta imply that the NFL.com power ranking does not apply much weight towards the point differential within a game. Thus, teams with higher point differential recent games are most likely to be ranked highest in NFL.com power rankings.

	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
0	**<u>0.003842</u>**	0.005011	0.005936	0.006687	0.007308	0.007830	0.008275	0.008660	0.008994	0.009289	0.009550
0.1	0.004496	0.005531	0.006349	0.007013	0.007563	0.008025	0.008419	0.008759	0.009056	0.009316	0.009548
0.2	0.004999	0.005930	0.006666	0.007264	0.007759	0.008175	0.008530	0.008836	0.009103	0.009338	0.009546
0.3	0.005398	0.006246	0.006918	0.007463	0.007914	0.008293	0.008617	0.008897	0.009140	0.009354	0.009544
0.4	0.005721	0.006503	0.007122	0.007624	0.008040	0.008390	0.008688	0.008946	0.009170	0.009368	0.009543
0.5	0.005988	0.006715	0.007291	0.007758	0.008144	0.008470	0.008747	0.008987	0.009196	0.009379	0.009542
0.6	0.006214	0.006894	0.007433	0.007870	0.008232	0.008537	0.008797	0.009021	0.009217	0.009389	0.009541
0.7	0.006406	0.007047	0.007554	0.007966	0.008307	0.008594	0.008839	0.009051	0.009235	0.009397	0.009541
0.8	0.006572	0.007178	0.007659	0.008049	0.008372	0.008644	0.008876	0.009076	0.009250	0.009404	0.009540
0.9	0.006716	0.007293	0.007750	0.008121	0.008429	0.008687	0.008908	0.009098	0.009264	0.009410	0.009539
1	0.006844	0.007394	0.007831	0.008185	0.008478	0.008725	0.008936	0.009117	0.009276	0.009415	0.009539

Figure 15: 2016-17 CFB heatmap (ESPN, Quartiles On)

Figure 15 shows the Alpha/Beta heatmap for ESPN's power ranking for the 2016-17 NCAA College Football (CFB) season with Division I teams. In all NCAA College Football heatmaps on external rankings, the total backedge weight values are smaller compared to other sports because these external rankings only consider the top-25 of 128 teams, so many of the edge weights for lower ranked teams are not considered.

As shown in the heatmap in Figure 15, small Alpha and Beta values return low total backedge weights, where the lowest appears with an Alpha/Beta of (0,0). In general, this means that ESPN's power ranking favors recent wins and high point differentials when ordering teams. However, increases in the Beta value result in much greater total backedge weight than increases in Alpha. We suspect that this is because backedges of the ranking are more likely to be close games than older games in the season. Additionally, increases in Alpha result in less total backedge weight increase when Beta is high compared to when Beta is low. We hypothesize that this pattern occurs because backedges have their edge weights minimized much less with Alpha than Beta, but with a higher Beta, the total graph weight is much larger, resulting in smaller impacts from increases in Alpha.

	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
0	**<u>0.261899</u>**	0.262495	0.262992	0.263412	0.263772	0.264084	0.264356	0.264597	0.264811	0.265002	0.265174
0.1	0.262867	0.263588	0.264188	0.264695	0.265129	0.265505	0.265834	0.266123	0.266381	0.266611	0.266818
0.2	0.26368	0.264505	0.265191	0.265771	0.266266	0.266695	0.26707	0.267401	0.267694	0.267957	0.268193
0.3	0.264373	0.265286	0.266045	0.266685	0.267233	0.267707	0.268121	0.268486	0.26881	0.269099	0.269359
0.4	0.264971	0.265959	0.26678	0.267473	0.268065	0.268577	0.269024	0.269418	0.269768	0.27008	0.270361
0.5	0.265491	0.266545	0.26742	0.268158	0.268788	0.269333	0.269809	0.270229	0.2706	0.270933	0.271231
0.6	0.265949	0.26706	0.267982	0.268759	0.269423	0.269997	0.270498	0.270939	0.27133	0.27168	0.271994
0.7	0.266354	0.267516	0.268479	0.269291	0.269985	0.270584	0.271107	0.271567	0.271976	0.27234	0.272668
0.8	0.266716	0.267923	0.268923	0.269766	0.270485	0.271107	0.271649	0.272127	0.27255	0.272928	0.273268
0.9	0.26704	0.268287	0.269321	0.270191	0.270934	0.271575	0.272135	0.272628	0.273065	0.273455	0.273806
1	0.267333	0.268616	0.269679	0.270574	0.271338	0.271998	0.272573	0.27308	0.273529	0.27393	0.27429

Figure 16: 2014-15 NHL heatmap (NHL.com, Quartiles On)

Figure 16 shows the heatmap of NHL.com's standings after the 2014-15 NHL season. In this heatmap, the lowest total backedge weight occurs when Alpha and Beta are both 0, but still retain low total backedge weight with small values for both. Increases in Alpha and Beta appear to increase the total backedge weight similarly, with more weight being added for increases in Alpha. This may be because backedges for this ranking are more likely to be earlier games in the season, which have more weight when no decay is applied, than point differential. However, because of the increases in both, we can state that both recency of game and point differential are considered within NHL.com's standings, which appears counterintuitive towards the basis of standings instead of rankings. This shows that total backedge weight may not be the best metric for determining the optimal Alpha and Beta configuration for an external ranking.

	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
0	**<u>0.221553</u>**	0.227557	0.231426	0.234128	0.23612	0.237651	0.238864	0.239848	0.240663	0.241349	0.241934
0.1	0.221997	0.227673	0.231332	0.233886	0.23577	0.237217	0.238364	0.239295	0.240065	0.240714	0.241267
0.2	0.222368	0.227771	0.231252	0.233683	0.235476	0.236854	0.237945	0.23883	0.239564	0.240181	0.240708
0.3	0.222684	0.227853	0.231185	0.233511	0.235226	0.236544	0.237588	0.238436	0.239138	0.239728	0.240232
0.4	0.222956	0.227925	0.231127	0.233362	0.235011	0.236278	0.237282	0.238096	0.238771	0.239338	0.239823
0.5	0.223192	0.227986	0.231076	0.233233	0.234824	0.236047	0.237015	0.237801	0.238452	0.238999	0.239467
0.6	0.2234	0.228041	0.231032	0.23312	0.23466	0.235843	0.236781	0.237542	0.238172	0.238702	0.239154
0.7	0.223583	0.228089	0.230993	0.23302	0.234515	0.235664	0.236574	0.237312	0.237924	0.238439	0.238878
0.8	0.223747	0.228132	0.230958	0.23293	0.234386	0.235504	0.236389	0.237108	0.237703	0.238204	0.238631
0.9	0.223894	0.22817	0.230926	0.23285	0.23427	0.23536	0.236224	0.236925	0.237505	0.237994	0.238411
1	0.224026	0.228205	0.230898	0.232778	0.234165	0.23523	0.236074	0.236759	0.237327	0.237804	0.238211

Figure 17: 2015 MLB heatmap (baseball-reference.com, Quartiles On)

In Figure 17, the heatmap for baseball-reference.com’s standings of the 2015 MLB season is displayed. The lowest total backedge weight in this heatmap occurs when Alpha and Beta are both 0; however, low total backedge weights occur with small Alpha and Beta values, which indicates that this ordering favors the most recent, highest point differential games. An unusual trend with this heatmap is the large increase in total backedge weights with a small Alpha value and a large Beta value when compared to the minimal change in total backedge weight with a large Alpha value and a small Beta value. We suspect that this may be due to recent games with small point differentials being backedges for this ordering, where the overall graph weight is small from discounting early games. With a larger Alpha, when earlier games have more weight, the total backedge weight is slightly lower, likely because the forward edges for this ranking have more weight and skew the normalized weight.

6.3.1: Discussion on Alpha/Beta Heatmap Testing

We notice from these tests that, regardless of sport, the Alpha and Beta values for our external rankings which frequently yielded the lowest total backedge weight are (0,0). Despite the frequency of these results, we are concerned that there might be a flaw in the Alpha/Beta methodology which would favor these values. We suspect that the Alpha/Beta process favors minimizing the total backedge weight for a ranking rather than identifying the factors which best match an external ranking.

We reason that 0 for the Alpha value is favored because it allows the entire earliest set of games to be ignored. As detailed in Section 6.1.1, the oldest edges or games in any set of sports data are completely removed from the graph when the Alpha value is 0. In a weekly sport such as NFL football, this results in the removal of the entire first week of games. This is mitigated somewhat in sports that use dates, such as Baseball, since it would only completely remove games from the first day. An Alpha value of 0 also removes the most amount of edge weight from a graph of all Alpha values, as the oldest games have their values for edge weight decayed as much as possible. Overall, an Alpha value of 0 favors reducing the total normalized backedge weight because games that could have critical backedges would be weighed low enough that their impact is reduced on the total backedge weight of an ordering.

We reason that a Beta value of 0 is favored because it causes the highest reduction in weight of close games. We speculate that many of the upset victories that are added to the total backedge weight can be attributed to close games, with the point differential usually within one score. Moreover, a Beta value of 0 is favored over another low value such as 0.1 because a Beta value of 0 completely removes the closest games with a low point differential. Removing controversial or close games naturally causes the total backedge weight to be reduced, and reduces the weight of backedges which are not in the highest Beta interval.

Ultimately, our heuristic is measuring the ability for each Alpha/Beta combination to reduce the total normalized edge weight. As explained above, the properties of a combination of (0,0) greatly favor our heuristic.

6.4: Summary

This chapter introduced factors and methodologies we considered when determining how to weigh the edges in our sports graphs. We considered linear decay for recency of game, score normalization for close games and running up the score, and the quartile of the losing team for strength of schedule; three factors that were likely important during the development of external rankings. Each of these factors is explained alongside the rationale of how they were implemented and represented in our program. Finally, we explored the application of these factors to rankings our algorithms generated and the possible flaw behind the favored Alpha/Beta values of (0,0).

Chapter 7: Top-N Truncated Rankings

In this chapter, we discuss our methodologies for handling truncated rankings, or rankings of the “top n” number of teams. We open by exploring the methods of rank snipping, which maintain the score of the full ranking while returning only the top n teams. We then introduce formal rank truncation and the N+1 node, alongside the justifications behind truncating internally or externally. We then discuss how we implemented these forms of truncation alongside considerations to reduce complications with existing functionality. Finally, we provide examples of truncation methods, demonstrating their practical applications.

7.1: Top-N Truncated Ranking Design

The need for some form of truncation arose out of the prevalence of rankings that only included a subset of the total number of teams in competition. There are a plethora of sports rankings that only include the top-5, top-10, or top-25 teams. In order to create comparable rankings generated by our algorithms, we designed a system to truncate rankings. We reasoned that there is much less interest between the relative rankings of teams ranked 120 and 121 compared to the relative rankings of teams in the top five. There is a much greater focus on the relative rankings of teams with more importance to the audience of the rankings, teams which are usually ranked the highest.

7.1.1: Rank Snipping for Top-N

Rank snipping is the simplest form of truncated rank generation. The general process for rank snipping has two main parts. The first part of this method of truncation actually did not involve any truncation at all; rather, the first step was to generate a ranking of the complete set of input teams as if we were not truncating at all. Then, the complete ranking would be “snipped” so only the top-n teams would remain. The result of the rank snipping process would be a ranking of the top-n teams as generated by our project using all of the available information given as sports data to the program. We figured that this form of truncation was valid because external rankings that only published top-n rankings likely used the same full season of data that we did when generating our top-n rankings after snipping.

7.1.2: Internal Truncation and the N+1 Node

Internal truncation is the process of only working with a subset of the total number of nodes on the graph. Applied to the domain of sports rankings, there would be fewer teams considered as eligible to be in the ranking. Internal truncation was designed to reduce the computation necessary to develop a ranking. With fewer nodes in the graph, the generation of permutations would become easier for the program. Internal truncation was rationalized with the reasoning that we only care about computing the ranking of a subset of teams if we only want a subset ordering as an end result. Internal truncation was also poised as a tool to allow brute force computation of certain subsets of real data where there were normally a prohibitively large number of teams in the set.

However, there was a fundamental problem with not considering all of the nodes in the input data: there is a loss of information that would occur as teams and games from outside the top-n teams would be discarded. Furthermore, internal truncation begs the question of how to deal with games that were played between teams of the top-n subset and teams outside the subset. Dropping the games between teams within the top-n and teams outside the top-n could result in lost information of important wins or losses which would have an impact on the final ranking.

In order to mitigate the loss of information in the truncation process, we developed the concept of the N+1 node. The N+1 node was another node that acted in place of "all other teams" in the internal truncation of the complete set. The N+1 node was generated by the program rather than being parsed from sports data as in the other teams in the input file. To retain the information in the truncation process, all edges from games played between top-n teams and teams excluded from the top-n subset would not be removed, but would point instead to the N+1 node. The N+1 node served to preserve forward edges and backedges between the teams in the top-n so that they would not be lost in the truncation process. Only edges between nodes that were placed within the N+1 node, which were no longer relevant to the top-n ranking, would be removed. The N+1 node was named as such because it corresponded to the collection of nodes that would logically be placed outside of the top-n nodes during the truncation process. The N+1 node workaround allowed internal truncation to be viable in reducing computation necessary to develop rankings without a significant loss in information used to rank the teams in the produced ordering.

7.1.3: Internal Versus Externally Specified Top-N

The final point of consideration in internal truncation was how to decide which teams were eligible to be ranked after truncation. We needed to determine which nodes deserved to be ranked in the top-n. We developed two different methods for determining the top-n nodes which are outlined below.

The first method for determining the top-n subset of teams for internal truncation was to use the internally calculated edge weights. This strategy involved first sorting all nodes by their net edge weight. Though this preordering had no bearing on the final ranking, it was used as the metric for deciding which nodes were eligible and relevant for the top-n nodes.

The second method for determining the top-n subset of teams was to use an external listing of teams to specify the subset of teams to be kept after truncation. The complete set of teams was truncated to match the list of teams present in the external list. This method of truncation was designed primarily to aid in the comparison with external rankings of subsets of the teams within a league. For example, if we wanted to compare our internally truncated ranking of teams from the 128-team set of college football data to a top-25 ranking of college football teams, we could use the external top-n truncation functionality to truncate our set of eligible teams to the exact same teams as the top-25 external ranking. From that point, we could compare the differences between the placements of teams in our top-25 ranking versus the placements of top-25 teams from the external ranking. This method of truncation became particularly useful in creating comparable rankings to external rankings since this system only ordered the exact same teams as the external one.

7.1.4: Conclusions on the Comparison of Truncated Rankings

A ranking that is truncated is much different than a ranking of the whole set of inputs. A complete ranking organizes all entities into an ordering, where teams higher in the ranking are better than teams lower in the ranking, based upon some heuristic. Though complete rankings are relatively straightforward in their significance, truncated rankings are notably more complicated.

We concluded that there are two main points of interest and significance that are integral to top-n truncated rankings: the relative orderings between the top-n teams, and the assertion that the subset of top-n teams are better than the remainder of the teams in the complete set. In the process of comparing ranking that were of the top-n variety, such as top-10 and top-25, these two factors needed be considered. In comparing two rankings of a subset of teams, either of these points could be used as the metric by which the comparison was conducted. Depending on which of these two main points of interest was more valuable or of greater importance, the method of truncation needed to be adapted to suit it.

We decided that using an externally specified top-n was useful for comparison of the first point of interest: whether the specific ordering of the top-n teams was correct. Since this method for truncation took an external ranking's specification for which nodes deserve to be in the top-n, the rankings generated by our algorithms were solely focused on the relative ordering between the top-n teams of each truncated ranking. Therefore, if we were to compare two truncated rankings for the relative differences between the placements of the top-n teams, truncation using the nodes specified by the external ranking would be most sensible. We found that the drawback of this approach was that it could not be used to explore the second point above, as the ranking generated could not be used to assert that the teams shown in the top-n ranking were indeed the top-n of the whole set.

In order to maintain the comparison of both points above, we decided to use rank snipping to generate top-n rankings for the purposes of comparing multiple truncated orderings, whether internal or external. Though rank snipping is not as direct a comparison in regards to the relative ordering of teams between the two top-n teams from the first point, we determined that rank snipping for a top-n ranking weighed both of the above points to a satisfactory degree. The use of rank snipping was supported because it was similar to the methodology for creating a top-n ranking from a non-truncated source of sports data. We used the rationale that a top-n ranking output by the program should be created as the top n teams of the best ranking that we could create by not truncating and leaving all nodes and edges in the graph. The best possible ranking would necessarily utilize all of the information available to the program, and from that, a top-n could be extracted. This truncated output ranking could then be read into the program and handled as we would handle any other external ranking. We reasoned that external top-n rankings were created and truncated through a methodology that could not be inferred from the outside. We decided to mimic this "black box" approach. Therefore, using rank snipping to create a ranking as a "black box” ranking seemed to be the most comparable approach when creating truncated rankings. Given the “black box” approach, we simply needed to create what we thought would be the best ranking possible, and truncate it afterwards to match via rank snipping. If it was necessary to compare exclusively on the first point above, we could utilize internal truncation on an external ranking for the basis of determining which teams were allowed into the top-n ranking. However, permitting the computation time requirements, it was more accurate of a comparison to use truncated rankings through rank snipping than through internal truncation.

7.2: Top-N Truncated Ranking Implementation/N+1 Node

The implementation of the internal top-n truncation design had to handle several difficult issues. The first major challenge was how to deal with the truncation of the adjacency matrix when the list of acceptable nodes was not guaranteed to be known until runtime. The second major hurdle of the internal top-n truncation was how to create and maintain the N+1 node. Finding an efficient solution would not only require implementation of the designed features, but would also be standalone enough to not require extensive rewrites of existing functionality. By the time of the implementation of top-n truncation, there were many features already implemented in the program, so compatibility with all existing features of the program was the most significant implementation goal.

7.2.1: Truncation of the Adjacency Matrix

Truncation of the adjacency matrix was a difficult process to implement because the nodes allowed in the top-n were not specified before runtime. Whether by external list or parsing of the net total weights on the nodes in the graph, the nodes to include in the internal top-n subset would be known only after the adjacency matrix had been initially configured. Without significant arduous rewrites to the complex input file read process, the truncation had to occur after the normalized and non-normalized adjacency matrices had already been configured. The implementation of internal truncation could not simply create another adjacency matrix because it would involve greater performance overhead, require all new evaluation functions, and all existing algorithms would need to be adapted to handle the new evaluation functions and pull from the new matrix. Additionally, all new features added to the program would require additional work to compensate for use of the additional matrix. Therefore, another option had to be considered.

The approach for the implementation of internal truncation turned to modification of the existing adjacency matrices. Despite the implementation challenge it presented, modification of the existing adjacency matrices would not require extensive rewrites of existing systems including evaluation functions and ranking algorithms. The functions in the program were already designed to handle an arbitrarily sized adjacency matrix, so modification of the size and contents of the existing adjacency matrices seemed to be the most sensible choice because it allowed all other systems of the program to function with internal truncation without modification.

The modification of the existing adjacency matrices for the internal truncation was a challenge to implement because it needed to be done in such a way that it would not impact functionality in the rest of the program. The modification process had to replicate each adjacency matrix as if it had been input in its truncated state. Once the list of nodes to maintain through the truncation process had been determined, whether through an external listing or from the top-n nodes based upon net weight of each node, both the normalized and non-normalized adjacency matrices could be modified. The first step in the modification process was to create a new version of the node dictionary, called the NodeDict. The NodeDict held the associations between the team names for each node and the corresponding index in each adjacency matrix. The modified NodeDict would contain the associations for just the top-n teams with their new indices. Once the new NodeDict was fully configured, the values in each adjacency matrix could be modified as well. A temporary truncation matrix was created to hold the contents of the truncated adjacency matrix during the process of transferring the existing edge weights from each adjacency matrix to the truncated version. Once the transfer of the relevant edge weights for the top-n teams between the main adjacency matrix and the truncated version had completed, then the main adjacency matrix was overwritten with the new values. The truncated version of the NodeDict was also set to overwrite the original NodeDict's values so that the node indices of each adjacency matrix would be accurate to the truncated version. After this process, the program could continue to utilize the adjacency matrix and NodeDict without issue.

7.2.2: Creating and Maintaining the N+1 Node

As explained in Section 7.1.2, the N+1 node was a node that would hold all of the edges between nodes in the top-n subset and nodes outside the top-n subset. The implementation of the N+1 node presented a challenge because it was an artificial node created and maintained entirely by the program. This node was appended to the truncated graph during the adjacency matrix modification process detailed in Section 7.2.1. Weights were transferred to the N+1 node by comparing the original adjacency matrix with the truncated version and adding edges to the N+1 node in the truncated adjacency matrix to and from nodes outside the top-n subset. An entry in the NodeDict was also made for the N+1 node with the proper index in each adjacency matrix. While using internal truncation, the index of the N+1 node was always the last available index in the NodeDict. Instead of a special placeholder, the N+1 node was implemented to be treated just like any other node by using its index, so that it was always known.

Despite the high compatibility, this implementation of the N+1 node did have several drawbacks. The main complication caused by this implementation of the N+1 node was that it needed to be handled in a special way in the testing functionality of the program. Since it was treated as a normal node, rank generation through the ranking algorithms required no extra work to get truncated orderings. However, exceptions had to be created for any functionality that interacted with external orderings. Interfacing with external orderings occurred in the testing modules of the program, including external rank evaluation from Section 4.1.1, Alpha/Beta heatmap testing from Section 6.2.1, rank comparisons from Section 4.5, and truncation by an external list. Since the N+1 node was generated by the program, there was no consistent basis for the inclusion on the N+1 node in external rankings. There were occasions when the N+1 node was needed by the program but not supplied in the external ranking, and vice versa. This implementation of the N+1 node, where it is treated as any other normal node in the graph, added extra programming overhead when dealing with any external rankings. Workarounds and exceptions for the N+1 node had to be made in all existing and future interactions with external ranking files. There were a few other minor complications including preventing an input file from taking the designated name for the N+1 node, but those issues were dealt with relatively easily. Though this implementation of the N+1 node simplified rank generation for truncated orderings, it required additional effort when implementing systems interfacing with external ranking files.

7.3: Top-N Truncated Ranking Results

In this section, we apply rank snipping and internal truncation on our sports data to illustrate the usage of each. We anticipated that this functionality would be useful in the case of college football ranking analysis because many of the rankings published only considered the top-25 teams in the league. Therefore, we decided to first test both rank snipping and internal truncation on our 2016-17 NFL dataset to analyze how the total backedge weight and ranking correctness would compare in general. The following sections discuss our results alongside analysis and explanation.

7.3.1: Rank Snipping Example on 2016-17 NFL Data

In application, rank snipping was used to reduce the size of a ranking generated by the program. In Table 19, we display the results of two tests on the 2016-17 NFL dataset. The following tests were conducted using rank snipping on the Hill Climbing algorithm. For brevity, we are only displaying the top-10 of each ranking as our basis for comparison.

Table 19: Rank Snip on Hill Climb Ranking on 2016-17 NFL Dataset

Rank Snip Size: 10	**Rank Snip Size: 5**
New England Patriots	New England Patriots
Dallas Cowboys	Dallas Cowboys
Atlanta Falcons	Atlanta Falcons
Kansas City Chiefs	Kansas City Chiefs
Pittsburgh Steelers	Pittsburgh Steelers
Oakland Raiders	----
New York Giants	----
Seattle Seahawks	----
Green Bay Packers	----
Miami Dolphins	----
2.201/119.492 (1.84%)	0.773/119.492 (0.65%)

For reference, the total backedge weight of the non-snipped resulting ranking from the Hill Climbing algorithm on the dataset is 19.846/119.492 (16.61%). As shown in Table 19, the rank snipping approach removes all teams beyond the designated maximum amount of teams to retain in the rank snip. For example, in Table 19, the teams are ranked identically until rank index 6, where the rank snip of size 5 removes the remaining teams from the output ranking. It is important to note the different total backedge weights between each level of rank snipping. In the non-snipped ranking, the total backedge weight is 16.61%. The total backedge weight of the rankings rank snipped to size 10 and size 5 are 1.84% and 0.65%, respectively. Even though all three amounts of total backedge weight derived from identical ranking, the rankings with fewer teams have a smaller amount of backedge weight. This is by design; since there are fewer teams to consider, there are fewer backedges to be summed as a part of the total backedge weight. Total backedge weight cannot be calculated from edges to nonexistent teams or teams that are not present in the ranking. Therefore, it is important to only use rank snipping for comparison between rankings of the same size.

As demonstrated in Table 19, rank snipping is useful when we need to conform a ranking generated by our program to a specified size. Using the rank snip approach, we are able to utilize all of the available information and conduct the full set of calculations to generate a ranking of all teams in the dataset before snipping the ranking to the desired size. Since the full set of calculations are conducted on the full dataset, this process is not useful for reducing the required computation time for generating a ranking. The utility of rank snipping is the ability to appropriately compare rankings of sizes other than the total amount of teams in the dataset. Since all of our approximation algorithms executed in a reasonable amount of time for testing, as demonstrated in Section 5.2, the performance penalty for rank snipping was not a concern for the majority of our testing.

7.3.2: Internal Truncation Example on 2016-17 NFL Data

As discussed in Section 7.1.1, rank snipping was used to reduce the size of the sports data input by restricting which teams could be allowed in the ranking. In order to preserve the validity of the ranking, the same set of eligible teams needs to be used in both teams, since all computation on the part of the program is limited to those eligible teams. In practice, the set of eligible teams was given by the set of teams in the external ranking that the truncated ranking output from the program would be compared to. In Table 20, we display the results of input truncation on the NFL 2016-17 dataset. In this simplified test, the input used for the truncation was the ESPN power ranking for the 2016-17 season, modified to reflect only the top 10 teams in that ranking. The ranking generated by our program in the following test was created by the Hill Climbing algorithm.

Table 20: Internal Truncation on Hill Climb Ranking on 2016-17 NFL Dataset

ESPN Ranking	**Hill Climbing**
New England Patriots	New England Patriots
Dallas Cowboys	Dallas Cowboys
Pittsburgh Steelers	Pittsburgh Steelers
Kansas City Chiefs	Kansas City Chiefs
Green Bay Packers	Green Bay Packers
Seattle Seahawks	Atlanta Falcons
Atlanta Falcons	Oakland Raiders
New York Giants	New York Giants
Oakland Raiders	Seattle Seahawks
Detroit Lions	Detroit Lions
--Other_Teams--	--Other_Teams--
9.442/65.106 (14.503%)	9.606/65.106 (14.755%)

As shown in Table 20, the truncated Hill Climbing ranking uses the exact same teams as the ESPN ranking. Also shown at the bottom of each ranking is the N+1 node, called "--Other_Teams--". The rankings are very similar due to the small number of teams retained after truncation. The total backedge weight of the ESPN ranking is 14.503% while the total backedge weight of the Hill Climbing algorithm ranking is 14.755%. This result was surprising since most of our tests showed that our ranking algorithms usually perform better than external rankings, as demonstrated in Section 9.1. We reasoned that the cause of the slightly increased total backedge weight was the small size of the truncated input.

We investigated the total backedge weights of rankings generated from the same truncated dataset using several of our ranking algorithms. We performed trials with quartile evaluation enabled and disabled. The results are displayed in Figures 18 and 19. Note that Y3 refers to the Berger/Shor approximation algorithm with node sorting by win-loss ratio, and Y4 refers to the Berger/Shor approximation algorithm with node sorting by outgoing edge weight descending.

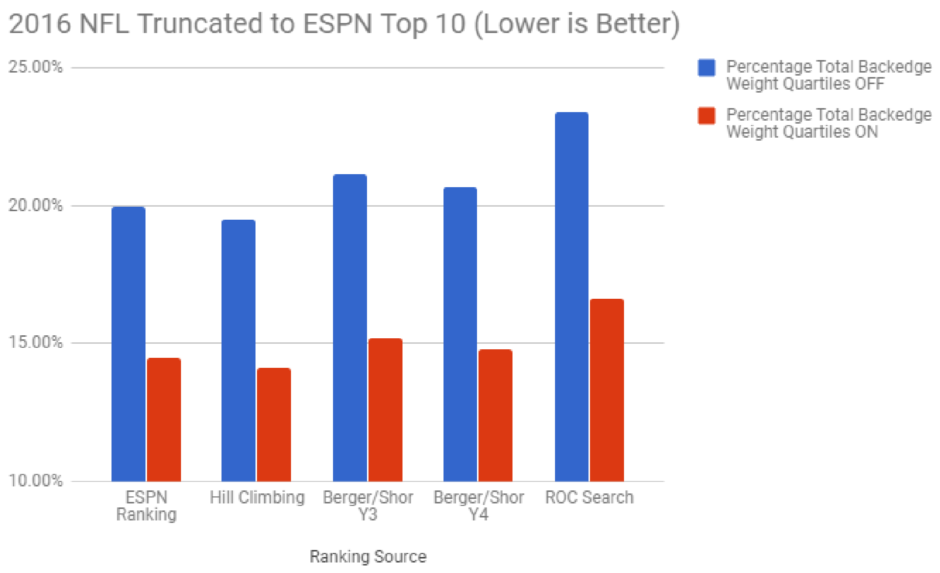


Figure 18: 2016-17 NFL truncated to ESPN top-10 testing

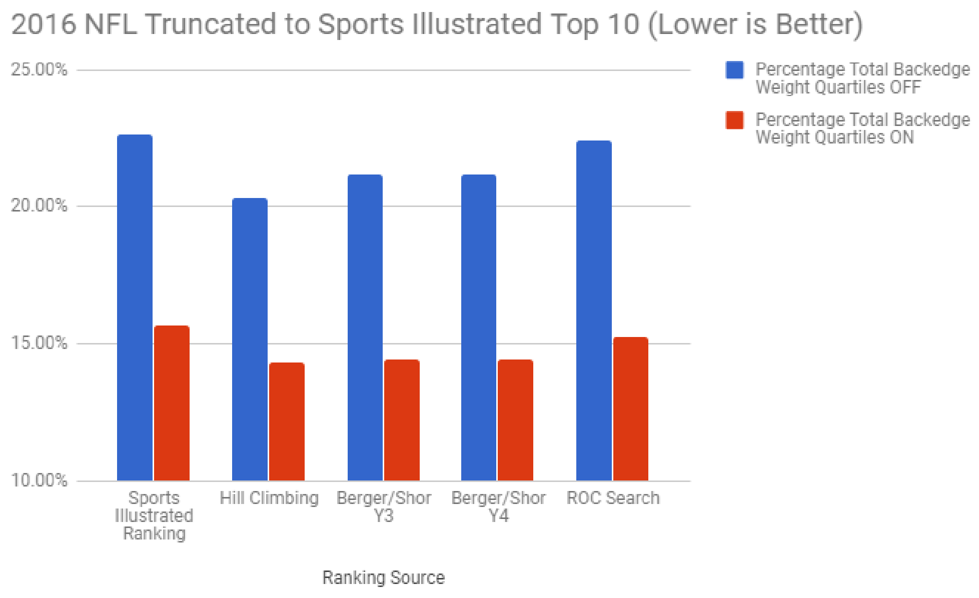


Figure 19: 2016-17 NFL truncated to Sports Illustrated top-10 testing

The charts in Figures 18 and 19 seem to be inconclusive over the claim that our algorithms do not perform as well in minimizing backedge weight when the ranking is truncated to a low number of teams. After performing the aforementioned tests, we concluded that as the input size is truncated to smaller numbers of teams, the consistency with which we outperform external rankings is reduced. However, our algorithms still generally outperform external rankings. Better performance of our algorithms against external rankings seems to be more situational when using smaller input datasets from internal truncation.

7.4: Summary

In this chapter, we introduced truncation and its applications to our project. We explained our explorations with rank snipping, internal truncation, and external truncation. The implementation process and drawbacks we discovered were described and explained. Finally, we applied our rank truncation functionality to a practical sports ranking application, demonstrating how it can be utilized in this project.

Chapter 8: Range of Correctness

In this chapter, we discuss the concept of Range of Correctness in relation to our rankings and how we utilized the Range of Correctness heuristic to design one of our algorithms. We first introduce the theory behind the Range of Correctness concept and explain how it is calculated. We discuss how this translated to implementation next. Then we explain our approach in adapting Range of Correctness into a post-process ranking system that can be run as its own algorithm.

8.1: Range of Correctness Design

Consideration of our total backedge weight methodology and the equivalency of rankings begged the question of how we could improve our generated rankings. From analysis of our graph-based approach, we realized that because not every team plays each other in most major sports, there likely existed some flexibility with regards to the placement of teams in our generated rankings. In this regard, some of the placement could be due purely because of how our sorting or preprocessing algorithms were implemented. We refer to this flexibility in rankings as Range of Correctness.

When developing a ranking for a graph where not all teams have played each other and the tournament constraint is not satisfied, we can apply a Range of Correctness to each team. The Range of Correctness for each team is a set of two bounds, an upper bound and a lower bound. We define **Range of Correctness** for any team as an exclusive range where the upper bound index is of the lowest-ranked team it lost to, and the lower bound index is of the highest-ranked team that it won against. In this case, **bounds** for Range of Correctness are the inclusive uppermost or lowermost position in the ranking whereby any swaps of a team's ranking within each bound would result in the same or reduced total backedge weight for a ranking. From a ranking perspective, the Range of Correctness for a team states, "I can be placed anywhere below the rank of the worst team I lost to and above the rank of the best team I won against."

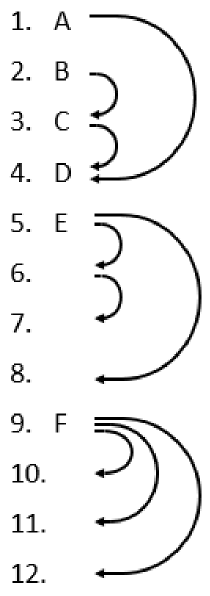
We implemented a function to calculate and output the Range of Correctness alongside every team when generating a ranking to analyze with our test data. The variances in ranges that occurred indicated room for improvement with post-processing, and typically indicated the density of the sports graph. A higher density would result in a smaller Range of Correctness for each team because there are more games and edges to contradict rankings, resulting in fewer valid rankings.

8.1.1: Rank Pool

To further improve the correctness of our rankings, we applied the Range of Correctness heuristic in a post-processor. All three of our algorithms had been evaluated using only total backedge weight as a metric, but exploration into Range of Correctness demonstrated that there was still room for improvement by considering forward edges. We designed a pool data structure based around Range of Correctness, which we define as the **rank pool**, which maintains the index for the rank position it is selecting a node for. Nodes are added into the pool once the pool index reaches the upper bounds of their Ranges of Correctness. The pool then checks if any nodes have a lower bound of the index being evaluated. If one does, that node is selected for the given ranking; otherwise, the pool sorts its nodes by net edge weight, assigning priority to teams in our graph who have won more important games. The node with the highest net edge weight is selected for the given index, then the pool increments its index and repeats the process until all nodes have been post-processed.

However, we discovered a counterexample which proved that the Range of Correctness could not maintained for every team in all cases. In general, this occurs when one or more teams with larger Ranges of Correctness are placed within the range of one or more teams' Ranges of Correctness, resulting in a conflict where at least one team must be displaced. The full counterexample is provided and discussed in the following section.

8.1.2: Rank Pool Counterexample and Proof



	Indegree	Outdegree	Range of Correctness	Rank Pools	
1. A	0	1	[1, 3]	1:	A, B, E, F
2. B	0	1	[1, 2]	2:	A, B, E
3. C	1	1	[3, 3]	3:	A, C, E
4. D	2	0			
5. E	0	2	[1, 5]		
6.					
7.					
8.					
9. F	0	3	[1, 9]		
10.					
11.					
12.					

Figure 20: Initial rank pool counterexample

Suppose we have the ranking generated in Figure 20. A has no losses, but has one win over D, ranked fourth, so A's Range of Correctness is [1,3]. B has no losses, but has one win over C, ranked third, so B's Range of Correctness is [1,2]. C has one loss from B and one win over D, so C's Range of Correctness is [3,3]. Similarly, E's Range of Correctness is [1,5], and F's is [1,9]. Once the rank pool begins at index 1, nodes A, B, E, and F are added as they all have left bounds of 1. F is chosen for rank 1 in the final ranking because no nodes in the pool have a right bound of 1 and because F has the highest net edge weight of the nodes in the pool. The rank pool advances to index 2, where B is chosen because it has a right bound of 2, so it must be ranked at index 2. However, once the rank pool advances to index 3, we have two nodes A and C which both have right bounds of 3, meaning one node's Range of Correctness will be violated if we proceed. The violation of either node’s Range of Correctness means that a new backedge will be introduced and we can no longer guarantee that this post-processor will produce a better ranking.

8.1.3: Revised Rank Pool

To address the aforementioned counterexample, we revisited the requirements for the Range of Correctness. We determined that, after one iteration of reordering with the rank pool, we were left with a new, unique ranking. This ranking, because it followed just one iteration of the rank pool process and completed one reordering within the Range of Correctness, creates no new backedges compared to the original ranking. Then, this new ranking has its Ranges of Correctness generated, by which the nodes in the rank pool are updated, and the algorithm continues until all nodes have been post-processed. Furthermore, any conflicts that may occur by selecting one team for a given rank over another are resolved when the new Range of Correctness is generated, as any boundaries of the range are shifted as the teams in the ranking get swapped. After this resolution, we implemented the rank pool post-processor to begin with a ranking and its respective Range of Correctness, and conduct the removal of teams from pools and re-evaluation of the Range of Correctness until the new ranking is finalized.

8.2: Range of Correctness Implementation

The implementation for the Range of Correctness was designed to be as compatible as possible with the existing internal ranking structure. Since Range of Correctness needed to be applied to internal rankings from multiple different ranking algorithm sources as well as work with external rankings, generating the Range of Correctness needed to be a standalone process. Range of Correctness was implemented to be a lightweight and simple module that could take rankings of different input types and generate the Range of Correctness in a standard and useful format. We agreed upon a standard input and output to the Range of Correctness calculation function to allow any functionality that we might build off of Range of Correctness to not have to deal with the many alternative inputs of the rankings.

The function to calculate the Range of Correctness took a ranking as input and output the resulting ranges as two lists. We decided to represent the Range of Correctness as two lists of bounds, one list of upper bounds and one list of lower bounds. Using this system, we were able to know the upper bound and lower bound at each index in the ranking. Instead of creating and displaying the Range of Correctness immediately in the function, we reasoned that it was important to create a format that be held in memory and manipulated for further calculation.

The process to calculate each bound for a team was implemented in a relatively simple manner. First, the program looked at adjacent teams in the given ranking. Next, the process would search through the adjacency matrix to determine if there were any relevant edges between the control team and the test team. We define **relevant edges** as edges that would become backedges if the positions of teams were flipped, thus breaking the Range of Correctness. If a relevant edge was discovered, the search for a bound would terminate. As the process continued, it would look to each next most adjacent team in the ranking and check for relevant edges. This process is shown below in Figure 21. Once all of the upper and lower bounds were calculated for each team, the bounds were returned to be manipulated and displayed as needed.

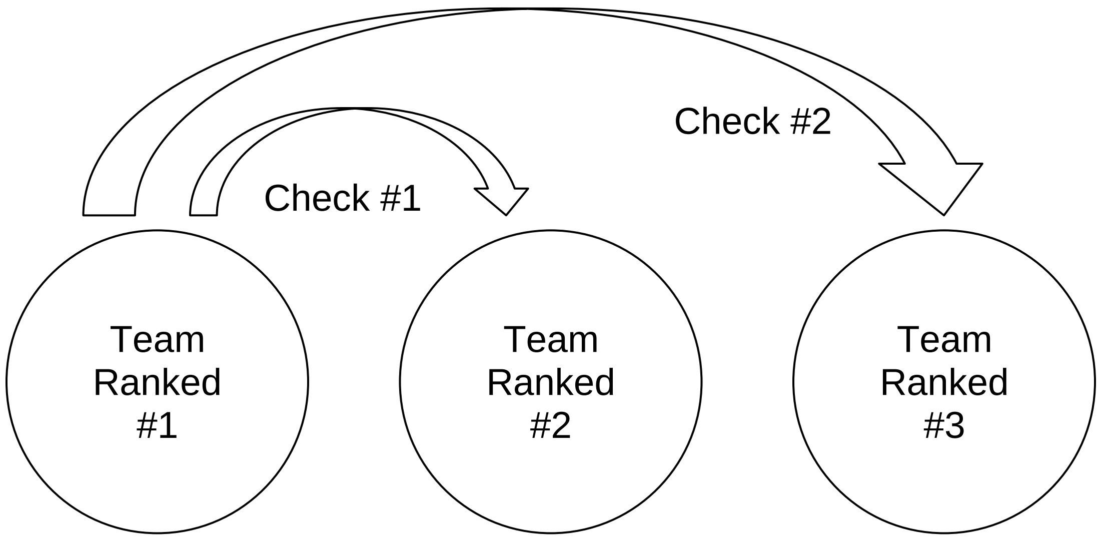
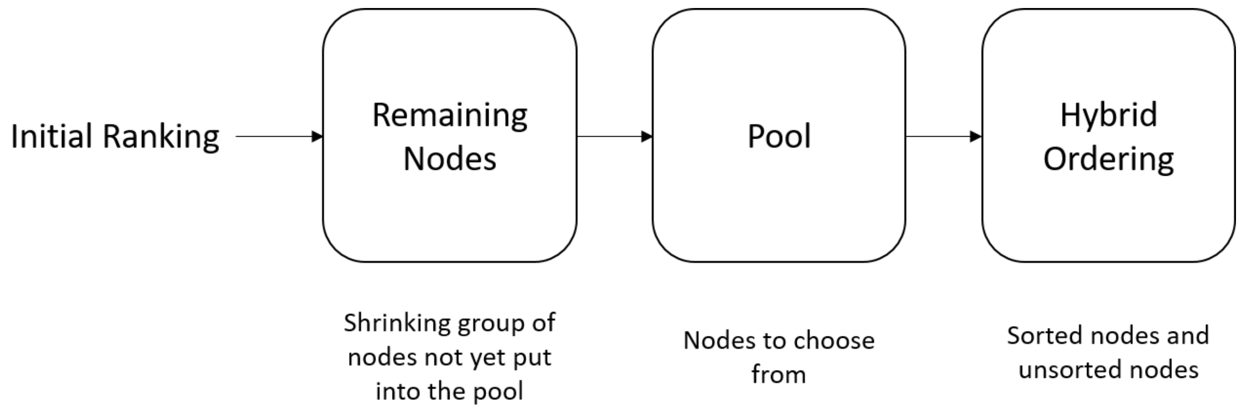


Figure 21: Lower bound determination in Range of Correctness

One drawback of this approach was the high memory overhead associated with retaining so many values in memory at once. The total memory usage of this process included the pointer to the array, as well as space for integers for twice the number of teams for each of the bounds. In combination with the rest of the program overhead, this was a lot of dynamic memory allocated to the stack. On several occasions, we ran into program crashes when this process caused the stack memory to run out. Despite the memory challenges, this approach to the implementation of the Range of Correctness afforded multiple benefits that ultimately outweighed the extra engineering required to accommodate the overhead. It allowed the Range of Correctness to be displayed, written to a results file, and used in further calculation. The modular nature of this implementation allowed the Range of Correctness to be queried in separate functions. The portability of the Range of Correctness calculation was useful in the implementation for the Rank Pool Post-Process.

8.2.1: Rank Pool Post-Process Implementation

The rank pool post-process was implemented to take in a ranking of nodes and modify it through the rank pool algorithm designed in Section 8.1.3. The input for the rank pool post-process is a reference to a vector ordering of nodes, which represents the ranking. There is no direct return output from the rank pool post-process as it was implemented to modify the initial ordering in-place to reduce memory overhead rather than preserve the initial ranking and return an entirely new ordering in memory. Before any computations can take place, the rank pool post-process first initializes the containers for the rank pool and generates the Range of Correctness for the input ranking to be computed upon. The rank pool computation procedure was implemented to use three main containers of nodes. The sorting process using the containers is illustrated in Figure 22 below.


Initial Ranking
Remaining Nodes
Pool
Hybrid Ordering
Shrinking group of nodes not yet put into the pool
Nodes to choose from
Sorted nodes and unsorted nodes
Figure 22: Rank pool node containers

As shown in Figure 22, nodes from the initial non-post-processed ranking are used to fill the three containers of nodes, where the entire post-process computation takes place. Emptiness in the containers was implemented using a special placeholder value that signified the absence of a node. Using the placeholder for emptiness, the statically sized memory for the containers could hold varying amounts of nodes.

The first container for the rank pool post-process is the "Remaining Nodes" container. This container holds all nodes from the initial ranking that have not been selected into the rank pool container. This container is initialized with all of the nodes in the initial ranking. The ordering of nodes in the "Remaining Nodes" container does not matter since nodes are considered unsorted before being chosen into the rank pool container. The contents of the "Remaining Nodes" container slowly diminish as the rank pool post-process executes and nodes become sorted. By the end of the rank pool post-process, this container is empty.

The second container for the rank pool post-process is the rank pool itself. As designed in Section 8.1.1 and revised in Section 8.1.3, the rank pool is a collection of nodes that are eligible and are being considered for sorting. The nodes in this container are unsorted as the process for choosing a node from the pool is not altered by the ordering of nodes.

The third container for the rank pool post-process is the hybrid ordering. The hybrid ordering is an ordering of all of the nodes where some of the nodes are sorted and some are not. The hybrid ordering holds a "work in progress" combination of the sorted-so-far finalized ordering and the ordering of nodes from the original ranking. The structure of the hybrid ordering is illustrated in Figure 23 below.

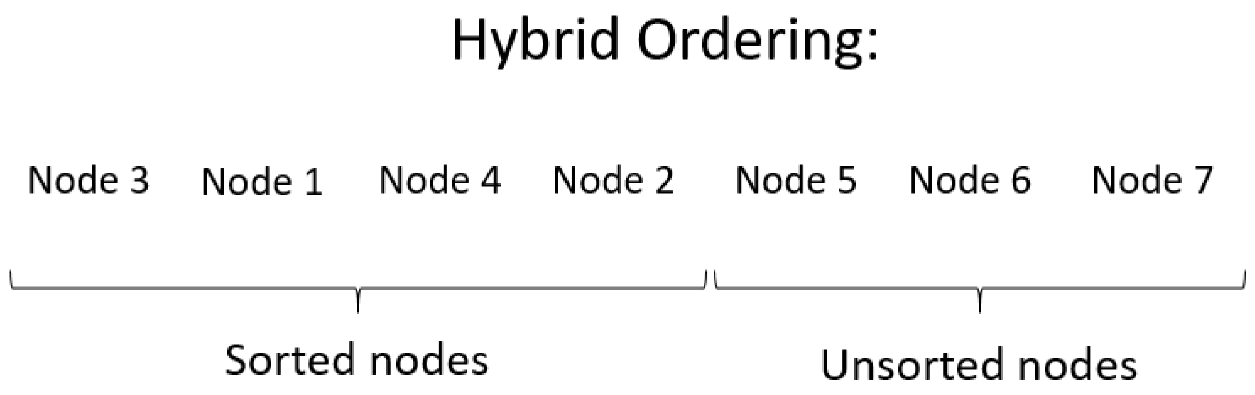


Figure 23: Composition of the hybrid ordering for rank pool post-process

As shown in Figure 23, the first part of the hybrid ordering is the growing ordering of nodes that have been sorted in the rank pool. The second part of the hybrid ordering is the remaining nodes that have not exited the rank pool, which remain in their ordering from the initial ranking. As the rank pool post-process iterates, nodes are selected from the rank pool to the finalized post-processed ordering. However, to retain integrity of the ordering, the Range of Correctness needs to be applied to the “work in progress” sorted ordering. The hybrid ordering was implemented as the data container to regenerate the Range of Correctness upon, as described in Section 8.3.1. The hybrid ordering is initialized as a copy of the initial non-post-processed ranking because none of the nodes have been sorted by the rank pool at the beginning of the rank pool post-process yet. Once the post-process has completed, the hybrid ordering will consist entirely of nodes sorted through the rank pool and therefore be the post-processed ordering.

The rank pool post-process generates a post-processed ranking by taking the initial ranking of nodes, sorting nodes in the rank pool, and iteratively generating a sorted list of nodes that becomes the post-processed result. The main steps of the process are outlined in Figure 24.

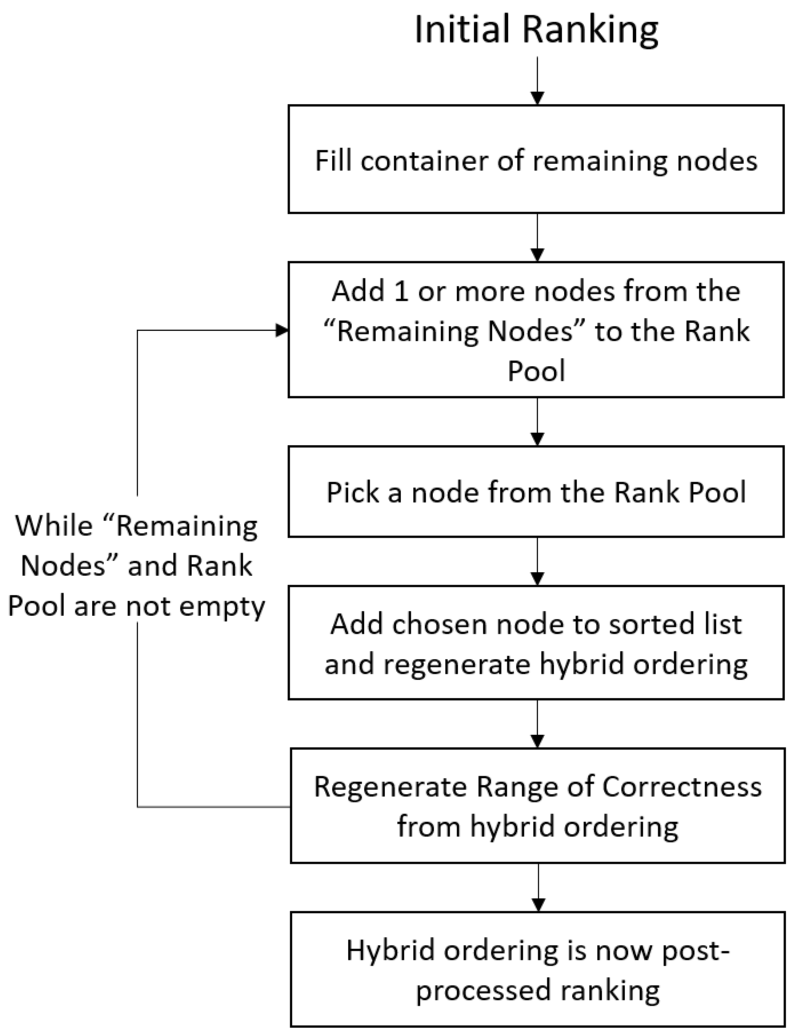

Initial Ranking
Fill container of remaining nodes
Add 1 or more nodes from the "Remaining Nodes" to the Rank Pool
Pick a node from the Rank Pool
Add chosen node to sorted list and regenerate hybrid ordering
Regenerate Range of Correctness from hybrid ordering
Hybrid ordering is now post-processed ranking
While "Remaining Nodes" and Rank Pool are not empty
Figure 24: Structure of rank pool post-process implementation

Figure 24 shows the workflow of the rank pool post-process on a single loop. There are three main steps to the loop that iteratively build the resulting post-processed ranking. The first main step is to transfer eligible nodes from the "Remaining Nodes" container to the rank pool. The size of the rank pool will grow, but the contents of the "Remaining Nodes" container will diminish by the same amount. The second step is to pick a single node from the rank pool. In this step, the rank pool diminishes in size by one node. The third step is to add the chosen node to the sorted part of the hybrid ordering and to regenerate the Range of Correctness from the new hybrid ordering. Once all of the nodes have been processed in the rank pool, the fully sorted hybrid ordering is transferred to the post-processed ranking.

8.2.2: ROC Search Algorithm

The ROC Search algorithm was inspired by the rank pool post-process. The ROC Search algorithm is the application of the rank pool post-process in a hill climbing fashion. During our implementation of the rank pool post-process, we noted the ability for the rank pool to optimize a given ranking in total backedge weight. We also recalled that the rank pool would function properly with any given input ranking. Since the rank pool post-process generated a complete ranking of a set of teams, we logically concluded that the rank pool could be adapted into a fourth ranking algorithm.

The ROC Search algorithm was implemented in a similar manner as our implementation of the Hill Climbing ranking algorithm. The ROC Search begins with a starting ordering based upon the descending net weight of each node in the ranking and calculates the current total backedge weight for comparison with the post-processed rankings. The algorithm then enters a while loop, where it continues to conduct the rank pool post-process algorithm on the current ordering. The post-processed ranking is then evaluated and its total backedge weight is compared to the best found so far. If the post-processed ranking is better, the new permutation and backedge weight are maintained; otherwise, ROC Search terminates and returns the best ranking found. Unlike our Hill Climbing algorithm, ROC Search does not utilize sideways moves or random restarts. The pseudocode for our ROC Search algorithm is listed below.

```
algorithm roc_search() is
      input: graph G = (V, E)
      output: total ordering T

      best_weight <- weight(G)
      best_permutation <- {}
      permutation <- order vertices by decreasing net edge weight
      continue <- true
      while continue is true do
            permutation <- rank_pool(permutation)
            weight <- total_backedge_weight(permutation)
            if weight < best_weight do
                  best_weight <- weight
                  best_permutation <- permutation
            else
                  continue <- false
            end if
      end while
      return best_permutation
end
```

8.3: Range of Correctness Results

Once the Range of Correctness concept was implemented in our program as the rank pool post-processor and ROC Search algorithm, we conducted test cases on both functionalities to see how they could be applied to reduce total backedge weights for input rankings. In this section, we provide two types of tests cases: Range of Correctness demonstrations showcasing the variability in ranges on our sports datasets, and rank pool post-processor tests, where we demonstrate the improvements in total backedge weight and overall correctness by evaluating rankings generated by our algorithms and external rankings to see how much they can be improved.

The first set of test cases we conducted explore the differences in Range of Correctness between different sports datasets which have different densities. As discussed in Section 8.1, we expect that denser graphs will result in smaller Ranges of Correctness on average due to less flexibility in the upper and lower bounds based on more win and loss edges. Our test cases were conducted on the 2016-17 NFL dataset and the 2015 MLB dataset. These tests utilize our Hill Climbing algorithm to generate the rankings shown, though only the top-10 teams are shown for brevity. In our data tables for this section, the Range of Correctness column contains the upper bound for the range as the left index and the lower bound for the range as the right index.

Table 21: Ranges of Correctness Between 2016-17 NFL and 2015 MLB

NFL 2016-17	Range of Correctness [upper - lower]	MLB 2015	Range of Correctness [upper - lower]
Patriots	[1-4]	Blue Jays	[1-1]
Falcons	[1-5]	Astros	[2-3]
Chiefs	[1-7]	Cubs	[1-3]
Cowboys	[1-4]	Giants	[4-4]
Steelers	[5-7]	Mets	[5-6]
Cardinals	[3-10]	Pirates	[5-6]
Packers	[5-8]	Nationals	[7-9]
Colts	[6-14]	Indians	[7-8]
Giants	[8-9]	Rangers	[9-9]
Bengals	[10-15]	Dodgers	[10-11]

Table 21 supports our hypothesis that the density of the input graph greatly impacts the Ranges of Correctness of our rankings. In graphs that are less dense, where there are relatively few games played per team, the Ranges of Correctness are larger. There are fewer games played per team in the NFL season compared to the MLB season, so it was expected that the Ranges of Correctness would be larger. As shown in Table 21, the Ranges of Correctness for the NFL rankings are larger than the Ranges of Correctness for the MLB rankings, confirming our hypothesis. In our testing, we found that rankings for seasons with higher density and greater connectedness have narrower ranges of correctness. The Ranges of Correctness in the rankings for college football have the greatest Ranges of Correctness from our datasets. In a tournament graph, where every team has played every other team, we can extend our hypothesis to state that there will be no Ranges of Correctness because each team has lost to the team ranked above it and won against the team ranked below it.

8.3.1: Rank Pool Results

The second set of tests we conducted were rank pool post-processor tests on our sports datasets, where we investigate if our rank pool post-processor is able to improve upon rankings, and to what degree if so. Our first test in this set was conducted using the Berger/Shor algorithm with the outgoing edge weight node sorting preprocessor (Y4) on our 2016-17 NFL dataset. The full rankings were computed for this test, but only the top-10 teams are displayed. Additionally, the total backedge weights displayed underneath are representative of the full ranking.

Table 22: Rank Pool Post-Process on 2016-17 NFL using the Berger/Shor Algorithm

Berger/Shor	**Berger/Shor (Post-Processed)**
South Florida	Clemson
San Diego State	Alabama
Air Force	Washington
Wisconsin	Oklahoma
Western Michigan	Western Kentucky
Oklahoma	Wisconsin
Oklahoma State	Western Michigan
Houston	San Diego State
Clemson	South Florida
Virginia Tech	Stanford
Total backedge weight: 9.25%	Total backedge weight: **8.67%**

In Table 22, the rank pool post-process is shown to improve the initial ranking generated by the Berger/Shor algorithm. The rank pool post-processor improves both the total backedge weight and the listing of teams. The ordering of teams in the top-10 of the initial Berger/Shor ranking does not align with any external rankings, so we have reason to believe it is not accurate. The included teams in the post-processed ranking are much closer to the teams one would expect in a top-10 college football teams ranking. In our testing with the rank pool post-processor, we observed total backedge weights for the post-processed rankings that were equivalent or better than without post-processing.

The second test case we conducted in this set was with the ESPN power ranking for the 2016-17 NFL dataset to determine if the rank pool post-processor can improve external rankings. Similar to the prior test, only the top-10 teams are shown, though the total backedge weight are reflective of the full ranking.

Table 23: Rank Pool Post-Process on 2016-17 NFL External Ranking (ESPN, Quartiles On)

NFL 2016-17 ESPN Ranking	NFL 2016-17 ESPN Ranking Post-Processed
New England Patriots	New England Patriots
Dallas Cowboys	Dallas Cowboys
Pittsburgh Steelers	Pittsburgh Steelers
Kansas City Chiefs	Kansas City Chiefs
Green Bay Packers	Green Bay Packers
Seattle Seahawks	Seattle Seahawks
Atlanta Falcons	Atlanta Falcons
New York Giants	Oakland Raiders
Oakland Raiders	New York Giants
Detroit Lions	Miami Dolphins
Denver Broncos	Tennessee Titans
Miami Dolphins	Arizona Cardinals
Tampa Bay Buccaneers	Detroit Lions
Baltimore Ravens	Denver Broncos
Tennessee Titans	Tampa Bay Buccaneers
19.45%	**17.38%**

Table 24: Rank Pool Post-Process Applied to 2016-17 NFL External Rankings

NFL 2016-17	ESPN	NFL.com	Sports Illustrated
Original	19.45%	18.65%	20.29%
Post-Processed	**17.38%**	18.65%	**19.33%**

As shown in Table 23, the rank pool post-process reorganizes the ESPN power ranking and improves the total backedge weight. We tested the rank pool post-processor on three external rankings. In each case, the post-processor returned an equivalent ranking or one with a lower total backedge weight, shown in Table 24. The accuracy of rankings from this post-processor is improved as a result, encouraging the use of this post-processor. In our testing, the rank pool post-processor always produced a better ranking by changing the ordering of teams, sometimes reducing the total backedge weight.

Table 25: Hill Climbing vs. ROC Search on 2017-18 NFL Dataset

Hill Climbing	ROC Search
New England Patriots	New England Patriots
Philadelphia Eagles	Philadelphia Eagles
Minnesota Vikings	Minnesota Vikings
Pittsburgh Steelers	Pittsburgh Steelers
Los Angeles Rams	Los Angeles Rams
13.56%	13.94%

Table 25 demonstrates the top-5 teams from the rankings generated by Hill Climbing and ROC Search on the 2017-18 NFL dataset. Both top-5 rankings are equivalent, where the only differences in ordering occurs near the bottom of the full ranking. Although Hill Climbing generates a ranking with a lower total backedge weight, ROC Search is competitive in the total backedge weight of its ranking. From this, we can conclude that considering the forward edge weights of rankings leads to similar improvements in the total backedge weight metric as minimizing backedges. The similarities in total backedge weights demonstrate that this heuristic has a suitable basis to generate and improve upon rankings.

8.4: Summary

In this chapter, we introduced the concept behind the Range of Correctness post-processor that we developed during this project. The theory behind the Range of Correctness was explored, alongside a counterexample and resolution with logical backing. We then detailed the implementation of Range of Correctness into our program and its abstraction into an algorithm for generating rankings. Finally, we provided an example of practical applications of the post-processor and ranking algorithm with improvements in correctness for given rankings.

Chapter 9: Testing with Sports Rankings

With all of the designed functionalities implemented and tested within our program, we tested our algorithms against the various sports datasets we had collected. Our test cases utilized one season dataset from several major sports: 2016-17 National Football League (NFL) regular season, 2016-17 College Football (CFB) full season, 2014-15 National Hockey League (NHL) regular season, and 2015 Major League Baseball (MLB) regular season. In this chapter, we evaluate test cases using our total backedge weight metric and our rank differential metric on these datasets. We review and compare the results and offer possible explanations for why our algorithms performed the way they did.

Conducting test cases with one season dataset per sport allowed us to understand how our algorithms performed in each sport and analyze the differences between sports. We expected that each sport would result in graphs with different characteristics which would impact the performance and correctness of our algorithms. These characteristics include number of nodes or teams, and the number of games per team, or density of the graph. Table 26 shows the number of teams and density of the graphs we tested with.

Table 26: Number of Teams and Graph Density for Sports Test Cases

	2016-17 NFL	2016-17 CFB	2014-15 NHL	2015 MLB
Number of teams	32	128	31	30
Games played per team (density)	16	12-13	82	162

9.1: Total Backedge Weight Results

The first set of full test cases we conducted utilized the total backedge weight metric discussed in Section 2.3.1. In these test cases, we applied our algorithms with all correctness optimizations enabled against external rankings and standings for these datasets. Each sports dataset was tested with four permutations of Alpha and Beta values: (0,0), (0,1), (1,0), and (0.5,0.5) to see which value pairs gave our rankings and the external rankings the lowest total normalized backedge weight. All of the rankings produced by our algorithms have been post-processed by the rank pool post-processor as described in Section 8.2.1.

9.1.1: National Football League (NFL) Ranking Results

Table 27: Algorithm Comparisons on 2016-17 NFL (Alpha 0, Beta 0)

ESPN	NFL.com	Sports Illustrated	Y3	Y4	Hill Climbing	ROC Search
Patriots	Patriots	Patriots	Patriots	Patriots	Patriots	Patriots
Cowboys	Cowboys	Cowboys	Steelers	Steelers	Falcons	Falcons
Steelers	Steelers	Falcons	Falcons	Falcons	Chiefs	Chiefs
Chiefs	Chiefs	Chiefs	Chiefs	Chiefs	Cardinals	Cardinals
Packers	Falcons	Steelers	Cardinals	Bengals	Cowboys	Cowboys
Seahawks	Packers	Packers	Bengals	Colts	Steelers	Bengals
Falcons	Giants	Raiders	Colts	Ravens	Bengals	Steelers
Giants	Seahawks	Giants	Packers	Packers	Colts	Colts
Raiders	Dolphins	Dolphins	Buccaneers	Seahawks	Packers	Saints
Lions	Lions	Buccaneers	Seahawks	Eagles	Saints	Buccaneers
16.40%	15.13%	15.98%	**7.37%**	7.98%	11.61%	10.41%

Table 28: Algorithm Comparisons on 2016-17 NFL (Alpha 0, Beta 1)

ESPN	NFL.com	Sports Illustrated	Y3	Y4	Hill Climbing	ROC Search
Patriots	Patriots	Patriots	Patriots	Patriots	Patriots	Patriots
Cowboys	Cowboys	Cowboys	Cowboys	Cowboys	Chiefs	Cowboys
Steelers	Steelers	Falcons	Steelers	Steelers	Cowboys	Steelers
Chiefs	Chiefs	Chiefs	Chiefs	Chiefs	Steelers	Chiefs
Packers	Falcons	Steelers	Falcons	Falcons	Falcons	Falcons
Seahawks	Packers	Packers	Dolphins	Dolphins	Dolphins	Raiders
Falcons	Giants	Raiders	Giants	Raiders	Raiders	Giants
Giants	Seahawks	Giants	Raiders	Colts	Giants	Dolphins
Raiders	Dolphins	Dolphins	Colts	Titans	Packers	Packers
Lions	Lions	Buccaneers	Titans	Packers	Titans	Titans
16.01%	15.29%	15.58%	11.15%	**11.05%**	15.69%	14.77%

Table 29: Algorithm Comparisons on 2016-17 NFL (Alpha 1, Beta 0)

ESPN	NFL.com	Sports Illustrated	Y3	Y4	Hill Climbing	ROC Search
Patriots	Patriots	Patriots	Patriots	Patriots	Patriots	Patriots
Cowboys	Cowboys	Cowboys	Cowboys	Cardinals	Falcons	Falcons
Steelers	Steelers	Falcons	Eagles	Steelers	Steelers	Steelers
Chiefs	Chiefs	Chiefs	Falcons	Chiefs	Cardinals	Cardinals
Packers	Falcons	Steelers	Steelers	Eagles	Cowboys	Cowboys
Seahawks	Packers	Packers	Cardinals	Falcons	Chiefs	Chiefs
Falcons	Giants	Raiders	Chiefs	Cowboys	Eagles	Seahawks
Giants	Seahawks	Giants	Chargers	Broncos	Bills	Eagles
Raiders	Dolphins	Dolphins	Colts	Colts	Seahawks	Bills
Lions	Lions	Buccaneers	Vikings	Bengals	Chargers	Chargers
15.93%	15.72%	18.99%	**5.06%**	6.86%	12.24%	9.07%

Table 30: Algorithm Comparisons on 2016-17 NFL (Alpha 0.5, Beta 0.5)

ESPN	NFL.com	Sports Illustrated	Y3	Y4	Hill Climbing	ROC Search
Patriots	Patriots	Patriots	Patriots	Patriots	Patriots	Patriots
Cowboys	Cowboys	Cowboys	Cowboys	Cowboys	Cowboys	Cowboys
Steelers	Steelers	Falcons	Steelers	Steelers	Falcons	Falcons
Chiefs	Chiefs	Chiefs	Chiefs	Seahawks	Chiefs	Chiefs
Packers	Falcons	Steelers	Giants	Falcons	Steelers	Steelers
Seahawks	Packers	Packers	Seahawks	Dolphins	Raiders	Raiders
Falcons	Giants	Raiders	Falcons	Chiefs	Giants	Giants
Giants	Seahawks	Giants	Dolphins	Raiders	Seahawks	Seahawks
Raiders	Dolphins	Dolphins	Raiders	Colts	Packers	Packers
Lions	Lions	Buccaneers	Colts	Titans	Dolphins	Dolphins
16.28%	15.60%	16.98%	13.64%	**13.37%**	16.61%	16.33%

Tables 27 to 30 show the results from our 2016-17 NFL dataset tests, with differing Alpha and Beta values. In these tests, we had three power rankings from ESPN, NFL.com, and Sports Illustrated as comparisons. Additionally, we used our Berger/Shor approximation algorithm with node sorting by win-loss ratio (Y3) and by outgoing edge weight decreasing (Y4), as well our Hill Climb and ROC Search algorithms.

The first point we noticed was that at least one of our algorithms beat all three external rankings for this dataset in every Alpha/Beta configuration, with improvements ranging from 4-7% less backedge weight from the graph. Of our algorithms, Y3 and Y4 did the best overall, where Y3 performed the best with a Beta value of 0, and Y4 performed the best with a Beta value greater than 0. Additionally, most of the top-10 teams in our rankings were present in the top-10 of the external rankings, with roughly half of the teams in each ranking being present in the external rankings. Of our algorithms, ROC Search performed better with all four Alpha/Beta values than Hill Climb. Additionally, for external rankings, NFL.com performed the best in all Alpha/Beta combinations, but ESPN performed better than Sports Illustrated with Beta values greater than 0, while Sports Illustrated performed better than ESPN with Beta values of 0.

With regards to the total backedge weight, the Alpha/Beta combination of (0.5,0.5) had the closest values. We suspect that this is because no edges are removed from the graph in this scenario; if Alpha is 0, the first week of games in the dataset are removed, and if Beta is 0, all games within one touchdown of the final score are removed. Without these edges removed, more backedge weight will exist that our algorithms cannot optimize around, so the total backedge weight of the resulting rankings will be higher. Additionally, the results show trials where the Beta value was 0 had larger variance in total backedge weights than trials where the Alpha value was 0, indicating that there were likely more edges to be lost in close games than in decay in the dataset.

A summary of the total backedge weight percentages for the 2016-17 NFL dataset with our algorithms and existing rankings are shown in Figure 25.

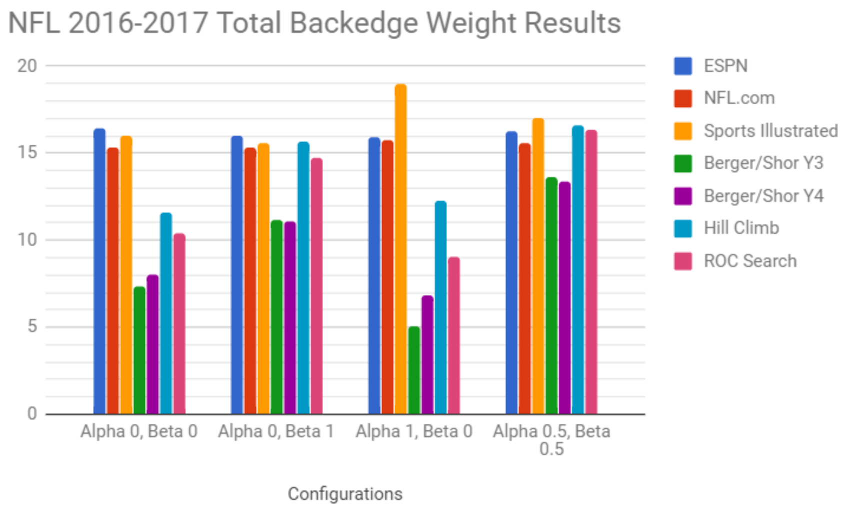


Figure 25: NFL 2016-2017 total backedge weight results

9.1.2: College Football (CFB) Ranking Results

Table 31: Algorithm Comparisons on 2016-17 CFB (Alpha 0, Beta 0)

AP	Bleacher Report	ESPN	Y3	Y4	Hill Climbing	ROC
Clemson	Clemson	Clemson	Alabama	Alabama	Alabama	Alabama
Alabama	Alabama	Alabama	Western Kentucky	Western Kentucky	Western Kentucky	Western Kentucky
South Carolina	Washington	South Carolina	Clemson	Clemson	Clemson	Clemson
Washington	South Carolina	Washington	Temple	Washington	Washington	Washington
Oklahoma	Ohio State	Penn State	Southern California	Temple	Temple	Temple
Ohio State	Oklahoma	Oklahoma	Washington	Southern California	Appalachian State	Appalachian State
Penn State	Wisconsin	Ohio State	Wisconsin	San Diego State	Arkansas State	Arkansas State
Florida State	Penn State	Florida State	Western Michigan	Wisconsin	Western Michigan	Penn State
Wisconsin	Florida State	Michigan	Louisiana State	Western Michigan	Oklahoma	Oklahoma
Michigan	Michigan	Wisconsin	Michigan	Louisiana State	Penn State	Southern California
0.53%	0.49%	0.38%	0.11%	**0.11%**[1]	0.47%	0.34%

[1] Although Y3 and Y4 have the same percentage, Y4 performed better (0.14375/130.95 vs 0.15/130.95); this difference was lost in the two-decimal precision.

Table 32: Algorithm Comparisons on 2016-17 CFB (Alpha 0, Beta 1)

AP	Bleacher Report	ESPN	Y3	Y4	Hill Climbing	ROC
Clemson	Clemson	Clemson	Clemson	Clemson	Clemson	Clemson
Alabama	Alabama	Alabama	Oklahoma	Oklahoma	Oklahoma	Oklahoma
South Carolina	Washington	South Carolina	Southern California	Alabama	Southern California	Southern California
Washington	South Carolina	Washington	Alabama	Southern California	Alabama	Alabama
Oklahoma	Ohio State	Penn State	Western Kentucky	Western Kentucky	Western Kentucky	Western Kentucky
Ohio State	Oklahoma	Oklahoma	Western Michigan	Western Michigan	Western Michigan	Old Dominion
Penn State	Wisconsin	Ohio State	Old Dominion	Old Dominion	Old Dominion	South Florida
Florida State	Penn State	Florida State	South Florida	South Florida	South Florida	Appalachian State
Wisconsin	Florida State	Michigan	Penn State	Penn State	Appalachian State	Stanford
Michigan	Michigan	Wisconsin	Wisconsin	Wisconsin	Stanford	Penn State
1.20%	1.28%	0.96%	0.42%	0.40%	0.55%	**0.39%**

Table 33: Algorithm Comparisons on 2016-17 CFB (Alpha 1, Beta 0)

AP	Bleach Report	ESPN	Y3	Y4	Hill Climbing	ROC
Clemson	Clemson	Clemson	Alabama	Alabama	Alabama	Alabama
Alabama	Alabama	Alabama	Michigan	Washington	Washington	Washington
South Carolina	Washington	South Carolina	Temple	Michigan	Michigan	Michigan
Washington	South Carolina	Washington	Appalachian State	Temple	Western Michigan	Western Michigan
Oklahoma	Ohio State	Penn State	Western Kentucky	Western Kentucky	Louisville	Temple
Ohio State	Oklahoma	Oklahoma	Clemson	Clemson	Appalachian State	Appalachian State
Penn State	Wisconsin	Ohio State	Wisconsin	Wisconsin	Temple	Louisville
Florida State	Penn State	Florida State	Western Michigan	Western Michigan	Western Kentucky	Western Kentucky
Wisconsin	Florida State	Michigan	Ohio State	Ohio State	Clemson	Clemson
Michigan	Michigan	Wisconsin	Penn State	Penn State	Ohio State	Ohio State
0.74%	0.63%	0.68%	0.19%	**0.04%**	0.57%	0.57%

Table 34: Algorithm Comparisons on 2016-17 CFB (Alpha 0.5, Beta 0.5)

AP	Bleacher Report	ESPN	Y3	Y4	Hill Climbing	ROC
Clemson	Clemson	Clemson	Clemson	Clemson	Alabama	Alabama
Alabama	Alabama	Alabama	Alabama	Alabama	Clemson	Clemson
South Carolina	Washington	South Carolina	Washington	Western Michigan	Western Michigan	Western Michigan
Washington	South Carolina	Washington	Oklahoma	Washington	Washington	Washington
Oklahoma	Ohio State	Penn State	Western Kentucky	Oklahoma	Oklahoma	Oklahoma
Ohio State	Oklahoma	Oklahoma	Wisconsin	Western Kentucky	Western Kentucky	Western Kentucky
Penn State	Wisconsin	Ohio State	Western Michigan	Penn State	Appalachian State	Appalachian State
Florida State	Penn State	Florida State	San Diego State	Wisconsin	Penn State	Penn State
Wisconsin	Florida State	Michigan	South Florida	Southern California	Wisconsin	Wisconsin
Michigan	Michigan	Wisconsin	Stanford	Ohio State	Southern California	Southern California
1.02%	0.99%	0.85%	**0.08%**	0.71%	1.08%	1.08%

In Tables 31-34, the total backedge weight percentages are shown for our rankings and external rankings on the 2016-17 college football season dataset. In these tests, we use rankings from Associated Press (AP), Bleacher Report, and ESPN. Our rankings were generated using our Berger/Shor algorithm implementation with node sorting by win-loss ratio (Y3) and outgoing edge weight decreasing (Y4), Hill Climbing implementation, and ROC Search algorithm.

Similar to our tests on the 2016-17 NFL dataset, at least one of our rankings outperforms each external ranking according to the total backedge weight metric. To make evaluation of the total backedge weight comparable between external rankings with the top-25 teams and our full rankings, we use rank snipping truncation for the top-25 teams in our rankings. However, because games between teams outside the top-25 are not considered after truncation, the total backedge weight values are much smaller as fewer edges and less edge weight are considered.

Overall, Y3 and Y4 perform the best in all Alpha and Beta combinations except (0,1), where the ranking from ROC Search has the lowest total backedge weight of those we evaluated. In some cases, Y3 and Y4 receive unusually low scores, such as 0.08% by Y3 with Alpha/Beta values of (0.5,0.5) and 0.04% by Y4 with Alpha/Beta values of (1,0). We suspect that this is because the ranking is optimized according to the total backedge weight metric: games between teams outside the top-25 are not considered, so these teams are placed in ranks which result in the minimum backedge weight from games between teams within the top-25. These low scores did surprise us, especially with Alpha/Beta values of (0.5,0.5), because no edges are removed from the decay or point differential modifiers. However, the difference in total backedge weight between our algorithms and external rankings is much smaller overall with Alpha/Beta values of (0.5,0.5), which we suspect allows for more accurate rank generation rather than optimization specifically for total backedge weight.

A summary of the total backedge weight percentages for the 2016-17 CFB dataset with our algorithms and existing rankings are shown in Figure 26.

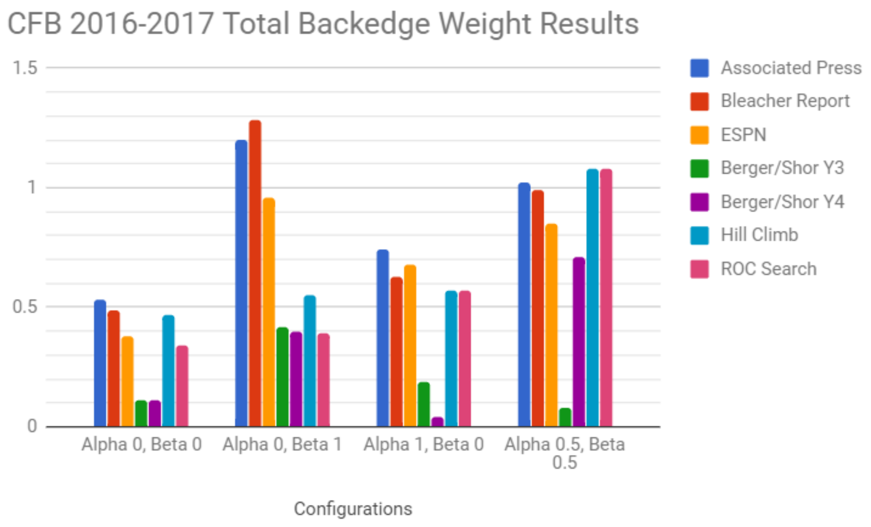


Figure 26: CFB 2016-2017 total backedge weight results

9.1.3: National Hockey League (NHL) Ranking Results

Table 35: Algorithm Comparisons on 2014-15 NHL (Alpha 0, Beta 0)

NHL.com	Y3	Y4	Hill Climbing	ROC Search
Rangers	Capitals	Rangers	Rangers	Rangers
Canadiens	Lightning	Canucks	Capitals	Capitals
Ducks	Rangers	Blues	Lightning	Lightning
Blues	Senators	Capitals	Wild	Blues
Lightning	Wild	Lightning	Blues	Senators
Predators	Blues	Jets	Jets	Wild
Blackhawks	Jets	Stars	Canadiens	Jets
Canucks	Canadiens	Canadiens	Senators	Canadiens
Capitals	Kings	Senators	Kings	Islanders
Islanders	Islanders	Wild	Islanders	Dallas Stars
22.16%	**18.35%**	20.09%	19.87%	18.80%

Table 36: Algorithm Comparisons on 2014-15 NHL (Alpha 0, Beta 1)

NHL.com	Y3	Y4	Hill Climb	ROC Search
Rangers	Rangers	Rangers	Rangers	Rangers
Canadiens	Blues	Blues	Blues	Blues
Ducks	Lightning	Wild	Ducks	Ducks
Blues	Ducks	Senators	Lightning	Lightning
Lightning	Wild	Ducks	Wild	Senators
Predators	Senators	Blue Jackets	Senators	Wild
Blackhawks	Canucks	Lightning	Blue Jackets	Capitals
Canucks	Canadiens	Canucks	Canucks	Blue Jackets
Capitals	Blue Jackets	Capitals	Canadiens	Canadiens
Islanders	Capitals	Blackhawks	Capitals	Canucks
24.19%	22.09%	**21.73%**	23.24%	22.69%

Table 37: Algorithm Comparisons on 2014-15 NHL (Alpha 1, Beta 0)

NHL.com	Y3	Y4	Hill Climbing	ROC Search
Rangers	Capitals	Lightning	Rangers	Capitals
Canadiens	Lightning	Rangers	Capitals	Rangers
Ducks	Rangers	Blackhawks	Lightning	Lightning
Blues	Blues	Blues	Blues	Blues
Lightning	Blackhawks	Canadiens	Blackhawks	Blackhawks
Predators	Senators	Canucks	Jets	Wild
Blackhawks	Wild	Capitals	Wild	Jets
Canucks	Jets	Penguins	Senators	Senators
Capitals	Kings	Jets	Kings	Canadiens
Islanders	Flames	Kings	Canadiens	Kings
22.40%	20.05%	**20.02%**	21.49%	22.00%

Table 38: Algorithm Comparisons on 2014-15 NHL (Alpha 0.5, Beta 0.5)

NHL.com	Y3	Y4	Hill Climbing	ROC Search
Rangers	Lightning	Rangers	Rangers	Rangers
Canadiens	Rangers	Blues	Blues	Blues
Ducks	Blues	Lightning	Lightning	Lightning
Blues	Canadiens	Canadiens	Canadiens	Canadiens
Lightning	Capitals	Canucks	Capitals	Capitals
Predators	Wild	Blackhawks	Wild	Ducks
Blackhawks	Blackhawks	Ducks	Blackhawks	Wild
Canucks	Ducks	Wild	Ducks	Blackhawks
Capitals	Senators	Capitals	Canucks	Canucks
Islanders	Canucks	Flames	Senators	Islanders
23.60%	**21.60%**	22.42%	23.04%	22.65%

Tables 35 to 38 show the results from our 2014-15 NHL dataset tests with differing Alpha and Beta values. In these tests, we utilized the NHL.com standings as our external ranking. These standings are based on win-loss ratio, which we expect may influence our results when compared to the power rankings utilized in the NFL and CFB tests. Additionally, we used our Berger/Shor approximation algorithm with node sorting by win-loss ratio (Y3) and by outgoing edge weight decreasing (Y4), as well our Hill Climb and ROC Search algorithms.

The tests in Tables 35 to 38 show that the total backedge weights are considerably high on average, especially when compared to the total backedge weights in the NFL tests. Additionally, in each Alpha/Beta configuration, all of our algorithms performed better than NHL.com's ranking, although we suspect this may be because standings were utilized in place of power rankings, so factors such as point differential were not considered. Either Y3 or Y4 performed the best in all cases, but we were surprised to see our ROC Search algorithm perform quite well, where it averaged slightly better than Hill Climb and beat Y4 with Alpha/Beta of (0,0).

When analyzing the teams in each ranking, we found that the majority of the teams placed in NHL.com top-10 standings were found in the top-10 rankings from our algorithms, with the exception of one or two teams. We also found that Alpha and Beta both have roughly equivalent impacts on the difference in minimum and maximum total backedge weight in our MLB rankings: Alpha/Beta of (0,0) have a difference of 3.81%, (0,1) have a difference of 2.46%, (1,0) have a difference of 2.38%, and (0.5,0.5) have a difference of 2%. We expect that this means similar weights from games are removed if the first set of games is removed with Alpha of 0 as if the set of games within one score is removed with Beta of 0.

A summary of the total backedge weight percentages for the 2014-15 NHL dataset with our algorithms and existing rankings are shown in Figure 27.

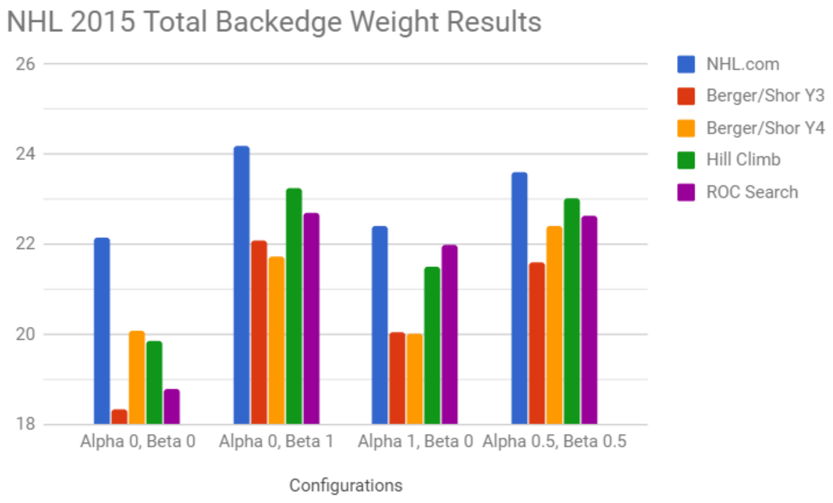


Figure 27: NHL 2014-2015 total backedge weight results

9.1.4: Major League Baseball (MLB) Ranking Results

Table 39: Algorithm Comparisons on 2015 MLB (Alpha 0, Beta 0)

Baseball-Reference	Y3	Y4	Hill Climbing	ROC Search
Cardinals	Blue Jays	Blue Jays	Blue Jays	Blue Jays
Pirates	Astros	Mets	Astros	Astros
Cubs	Cubs	Nationals	Cubs	Cubs
Royals	Giants	Indians	Giants	Mets
Blue Jays	Mets	Royals	Mets	Giants
Dodgers	Pirates	Giants	Pirates	Pirates
Mets	Nationals	Red Sox	Nationals	Nationals
Rangers	Rangers	Astros	Indians	Indians
Yankees	Indians	Rangers	Rangers	Rangers
Astros	Dodgers	Cubs	Dodgers	Dodgers
26.19%	25.36%	**24.98%**	25.06%	25.16%

Table 40: Algorithm Comparisons on 2015 MLB (Alpha 0, Beta 1)

Baseball-Reference	Y3	Y4	Hill Climbing	ROC Search
Cardinals	Blue Jays	Blue Jays	Cubs	Pirates
Pirates	Cubs	Cubs	Pirates	Cubs
Cubs	Pirates	Pirates	Blue Jays	Blue Jays
Royals	Cardinals	Cardinals	Cardinals	Cardinals
Blue Jays	Rangers	Rangers	Rangers	Rangers
Dodgers	Dodgers	Dodgers	Royals	Royals
Mets	Royals	Royals	Dodgers	Mets
Rangers	Mets	Mets	Mets	Dodgers
Yankees	Indians	Indians	Indians	Angels
Astros	Angels	Angels	Angels	Indians
26.52%	**25.88%**	26.08%	26.16%	26.23%

Table 41: Algorithm Comparisons on 2015 MLB (Alpha 1, Beta 0)

Baseball-Reference	Y3	Y4	Hill Climbing	ROC Search
Cardinals	Blue Jays	Blue Jays	Blue Jays	Blue Jays
Pirates	Astros	Royals	Astros	Astros
Cubs	Cardinals	Giants	Cardinals	Cardinals
Royals	Pirates	Mets	Giants	Giants
Blue Jays	Giants	Nationals	Pirates	Pirates
Dodgers	Dodgers	Indians	Dodgers	Dodgers
Mets	Royals	Astros	Royals	Royals
Rangers	Cubs	Yankees	Mets	Mets
Yankees	Mets	Cardinals	Cubs	Cubs
Astros	Nationals	Dodgers	Nationals	Nationals
26.73%	**26.31%**	26.62%	26.56%	26.56%

Table 42: Algorithm Comparisons on 2015 MLB (Alpha 0.5, Beta 0.5)

Baseball-Reference	Y3	Y4	Hill Climb	ROC Search
Cardinals	Blue Jays	Blue Jays	Blue Jays	Blue Jays
Pirates	Pirates	Pirates	Pirates	Pirates
Cubs	Cubs	Royals	Cubs	Cubs
Royals	Cardinals	Cubs	Cardinals	Cardinals
Blue Jays	Dodgers	Cardinals	Royals	Royals
Dodgers	Royals	Mets	Dodgers	Dodgers
Mets	Mets	Dodgers	Mets	Mets
Rangers	Astros	Rangers	Astros	Rangers
Yankees	Rangers	Indians	Rangers	Astros
Astros	Yankees	Yankees	Giants	Giants
26.93%	27.07%	**26.87%**	27.28%	26.90%

Tables 39 to 42 above show the results from our 2015 MLB dataset tests with differing Alpha and Beta values. In these tests, we utilized the baseball-reference.com’s standings as our external ranking. These standings are based on win-loss ratio, similar to NHL.com’s in our NHL tests, which we expect may influence our results when compared to power rankings. When testing, we used our Berger/Shor approximation algorithm with node sorting by win-loss ratio (Y3) and by outgoing edge weight decreasing (Y4), as well our Hill Climb and ROC Search algorithms.

Overall, scores for MLB rankings were much higher than other sports datasets we tested with, where many rankings contained more than 25% of the total graph weight as backedge weight. We expect that this is because the MLB graph is the densest graph in our test cases, with many games played between a small number of teams, such that a large number of backedges and backedge weight are unavoidable when generating rankings. Despite the high total backedge weights, at least one of our algorithms always outperformed the external ranking, and except for when Alpha/Beta was (0.5,0.5), all of our algorithms outperformed the external ranking. In general, either Y3 or Y4 performed the best, but only with an Alpha/Beta of (0,1) did Hill Climb or ROC Search not outperform either of the two variations.

Additionally, the total backedge weights of the rankings generated by our algorithms are much closer when compared to other data sets, which we suspect is also due to the graph density resulting in fewer opportunities to improve upon rankings. For example, with Alpha/Beta values of (0.5,0.5), the difference in minimum and maximum total backedge weight from all rankings we tested was only 0.41% of the total graph weight. We also found that our Hill Climb and ROC Search algorithms generated the same rankings with Alpha/Beta of (1,0).

A summary of the total backedge weight percentages for the 2015 MLB dataset with our algorithms and existing rankings are shown in Figure 28.

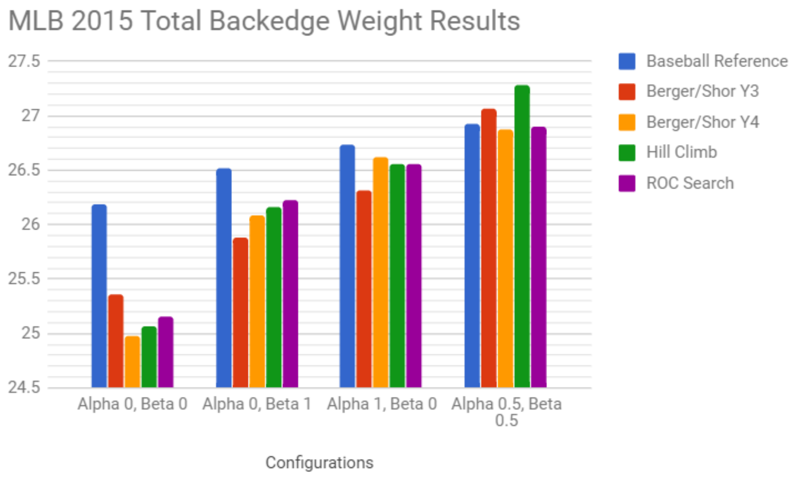


Figure 28: MLB 2015 total backedge weight results

9.2: Rank Differential Testing

The second full set of test cases conducted utilize the rank differential metric. Our test cases from the prior section demonstrate that our algorithms consistently outperform external rankings by the total backedge weight metric, but we wanted to determine how similar our rankings are to external rankings, which have consistent analysis and discussion on the ordering of teams before publication. As discussed in Section 2.3, the total backedge weight metric is rooted in our graph-based approach, but success with this metric is also dependent on weighing each edge appropriately. Therefore, comparison against external rankings and the factors they consider was a viable method to approximate our approach to weighing edges.

Given the nature of the rank differential metric, rank differentials need to be calculated from each test ranking to the control ranking, where several control rankings are present per sports dataset. Thus, we are not able to show each of the orderings and rank differentials generated, so we include a table of which Alpha and Beta values result in the lowest rank differential for our algorithms to the external rankings. Additionally, we provide heatmaps of the average rank differential which either align with the average trend of other algorithmically-generated rankings or are vastly different. Similarly to our external ranking heatmaps in Section 6.3, the heatmaps in this section show changes in Alpha on the vertical axis and changes in Beta on the horizontal axis. In each of our test cases, we utilize the normalized graph and enable quartile evaluation.

The first dataset we tested rank differential with is the 2016-17 NFL dataset. In this dataset, we compare with the ESPN, NFL.com, and Sports Illustrated power rankings. The best Alpha and Beta values to minimize our rank differential for each of our algorithms are shown in Table 43.

Table 43: 2016-17 NFL Rank Comparison Results (Alpha, Beta)

2016-17 NFL	Hill Climbing	Y3	Y4	ROC
ESPN	(1.0, 1.0)	(1.0, 0.5)	(0.5, 1.0)	(1.0, 1.0)
NFL.com	(0.75, 1.0)	(0.75, 1.0)	(0.5, 1.0)	(0.75, 0.75)
Sports Illustrated	(0.0, 0.75)	(0.75, 1.0)	(0.25, 1.0)	(0.25, 1.0)

As shown in Table 43, the best Alpha and Beta values to minimize rank differential for each external ranking from all of our algorithms are often both greater than 0.5 for ESPN and NFL.com, signifying that these rankings may not place as much emphasis on the recency of game and high point differentials as we expected when analyzing each ranking's heatmap. Sports Illustrated's best values were slightly different: it prefers small Alpha values but high Beta values. We suspect that SportsIllustrated places significant important on recency of game when generating their rankings as a result.

Two sample heatmaps of average rank differential for different Alpha and Beta values are shown in Figures 29 and 30. In both heatmaps, a clear trend is visible that medium to high Beta values are preferred and high Alpha values are preferred. Additionally, average rank differential increases significantly with small Alpha and Beta values, especially so when Beta is 0. We expect that this is because our algorithms continue to optimize against the total backedge weight metric, and that utilizing a Beta value of 0 results in the removal of games where the final point differential is one score or less, skewing the output ordering.

	0	0.25	0.5	0.75	1
0	5.6875	2.9375	2.1875	1.8125	1.875
0.25	6.3125	2.5625	1.6875	1.5625	1.5
0.5	6.625	2.875	1.5625	1.25	1.3125
0.75	6.625	3.375	1.625	**1.1875**	1.25
1	6.5	3.4375	1.75	1.625	1.5625

Figure 29: NFL.com vs. ROC Search rank differentials Alpha/Beta heatmap

	0	0.25	0.5	0.75	1
0	5.5625	3.375	2.75	2.5625	2.625
0.25	6.0625	3.0625	2.375	2.125	2.0625
0.5	6.5	2.9375	2.0625	1.8125	1.9375
0.75	6.625	3.1875	1.8125	1.6875	1.875
1	6.375	3.25	1.8125	1.6875	**1.625**

Figure 30: ESPN vs. Hill Climb rank differentials Alpha/Beta heatmap

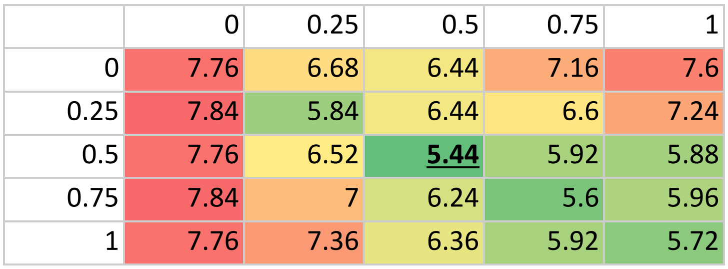
The second dataset we tested rank differential with is the 2016-17 NCAA College Football (CFB) Division I dataset. In this dataset, we compare with the Associated Press (AP) and Associated Press Coaches polls, Bleacher Report, ESPN, and Sonny Moore power rankings. The best Alpha and Beta values to minimize our rank differential for each of our algorithms are shown in Table 44.

Table 44: 2016-17 CFB Rank Comparison Results (Alpha, Beta)

2016-17 CFB	Hill Climbing	Y3	Y4	ROC
AP	(0.75, 1.0)	(0.5, 0.5)	(0.25, 0.75)	(0.5, 0.25)
Bleacher Report	(0.75, 1.0)	(1.0, 1.0)	(0.25, 0.75)	(0.5, 0.25)
AP Coaches	(0.75, 1.0)	(0.5, 0.5)	(0.25, 0.75)	(0.5, 0.25)
ESPN	(0.5, 0.25)	(0.5, 0.5)	(0.0, 0.25)	(0.5, 0.25)
Sonny Moore	(0.75, 0.75)	(0.5, 0.5)	(0.5, 0.5)	(0.5, 0.25)

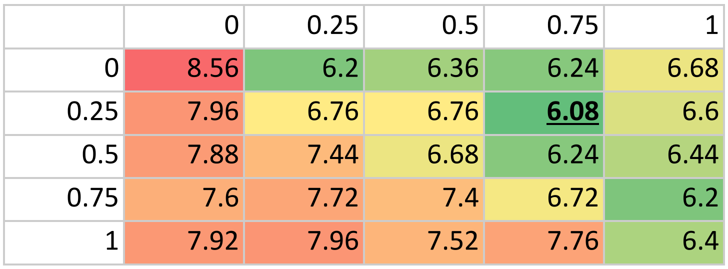
Table 44 shows much more variance in best Alpha and Beta values for the external rankings when compared to the 2016-17 NFL results in Table 43. Two interesting trends shown are that ROC Search always has the lowest rank differentials for all external rankings with Alpha/Beta values of (0.5,0.25), and that Hill Climb almost always has the lowest rank differentials for external rankings with an Alpha/Beta of (0.75,1.0). Additionally, Y4 appears to favor small Alpha and Beta values while Y3 prefers Alpha/Beta values near 0.5. Overall, both Associated Press polls, Bleacher Report, and Sonny Moore prefer mid-range Alpha and Beta values, while ESPN prefers lower values on average. This implies that ESPN's power ranking favors higher-scoring and recent games more than the other rankings.

The heatmaps in Figures 31 and 32 show the two trends that each algorithm tended to follow during the rank differential tests. Compared to the heatmaps from our 2016-17 NFL dataset, the lower rank differentials are far more defined in the center of the heatmaps. However, both still show that low Beta values increase rank differential significantly, and that high Alpha and Beta values tend to reduce rank differential. Additionally, when compared to the 2016-17 NFL rank differential heatmaps, the lowest average rank differential for this set is far greater, though the highest average rank differential is only slightly higher.



	0	0.25	0.5	0.75	1
0	7.76	6.68	6.44	7.16	7.6
0.25	7.84	5.84	6.44	6.6	7.24
0.5	7.76	6.52	**5.44**	5.92	5.88
0.75	7.84	7	6.24	5.6	5.96
1	7.76	7.36	6.36	5.92	5.72

Figure 31: Associated Press vs. Berger/Shor (Y3) rank differentials Alpha/Beta heatmap



	0	0.25	0.5	0.75	1
0	8.56	6.2	6.36	6.24	6.68
0.25	7.96	6.76	6.76	**6.08**	6.6
0.5	7.88	7.44	6.68	6.24	6.44
0.75	7.6	7.72	7.4	6.72	6.2
1	7.92	7.96	7.52	7.76	6.4

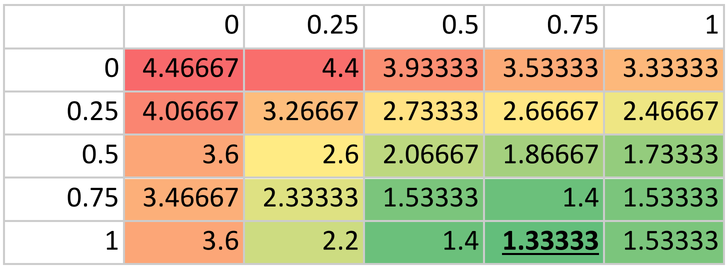
Figure 32: Bleacher Report vs. Berger/Shor (Y4) rank differentials Alpha/Beta heatmap

Table 45 shows the third set of rank differential tests, which were conducted on our 2014-15 NHL dataset. In this test case, we show the Alpha and Beta values for the lowest average rank differential on the NHL.com standings.

Table 45: 2014-15 NHL Rank Comparison Results (Alpha, Beta)

2014-15 NHL	Hill Climbing	Y3	Y4	ROC
NHL.com	(1.0, 0.75)	(1.0, 0.75)	(0.75, 0.75)	(1.0, 0.75)

In Table 45, there is minimal variation between algorithms over which Alpha and Beta values result in the lowest average rank differential. In all cases except Y4, the best combination is (1.0,0.75); in the case of Y4, this combination is similar at (0.75,0.75). We can infer from this that these standings do not assign any value to the recency of game, and tend to not apply much more weight for a large point differential compared to a small one. This trend manifests itself in the heatmap in Figure 33, which also suggests that higher Alpha and Beta values result in lower average rank differentials. From this, we can infer that lower Alpha and Beta values assign more edge weight to backedges of this ranking, which are more recent and higher-scoring games. We expect that this is because our ranking was retrieved from NHL.com standings rather than a power ranking.



	0	0.25	0.5	0.75	1
0	4.46667	4.4	3.93333	3.53333	3.33333
0.25	4.06667	3.26667	2.73333	2.66667	2.46667
0.5	3.6	2.6	2.06667	1.86667	1.73333
0.75	3.46667	2.33333	1.53333	1.4	1.53333
1	3.6	2.2	1.4	**1.33333**	1.53333

Figure 33: NHL.com vs. Hill Climb rank differentials Alpha/Beta heatmap

The final rank differential tests we conducted were on the 2015 MLB dataset. The results from our rank comparison with baseball-reference.com's standings are shown in Table 46.

Table 46: 2015 MLB Rank Comparison Results (Alpha, Beta)

2015 MLB	Hill Climbing	Y3	Y4	ROC
Baseball Reference	(0.75, 1.0)	(0.75, 1.0)	(0.75, 1.0)	(0.75, 1.0)

Table 46 shows that the best Alpha and Beta values for minimizing the average rank differential for this ranking is (0.75,1.0). An interesting pattern is that this combination of values produces the lowest average rank differential for all of our algorithms. Additionally, the preference for higher values of Alpha and Beta aligns with the rank differential tests of all other datasets we examined. The heatmap in Figure 34 highlights this trend, where the highest Alpha and Beta values result in the lowest rank differentials. However, unlike the NHL ranking, this ranking appears to perform better with smaller Alpha values than smaller Beta values, indicating that the removal of all games within one score reduced much of the overlapping forward edge weight our ranking shared with the external ranking.

	0	0.25	0.5	0.75	1
0	5.06667	3.93333	2.86667	2.33333	2.46667
0.25	5.2	3.06667	2	2.06667	1.8
0.5	4.86667	3.06667	2.06667	1.2	1.66667
0.75	5.2	2.93333	2.2	1.06667	**0.73333**
1	5.2	3.6	2.66667	0.93333	**0.73333**

Figure 34: Baseball Reference vs. Berger/Shor (Y4) rank differentials Alpha/Beta heatmap

9.3: Summary

In this chapter, we discussed all of the test cases conducted on our rankings and external rankings on four sports seasons datasets: 2016-17 National Football League (NFL) regular season, 2016-17 College Football (CFB) full season, 2014-15 National Hockey League (NHL) regular season, and 2015 Major League Baseball (MLB) regular season. We introduced our total backedge weight metric test cases, where we tested each of our algorithms with four different Alpha and Beta combinations to determine where our algorithms performed the best. In all of these test cases, at least one of our algorithms outperformed every external ranking tested based on our total backedge weight metric. However, some of the best performances of our algorithms occurred when both Alpha and Beta were zero, which resulted in optimization against this metric rather than accuracy of ranking.

Additionally, we conducted rank differential test cases in the interest of generating rankings similar to external rankings to investigate similarities and trends between different Alpha and Beta values. In these test cases, we analyzed trends across sports and different rankings to determine if a correlation existed between certain Alpha and Beta values and improved accuracy within a given sport. In general, we found that, across the sports datasets we tested, Alpha and Beta values of greater than 0.5 each resulted in the lowest average rank differential.

Chapter 10: Conclusion

With the spread of technology and communication, people around the world are now able to further share their interests in sports and sports rankings with each other instantaneously. New forms of analytics, power rankings, and blogs continue to thrive and spark discussion among sports fanatics and enthusiasts about which teams played well over the past weeks and their speculations of upcoming games. The field of sports rankings remains a vibrant topic as people and systems tend to analyze metrics and outcomes of games in different manners, internally weighing factors uniquely to output an overall ranking of teams within sports leagues.

In this project, we explored sports rankings using a graph-based approach, in contrast to a more traditional formulaic approach. This graph-based approach applies a season of sports data to a graph data structure, where each team is a node, and each game is a directed edge between two nodes, pointing from the winning node to the losing node. The foundation of the graph-based approach is to approximate the Minimum Feedback Arc Set for each sports dataset, or the set of directed edges in our sports graph which remove cycles or upsets from the graph with as little edge weight as possible. Thus, the most accurate rankings for sports datasets would minimize the total edge weight of all backedges, or edges of upsets and disagreements with the rankings, which is found through the minimum feedback arc set. However, since the Minimum Feedback Arc Set problem is NP-Complete and because most computer scientists believe that $P \neq NP$, it is not computationally feasible to generate these rankings. Therefore, approximations to the solutions are utilized instead.

Over the course of this project, we provide several contributions to rank generation in the context of sports rankings. Specifically, we:

1. apply graph theory and the Minimum Feedback Arc Set problem to sports rankings with a unique approach different from the more common formulaic ranking approach,
2. implement rank generation through a brute force approach and three approximation algorithms for the Minimum Feedback Arc Set problem,
3. reverse-engineer sports rankings published online utilizing our graph data structures, and
4. outperform existing sports rankings using our own algorithms according to our evaluation metric.

We provide each of the above contributions through the different methodologies and test cases we conducted during this project. Our first contribution of graph theory application to sports rankings was completed through our program, approximation algorithms, and edge weight algorithms which convert raw sports season data into a graph data structure internally. We accomplished our second contribution of algorithm implementation through background research, design, testing, and optimizing four algorithms to generate and approximate rankings, one of which was developed using our own evaluation metric of Range of Correctness. The third contribution of reverse-engineering external rankings for this project was performed by our Alpha/Beta heatmap and rank differential test cases, which allowed us to determine which of our edge weight factors influenced external rankings the most, and which factors brought our own algorithmically-generated rankings closest to the external rankings. Our final contribution of outperforming existing rankings was completed through the correctness testing of our algorithms against external rankings on datasets from various sports seasons using the total backedge weight concept as our evaluation metric.

From the test cases conducted during our project, we came to three conclusions. The first conclusion we came to was that our algorithms tended to optimize more effectively for the total backedge weight metric in sparse graphs and small Alpha and Beta values, where significant amounts of edge weight in the graph could be removed. Secondly, we deduced that, even with Alpha and Beta values optimized for external rankings, we were still able to outperform them. Finally, we concluded that overall, external rankings tend to favor Alpha and Beta values of greater than 0.5, indicating reduced consideration of the recency of games and margin of victory.

10.1: Future Work

During this project, several topics came up in discussion that we did not have the chance to explore further. The first of these topics is research into additional factors that can be considered when weighing edges. Our approach considered point differential, strength of schedule, and recency of game, which allowed for flexibility across many sports, but by discounting other factors we sacrificed accuracy. We recommend further investigation into program logic for classification of sports to account for different factors dynamically depending on which sport is provided as input, and for different factors such as roster changes and importance of playoff games to be considered. Furthermore, we suggest research into applying exponential decay with recency of games, as this could further increase accuracy when analyzing rankings.

The second topic we did not have time to explore was modeling sports where more than two players or teams participated in a match, such as swimming or track and field. When modeling two-team sports, such as football or hockey, the Minimum Feedback Arc Set problem can be applied by treating each game as a directed edge between two nodes representing the teams. However, when considering matches with more than two teams, a hypergraph-based approach may need to be considered, as an edge for one match would need to connect more than two nodes. We are unsure of how this would be modeled or if the Minimum Feedback Arc Set problem could be applied to hypergraphs, but we think that investigation into this problem in the interest of analyzing different sports could yield great benefits.

Additionally, we did not have time to explore alternative heuristics for evaluating external rankings outside of total backedge weight and average rank differential. We are skeptical of the utility of our metrics for use in better understanding external rankings. The Alpha/Beta optimization for total backedge weight seemed to favor values where as much edge weight could be minimized as possible, leading to Alpha/Beta values that did not reflect the priorities of the external ranking. Furthermore, since our algorithms frequently outperformed external rankings on the heuristic of total backedge weight, we cannot accurately apply this metric even with the optimal

Alpha/Beta configuration for external rankings determined by rank differential because the total backedge weight metric optimizes towards reducing edge weight where possible. We recommend further exploration into alternative ways of measuring accuracy in rankings to address this.

Finally, we suggest testing with a wider range of sports seasons in the future. Our test cases consisted mostly of recent NFL, NHL, MLB, and college football datasets. We think that further data and external ranking collection could allow for deeper analysis into the underlying factors of these rankings, as trends could be analyzed over several years of rankings. Additionally, collection of data and rankings for other league sports could provide great insight on how ranking factors vary across sports.

Bibliography

[1] Berger, B., and Shor, P. W. (1990, January). Approximation algorithms for the maximum acyclic subgraph problem. In Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms (SODA '90). Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 236-243.

[2] Bleacher Report. (n.d.). Retrieved February 17, 2018, from http://bleacherreport.com/

[3] Bresiger, G. (2015, September 5). Nearly 75M people will play fantasy football this year. Retrieved January 31, 2018, from https://nypost.com/2015/09/05/nearly-75m-people-will-play-fantasy-football-this-year/

[4] Burke, C., Jacobs, M., and Bedard, G.A. Week 17 Power Rankings: How all 32 teams end 2016, SI.com. (2016). https://www.si.com/nfl/2016/12/28/nfl-power-rankings-week-17-standings-playoffs

[5] Butterfield, A., and Ngondi, G. (Eds.). (2016). A Dictionary of Computer Science (7th ed.) [2016]. Retrieved February 2, 2018, from http://www.oxfordreference.com/view/10.1093/acref/9780199688975.001.0001/acref-9780199688975-e-6317

[6] Charbit, P., Thomassé, S., and Yeo, A. The Minimum Feedback Arc Set Problem is NP-hard for Tournaments. Combinatorics, Probability and Computing, Cambridge University Press (CUP), 2007, 16, pp.01-04. . <10.1017/S0963548306007887>. Found from: https://hal.inria.fr/file/index/docid/140321/filename/feedback.pdf

[7] Chen, J., Liu, Y., Lu, S., O'Sullivan, B., and Razgon, I. (2008). A fixed-parameter algorithm for the directed feedback vertex set problem. J. ACM 55, 5, Article 21 (November 2008), 19 pages. DOI=http://dx.doi.org/10.1145/1411509.1411511, accessed February 2nd, 2018

[8] Fortnow, L. (2009, September). The status of the P versus NP problem. *Communications of the ACM*, 52(9), 78-86. doi:10.1145/1562164.1562186

[9] Glickman, M. E., "A Comprehensive Guide to Chess Ratings"(1995). A subsequent version of this paper appeared in the American Chess Journal, 3, pp. 59--102., accessed 2/10/2018, http://glicko.net/research/acjpaper.pdf

[10] Glickman, M. E., and Jones, A. C., "Rating the chess rating system" (1999), Chance, 12, 2, 21-28, http://glicko.net/research/chance.pdf

[11] Heitner, D. (2015, October 19). Sports Industry To Reach $73.5 Billion By 2019. Retrieved January 31, 2018, from https://www.forbes.com/sites/darrenheitner/2015/10/19/sports-industry-to-reach-73-5-billion-by-2019/#6793ab121b4b

[12] Karp, R. M. (1972) Reducibility among Combinatorial Problems. In: Miller R. E., Thatcher J. W., Bohlinger J. D. (eds) Complexity of Computer Computations. The IBM Research Symposia Series. Springer, Boston, MA

[13] League Index. (n.d.). Retrieved February 17, 2018, from https://www.hockey-reference.com/leagues/

[14] MLB Stats, Scores, History, & Records. (n. d.). Baseball-Reference.com, Retrieved October 11, 2017, from www.baseball-reference.com/

[15] Moore, S. (n.d.). Sonny Moore. Retrieved February 17, 2018, from http://sonnymoorepowerratings.com/archive.htm

[16] NBC Sports. (2018, February 15). Retrieved February 17, 2018, from http://www.nbcsports.com

[17] Newell, A. *What is Elo? An explanation for competitive gaming's hidden rating system*. Dot Esports, Retrieved 27 Jan. 2018, from www.dotesports.com/general/news/elo-ratings-explained-20565#list-1

[18] *Notes on the Complexity of Search* [PDF]. (2003, September 23). Massachusetts Institute of Technology. Retrieved February 29th, 2018 from: http://www.ai.mit.edu/courses/6.034b/searchcomplex.pdf

[19] Official Site of the National Football League. NFL.com. Retrieved February 17, 2018, from https://www.nfl.com/

[20] Official Site of the National Hockey League. NHL.com, National Hockey League, www.nhl.com/

[21] Paine, N. (n.d.). Five Thirty Eight. Retrieved February 17, 2018, from https://fivethirtyeight.com/sports/

[22] Rosen, K. H. (2012). Discrete Mathematics and Its Applications (7th ed). New York, NY: McGraw-Hill.

[23] Russell, S and Norvig, P. (2010). *Artificial Intelligence: A Modern Approach (3rd ed).* Upper Saddle River, NJ: Pearson.

[24] Sorensen, S. P. "An Overview of Some Methods for Ranking Sports Teams." pp. 1–14., www.phys.utk.edu/sorensen/ranking/Documentation/Sorensen_documentation_v1.pdf .

[25] Sports Illustrated. (n.d.). Retrieved February 17, 2018, from https://www.si.com/

[26] Tarjan, R. (1972). Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, *1*(2), 155-159.10.1137/0201010

[27] The Worldwide Leader in Sports. ESPN.com. Retrieved February 17, 2018, from http://www.espn.com/

[28] "What Is an Elo Rating?" What Is an ELO Rating in Chess, The Chess Piece, www.thechesspiece.com/what_is_an_elo_rating.asp.

[29] World Football Elo Ratings. Eloratings.net, Retrieved 1 May 2015, from www.eloratings.net/about.

[30] Younger, D., "Minimum Feedback Arc Sets for a Directed Graph," in IEEE Transactions on Circuit Theory, vol. 10, no. 2, pp. 238-245, Jun 1963. doi: 10.1109/TCT.1963.1082116 keywords: {Feedback loop;Impedance;Switching circuits;Terminology;Topology;Tree graphs}, URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1082116&isnumber=23378

Appendix A: Glossary

- **Graph**: A data structure consisting of V, a nonempty set of vertices (or nodes) and E, a set of edges
- **Vertex / Node**: An entity or datapoint within a graph
- **Edge**: An association between one or two vertices
- **Endpoint**: A vertex associated with an edge
- **Unweighted graph**: A graph whose edges do not contain edge weights
- **Weighted graph**: A graph whose edges each have a numerical value as an edge weight
- **Directed graph**: A graph whose edge set contains directed edges
- **Directed edge / Arc**: An edge associated with an ordered pair of vertices, where the first vertex is the starting vertex and the second vertex is the ending vertex
- **Indegree**: The number of edges with a given vertex as its ending vertex
- **Outdegree**: The number of edges with a given vertex as its starting vertex
- **Path**: A sequence of edges $(x_0, x_1), (x_1,x_2), (x_2,x_3), \ldots , (x_{n-1}, x_n)$ in G, where n is a nonnegative integer, and $x_0 = A$ and $x_n = B$
- **Cycle**: A path of length $n \geq 1$ that begins and ends at the same vertex
- **Cyclic graph**: A graph that contains one or more cycles
- **Acyclic graph**: A graph that does not contain cycles
- **Ranking**: A unique ordering of some collection of entities, with an implied hierarchy of which entities are better than others based upon a given comparison metric
- **Forward edge**: An edge that agrees with a ranking, whose existence indicates that the ordering of one node above another is correct
- **Backward edge / Backedge**: An edge that disagrees with the ranking, whose existence indicates that the ordering of two nodes may be incorrect
- **Strongly connected**: The property of a graph where there is a sequence of directed edges from any vertex in the graph to all other vertices in the graph
- **Strongly connected component**: A subgraph of a directed graph G that is strongly connected but not contained in larger strongly connected subgraphs
- **Total backedge weight**: The sum of all backedges for a ranking for a graph
- **Rank differential**: A comparative metric measuring the average difference in rank of each team between a control ranking and a test ranking
- **Control ranking**: The ranking in a rank differential used as the basis for comparison
- **Test ranking**: The ranking in a rank differential to compare with
- **Adjacency matrix**: "the n x n zero-one matrix with 1 as its (i, j)th entry when v_i and v_j are adjacent, and 0 as its (i, j)th entry when they are not adjacent" [22]
- **Alpha**: The factor in our edge weight algorithm which represents recency of game and linear decay
- **Beta**: The factor in our edge weight algorithm which represents the edge weight before normalization to total graph weight

- **Beta Normalization**: The process of assigning edge weights based on the beta interval in which the point differential is contained
- **Beta Interval**: The number of points assigned to each range in Beta Normalization
- **Heatmap**: A matrix of total backedge weight values for a given ranking where the cells are colored in varying gradients based on their value, with the horizontal axis indicating changes in Alpha and the vertical axis indicating changes in Beta
- **Range of Correctness**: An exclusive range for a ranking where the lower-bound index is of the lowest-ranked team it lost to, and the upper-bound index is of the highest-ranked team that it won against.
- **Bounds (Range of Correctness)**: The inclusive uppermost or lowermost position in the ranking whereby any swaps of a team's ranking within each bound would result in the same or reduced total backedge weight for a ranking
- **Rank pool**: A pool data structure which rearranges teams in a ranking based on their Range of Correctness
- **Relevant edges (Rank pool)**: The edges dictating the upper and lower bounds for a team's Range of Correctness
- **CFB**: Abbreviation for "College Football"

Appendix B: Alpha/Beta Heatmap Testing (cont.)

In this Appendix, we present all of the Alpha/Beta heatmaps created and analyzed but not included in our results. The analysis of the heatmaps in our results can be found in Section 6.3.

	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
0	0.164039	0.1628	0.162039	0.161525	0.161154	0.160874	0.160655	0.160479	0.160334	0.160213	0.160111
0.1	0.163211	0.162493	0.162054	0.161758	0.161545	0.161384	0.161259	0.161158	0.161076	0.161006	0.160948
0.2	0.162508	0.162233	0.162066	0.161954	0.161873	0.161812	0.161764	0.161726	0.161695	0.161668	0.161646
0.3	0.161906	0.162012	0.162077	0.16212	0.162151	0.162174	0.162193	0.162207	0.162219	0.162229	0.162238
0.4	0.161382	0.16182	0.162086	0.162263	0.162391	0.162486	0.162561	0.162621	0.16267	0.162711	0.162745
0.5	0.160924	0.161653	0.162093	0.162388	0.162599	0.162757	0.162881	0.16298	0.163061	0.163128	0.163185
0.6	0.160519	0.161506	0.1621	0.162498	0.162782	0.162995	0.163161	0.163294	0.163403	0.163494	0.163571
0.7	0.160159	0.161375	0.162106	0.162595	0.162944	0.163205	0.163409	0.163572	0.163706	0.163817	0.163911
0.8	0.159836	0.161258	0.162112	0.162681	0.163088	0.163393	0.16363	0.16382	0.163975	0.164104	0.164214
0.9	0.159546	0.161153	0.162117	0.162759	0.163217	0.16356	0.163827	0.164041	0.164216	0.164362	0.164485
1	**0.159283**	0.161058	0.162121	0.162829	0.163333	0.163712	0.164006	0.164241	0.164433	0.164593	0.164729

Figure 35: 2016-17 NFL heatmap (ESPN, Quartiles On)

The heatmap shown in Figure 35 represents the total backedge weights with different Alpha/Beta combinations of ESPN's power ranking for the 2016-17 NFL dataset. Unlike most other heatmaps, this heatmap shows two separate combinations which scored well: the lowest total backedge weight with Alpha/Beta values of (1,0), but also low backedge weight with values of (0,1). However, if Alpha and Beta were both small, which applies minimal values to the earliest and closest games in the season, the total backedge weight was quite high; further, if Alpha and Beta were both large, which treats recency of game and point differential as less important than the wins and losses themselves, the total backedge weight was nearly equivalent. We suspect that the favoring of either Alpha or Beta having a small value while the other has a large value is dependent on season specific data, where ESPN's power ranking maintains team placement such that discounting early games or discounting close games resulted in the reduction of "equivalently-weighted" backedges.

	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
0	0.159785	0.158517	0.15774	0.157214	0.156834	0.156548	0.156324	0.156144	0.155996	0.155872	**0.155767**
0.1	0.165025	0.163104	0.16193	0.161139	0.160569	0.16014	0.159804	0.159535	0.159313	0.159129	0.158972
0.2	0.169467	0.166973	0.165455	0.164433	0.163699	0.163146	0.162714	0.162368	0.162084	0.161847	0.161646
0.3	0.173281	0.17028	0.168459	0.167237	0.16636	0.165699	0.165184	0.164771	0.164433	0.164151	0.163911
0.4	0.176591	0.17314	0.171052	0.169652	0.168649	0.167895	0.167307	0.166835	0.166449	0.166127	0.165855
0.5	0.179491	0.175637	0.173311	0.171755	0.17064	0.169802	0.16915	0.168627	0.168199	0.167842	0.16754
0.6	0.182052	0.177837	0.175298	0.173601	0.172387	0.171475	0.170766	0.170197	0.169732	0.169345	0.169016
0.7	0.184331	0.179789	0.177058	0.175236	0.173933	0.172955	0.172194	0.171585	0.171086	0.170671	0.170319
0.8	0.186371	0.181533	0.178629	0.176693	0.17531	0.174272	0.173465	0.172819	0.172291	0.171851	0.171478
0.9	0.188209	0.183101	0.18004	0.178	0.176544	0.175453	0.174604	0.173925	0.17337	0.172907	0.172516
1	0.189873	0.184518	0.181313	0.17918	0.177658	0.176517	0.17563	0.174921	0.174341	0.173858	0.17345

Figure 36: 2016-17 NFL heatmap (Sports Illustrated, Quartiles On)

The heatmap shown in Figure 36 represents the total backedge weights with different Alpha/Beta combinations of Sports Illustrated's power ranking for the 2016-17 NFL dataset. In this heatmap, the Alpha/Beta value with the lowest total backedge weight was (0,1). From this data point, we can infer that this power ranking favors games more based on their recency than on the point differential of the game, as a Beta value of 1 only considers wins instead of point differential and an Alpha of 0 allows for the maximum decay of earliest games possible. An interesting trend from this heatmap is that the total backedge weight increases significantly more if increasing Alpha when Beta is small. We suspect that this is because some edges which support Sports Illustrated's ranking have less weight than backedges with smaller Alpha/Beta values compared to when Beta is larger.

	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
0	**<u>0.005297</u>**	0.006666	0.007749	0.008628	0.009354	0.009966	0.010487	0.010937	0.011329	0.011673	0.011979
0.1	0.005759	0.007012	0.008003	0.008808	0.009473	0.010033	0.010510	0.010922	0.011282	0.011597	0.011877
0.2	0.006114	0.007277	0.008199	0.008946	0.009564	0.010085	0.010529	0.010912	0.011245	0.011539	0.011799
0.3	0.006394	0.007488	0.008353	0.009056	0.009637	0.010126	0.010543	0.010903	0.011217	0.011493	0.011738
0.4	0.006622	0.007658	0.008479	0.009145	0.009696	0.010159	0.010555	0.010896	0.011194	0.011455	0.011687
0.5	0.006811	0.007800	0.008583	0.009218	0.009744	0.010187	0.010564	0.010890	0.011174	0.011424	0.011646
0.6	0.006970	0.007919	0.008670	0.009280	0.009785	0.010210	0.010572	0.010885	0.011158	0.011398	0.011611
0.7	0.007105	0.008020	0.008745	0.009333	0.009820	0.010230	0.010579	0.010881	0.011144	0.011376	0.011581
0.8	0.007222	0.008108	0.008809	0.009379	0.009850	0.010247	0.010585	0.010878	0.011132	0.011356	0.011555
0.9	0.007324	0.008184	0.008866	0.009419	0.009877	0.010262	0.010591	0.010874	0.011122	0.01134	0.011533
1	0.007414	0.008252	0.008915	0.009454	0.009900	0.010275	0.010595	0.010872	0.011113	0.011325	0.011513

Figure 37: 2016-17 CFB heatmap (Associated Press, Quartiles On)

In Figure 37, we show the heatmap of Associated Press' poll for the 2016-17 NCAA College Football (CFB) Division I dataset. In general, this heatmap shows that this ranking has low total backedge weights from small Alpha and Beta values, with the best total backedge weight occuring with Alpha/Beta of (0,0). Overall, increases in Alpha for this heatmap result in minimal increases in total backedge weight, while increases in Beta result in substantial increases in total backedge weight. We can infer from this that point differential of the game has more of an influence than recency of game, as increases in edge weight for close games result in more backedge weight. One interesting trend of this heatmap is how, for high Beta values, increases in Alpha reduce the total backedge weight. We suspect this follows the same reasoning as above, where earlier games are more likely to be forward edges, so increasing their edge weights through Alpha increases the total forward edge weight more than the total backedge weight.

	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
0	**<u>0.005297</u>**	0.006464	0.007387	0.008135	0.008754	0.009275	0.009719	0.010103	0.010437	0.010730	0.010991
0.1	0.005759	0.006824	0.007667	0.008351	0.008917	0.009393	0.009799	0.010149	0.010455	0.010723	0.010961
0.2	0.006114	0.007101	0.007883	0.008517	0.009042	0.009483	0.009860	0.010185	0.010468	0.010718	0.010938
0.3	0.006394	0.007320	0.008054	0.008649	0.009141	0.009555	0.009908	0.010213	0.010479	0.010713	0.010920
0.4	0.006622	0.007498	0.008192	0.008755	0.009221	0.009613	0.009948	0.010237	0.010488	0.01071	0.010906
0.5	0.006811	0.007646	0.008307	0.008844	0.009288	0.009662	0.009980	0.010256	0.010496	0.010707	0.010894
0.6	0.006970	0.007770	0.008404	0.008918	0.009344	0.009702	0.010008	0.010272	0.010502	0.010704	0.010883
0.7	0.007105	0.007876	0.008486	0.008982	0.009392	0.009737	0.010031	0.010285	0.010507	0.010702	0.010875
0.8	0.007222	0.007967	0.008557	0.009036	0.009433	0.009767	0.010052	0.010297	0.010512	0.010700	0.010867
0.9	0.007324	0.008047	0.008620	0.009084	0.009469	0.009793	0.010069	0.010308	0.010516	0.010698	0.010861
1	0.007414	0.008117	0.008674	0.009126	0.009501	0.009816	0.010085	0.010317	0.010519	0.010697	0.010855

Figure 38: 2016-17 CFB heatmap (Coaches, Quartiles On)

In Figure 38, we show the heatmap of Associated Press' Coaches polls for the 2016-17 NCAA College Football (CFB) Division I dataset. This ranking aligns with the standard Associated Press poll shown in the heatmap in Figure 37, from favoritism towards small Alpha/Beta to decreases in total backedge weight when increasing the Alpha value with a high Beta value. We suspect that the reasoning behind these trends matches that of the standard Associated Press poll, where backedges are more likely to be close games than earlier games. Therefore, increasing Beta is more likely to create large amounts of backedge weight, while increasing Alpha creates backedge weight and forward edge weight, decreasing the total backedge weight in some combinations.

	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
0	**<u>0.004868</u>**	0.006485	0.007764	0.008802	0.009661	0.010383	0.010998	0.011530	0.011993	0.012400	0.012761
0.1	0.005174	0.006657	0.007831	0.008784	0.009571	0.010234	0.010799	0.011287	0.011712	0.012086	0.012418
0.2	0.005410	0.006790	0.007883	0.008769	0.009503	0.010120	0.010647	0.011101	0.011497	0.011845	0.012154
0.3	0.005596	0.006895	0.007924	0.008758	0.009449	0.010030	0.010525	0.010953	0.011326	0.011654	0.011945
0.4	0.005748	0.006980	0.007957	0.008749	0.009405	0.009956	0.010427	0.010833	0.011187	0.011499	0.011775
0.5	0.005873	0.007051	0.007984	0.008741	0.009368	0.009895	0.010345	0.010734	0.011072	0.011370	0.011634
0.6	0.005978	0.007111	0.008007	0.008735	0.009337	0.009844	0.010277	0.010650	0.010975	0.011262	0.011515
0.7	0.006068	0.007161	0.008027	0.00873	0.009311	0.009801	0.010218	0.010579	0.010893	0.011169	0.011414
0.8	0.006146	0.007205	0.008044	0.008725	0.009288	0.009763	0.010168	0.010517	0.010822	0.011089	0.011327
0.9	0.006214	0.007243	0.008059	0.008721	0.009269	0.009730	0.010123	0.010463	0.010759	0.011020	0.011251
1	0.006273	0.007277	0.008072	0.008717	0.009251	0.009701	0.010085	0.010416	0.010705	0.010959	0.011184

Figure 39: 2016-17 CFB heatmap (Bleacher Report, Quartiles On)

Figure 39 shows Bleacher Report's ranking of the 2016-17 NCAA College Football (CFB) Division I dataset. This ranking also produces lower total backedge weights with small Alpha and Beta values, with the lowest total backedge weight occurring with Alpha/Beta of (0,0). As shown in the heatmap, this ranking also has much greater increases in total backedge weight with higher Beta values than Alpha values, which we suspect is due to the favoritism of higher scoring games over older games. However, this ranking appears to align with many older games, as the total backedge weight for a high Beta value decreases substantially as Alpha increases.

	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
0	**<u>0.004653</u>**	0.005597	0.006344	0.006950	0.007452	0.007873	0.008233	0.008543	0.008813	0.009051	0.009262
0.1	0.005421	0.006293	0.006983	0.007543	0.008006	0.008396	0.008728	0.009015	0.009265	0.009485	0.009680
0.2	0.006011	0.006828	0.007474	0.007999	0.008433	0.008798	0.009110	0.009378	0.009613	0.009819	0.010002
0.3	0.006478	0.007251	0.007863	0.008360	0.008771	0.009117	0.009412	0.009666	0.009888	0.010083	0.010256
0.4	0.006857	0.007595	0.008179	0.008653	0.009045	0.009375	0.009657	0.009900	0.010112	0.010298	0.010464
0.5	0.007171	0.007880	0.008441	0.008896	0.009273	0.009590	0.009860	0.010094	0.010297	0.010476	0.010635
0.6	0.007436	0.008119	0.008661	0.009100	0.009464	0.009770	0.010031	0.010257	0.010453	0.010626	0.01078
0.7	0.007661	0.008324	0.008849	0.009275	0.009627	0.009924	0.010177	0.010396	0.010587	0.010754	0.010903
0.8	0.007855	0.008500	0.009011	0.009425	0.009768	0.010057	0.010303	0.010516	0.010702	0.010865	0.011009
0.9	0.008025	0.008654	0.009152	0.009557	0.009891	0.010173	0.010413	0.010621	0.010802	0.010961	0.011102
1	0.008174	0.008789	0.009277	0.009672	0.01	0.010275	0.010510	0.010713	0.010890	0.011046	0.011184

Figure 40: 2016-17 CFB heatmap (Sonny Moore Top-25, Quartiles On)

The heatmap of Sonny Moore's top-25 ranking for the 2016-17 NCAA College Football (CFB) Division I dataset is shown in Figure 40. Similar to the other college football heatmaps from this season, this ranking also performs best with small Alpha and Beta values, with the lowest total backedge weight occurring at Alpha/Beta of (0,0). In contrast to the other rankings, this ranking appears to increase linearly in backedge weight regardless of whether Alpha or Beta are increased, where large values of Alpha and Beta result in the highest total backedge weights. We expect that this is because the ordering of teams in the ranking results in roughly equivalent backedge weights from older games and from close games, as increases in value to either results in similar increases towards the total backedge weight.

	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
0	**<u>0.228729</u>**	0.237419	0.242752	0.246358	0.248959	0.250924	0.25246	0.253695	0.254708	0.255556	0.256274
0.1	0.232984	0.241241	0.246285	0.249686	0.252135	0.253982	0.255425	0.256583	0.257534	0.258328	0.259001
0.2	0.236592	0.244464	0.249256	0.25248	0.254797	0.256542	0.257904	0.258997	0.259894	0.260642	0.261276
0.3	0.239689	0.24722	0.251789	0.254857	0.257059	0.258716	0.260009	0.261045	0.261894	0.262603	0.263203
0.4	0.242377	0.249603	0.253975	0.256905	0.259006	0.260585	0.261817	0.262803	0.263611	0.264285	0.264856
0.5	0.244732	0.251684	0.25588	0.258688	0.260699	0.26221	0.263387	0.264329	0.265101	0.265745	0.26629
0.6	0.246812	0.253517	0.257555	0.260254	0.262185	0.263635	0.264763	0.265667	0.266407	0.267024	0.267546
0.7	0.248663	0.255143	0.259039	0.26164	0.263499	0.264894	0.26598	0.266849	0.26756	0.268153	0.268655
0.8	0.25032	0.256597	0.260364	0.262876	0.26467	0.266016	0.267063	0.267901	0.268586	0.269157	0.269641
0.9	0.251813	0.257903	0.261553	0.263984	0.26572	0.267021	0.268033	0.268843	0.269505	0.270056	0.270523
1	0.253165	0.259084	0.262626	0.264984	0.266667	0.267928	0.268908	0.269691	0.270332	0.270866	0.271318

Figure 41: 2017-18 NFL heatmap (NFL.com, Quartiles Off)

	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
0	**0.222222**	0.228702	0.232678	0.235367	0.237306	0.238771	0.239917	0.240837	0.241593	0.242225	0.242761
0.1	0.231285	0.236792	0.240157	0.242426	0.244059	0.245291	0.246254	0.247027	0.24766	0.24819	0.248639
0.2	0.238968	0.243617	0.246446	0.24835	0.249718	0.250749	0.251553	0.252198	0.252728	0.253169	0.253544
0.3	0.245564	0.249451	0.251809	0.253392	0.254528	0.255383	0.25605	0.256585	0.257023	0.257389	0.257699
0.4	0.251289	0.254495	0.256435	0.257735	0.258667	0.259368	0.259914	0.260352	0.260711	0.26101	0.261263
0.5	0.256304	0.2589	0.260467	0.261516	0.262267	0.262831	0.263271	0.263623	0.263911	0.264151	0.264355
0.6	0.260734	0.26278	0.264013	0.264836	0.265426	0.265868	0.266213	0.266489	0.266714	0.266903	0.267062
0.7	0.264675	0.266224	0.267154	0.267776	0.26822	0.268554	0.268813	0.269021	0.269191	0.269332	0.269452
0.8	0.268205	0.2693	0.269958	0.270397	0.27071	0.270945	0.271128	0.271274	0.271394	0.271493	0.271578
0.9	0.271383	0.272066	0.272475	0.272748	0.272942	0.273088	0.273201	0.273292	0.273366	0.273428	0.273481
1	0.274262	0.274566	0.274747	0.274869	0.274955	0.27502	0.27507	0.27511	0.275143	0.275171	0.275194

Figure 42: 2017-18 NFL heatmap (USA Today, Quartiles Off)

	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
0	**0.089660**	0.10661	0.118152	0.126518	0.132861	0.137835	0.14184	0.145135	0.147892	0.150234	0.152247
0.1	0.090446	0.107594	0.11926	0.127711	0.134115	0.139136	0.143177	0.1465	0.149282	0.151643	0.153673
0.2	0.091106	0.108419	0.120189	0.12871	0.135165	0.140224	0.144296	0.147643	0.150444	0.152822	0.154866
0.3	0.091667	0.109121	0.120978	0.129559	0.136057	0.141149	0.145245	0.148613	0.15143	0.153822	0.155878
0.4	0.092151	0.109725	0.121657	0.130289	0.136824	0.141943	0.146062	0.149446	0.152278	0.154681	0.156747
0.5	0.092572	0.11025	0.122248	0.130924	0.137491	0.142634	0.146771	0.150171	0.153014	0.155428	0.157502
0.6	0.092942	0.110711	0.122766	0.131481	0.138076	0.143239	0.147393	0.150806	0.15366	0.156082	0.158164
0.7	0.093269	0.111119	0.123224	0.131974	0.138593	0.143775	0.147942	0.151367	0.15423	0.156661	0.158749
0.8	0.093561	0.111483	0.123633	0.132412	0.139053	0.144252	0.148432	0.151866	0.154738	0.157175	0.159269
0.9	0.093823	0.111809	0.123999	0.132806	0.139466	0.144679	0.148871	0.152314	0.155193	0.157637	0.159736
1	0.094059	0.112103	0.124329	0.13316	0.139838	0.145064	0.149266	0.152718	0.155604	0.158052	0.160156

Figure 43: 2017-18 NFL heatmap (NFL.com, Quartiles On)

	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
0	**<u>0.066841</u>**	0.087941	0.10231	0.112725	0.120621	0.126814	0.1318	0.135901	0.139334	0.142249	0.144756
0.1	0.068019	0.089153	0.103533	0.113949	0.121842	0.12803	0.133011	0.137107	0.140535	0.143446	0.145948
0.2	0.069007	0.090169	0.104557	0.114973	0.122864	0.129047	0.134024	0.138116	0.141539	0.144446	0.146945
0.3	0.069848	0.091034	0.105427	0.115844	0.123731	0.129911	0.134884	0.138972	0.142392	0.145295	0.147791
0.4	0.070573	0.091778	0.106176	0.116593	0.124478	0.130654	0.135624	0.139708	0.143125	0.146025	0.148517
0.5	0.071204	0.092425	0.106828	0.117243	0.125126	0.1313	0.136266	0.140347	0.143761	0.146658	0.149148
0.6	0.071758	0.092993	0.107399	0.117814	0.125695	0.131866	0.13683	0.140908	0.144319	0.147214	0.149702
0.7	0.072248	0.093496	0.107905	0.118319	0.126198	0.132367	0.137328	0.141404	0.144812	0.147705	0.150191
0.8	0.072686	0.093944	0.108355	0.118769	0.126646	0.132812	0.137771	0.141845	0.145251	0.148142	0.150626
0.9	0.073078	0.094345	0.108759	0.119172	0.127048	0.133212	0.138168	0.14224	0.145645	0.148534	0.151016
1	0.073432	0.094708	0.109123	0.119536	0.12741	0.133572	0.138526	0.142596	0.145999	0.148886	0.151367

Figure 44: 2017-18 NFL heatmap (USA Today, Quartiles On)