Fall 2019

Predictive analysis of real-time strategy games using graph mining

Isam Abdulmunem Alobaidi

PREDICTIVE ANALYSIS OF REAL-TIME STRATEGY GAMES USING GRAPH

MINING

by

ISAM ABDULMUNEM ALOBAIDI

A DISSERTATION

Presented to the Faculty of the Graduate School of the

MISSOURI UNIVERSITY OF SCIENCE AND TECHNOLOGY

In Partial Fulfillment of the Requirements for the Degree

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

2019

Approved by:

Dr. Jennifer Leopold, Advisor
Dr. Ricardo Morales
Dr. Patrick Taylor
Dr. Peizhen Zhu
Dr. Robert Paige

PUBLICATION DISSERTATION OPTION

This dissertation consists of the following four articles, formatted in the style used by the Missouri University of Science and Technology, which have been submitted for publication as follows:

Paper I: Pages 11-33 have been published in ICDM 2019, 19th Industrial Conference on Data Mining ICDM 2019.

Paper II: Pages 34-58 have been published in ICDM 2019, 19th Industrial Conference on Data Mining ICDM 2019.

Paper III: Pages 59-91 have been submitted to IJDMTA, International Journal of Data Mining Techniques and Applications.

Paper IV: Pages 92-112 have been submitted to IEEE ICDE 2020, IEEE International Conference on Data Engineering.

ABSTRACT

Machine learning and computational intelligence have facilitated the development of recommendation systems for a broad range of domains. Such recommendations are based on contextual information that is explicitly provided or pervasively collected. Recommendation systems often improve decision-making or increase the efficacy of a task. Real-Time Strategy (RTS) video games are not only a popular entertainment medium, they also are an abstraction of many real-world applications where the aim is to increase your resources and decrease those of your opponent. Using predictive analytics, which examines past examples of success and failure, we can learn how to predict positive outcomes for such scenarios. To do this, one way to represent this type of data in order to model relationships between entities is by using graphs. The vast amount of data has resulting in complex and large graphs that are difficult to process. Hence, researchers frequently employ parallelized or distributed processing. But first, the graph data must be partitioned and assigned to multiple processors in such a way that the workload will be balanced, and inter-processor communication will be minimized. The latter problem may be complicated by the existence of edges between vertices in a graph that have been assigned to different processors. One objective of this research is to develop an accurate predictive recommendation system for multiplayer strategic games to determine recommendations for moves that a player should, and should not, make which can provide a competitive advantage. Another objective is to determine how to partition a single undirected graph in order to optimize multiprocessor load balancing and reduce the number of edges between split subgraphs.

ACKNOWLEDGMENTS

There are no proper words to convey my deep gratitude and respect for my dissertation and research advisor, Dr. Jennifer Leopold. She has inspired me to become an independent researcher and helped me realize the power of critical reasoning. I have learned a lot from her. Her guidance and encouragement helped me in finishing my dissertation. Also, I would like to express my great thanks to Dr. Ricardo Morales, Dr. Peizhen Zhu, Dr. Patrick Taylor, and Dr. Robert Paige for serving on my committee. They generously gave their time.

My sincere expression of appreciation and thanks goes to my beloved wife Hiba, who was always my support in the moments when there was no one to answer my queries. Thank you with all my heart and soul. A big and great thanks from my deep heart goes to my adorable daughters Maryam and Reetal for their love and for filling my life with happiness. I would like to thank my Mom, Dad, and Aunt for all of the sacrifices that they have made on my behalf. Your prayer for me is what sustained me thus far. All of you have instilled admirable qualities in me and given me a good foundation on which to build my life. No acknowledgment would be complete without giving thanks to my mother-in-law and father-in-law for their prayers and support.

Finally, I would like to acknowledge my friend and colleagues in the research — a very special thanks to Ali Al-Lami, you are always helpful.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

NOMENCLATURE

Symbol	Description
α	Percentage of graphs that discriminative subgraph need not be present in C^+ when relaxing conditions
β	Percentage of graphs that discriminative subgraph need not be present in C^- when relaxing conditions
τ	User-specified Threshold

1. INTRODUCTION

Technology is continuously developing, creating a rapid "inflation" process that compounds over time. Although this development has solved many problems, it has also created new challenges. One of these challenges is the volume of data produced. Countless sources can produce this kind of massive data, for example, medical records, mobile phone applications, automated data creation, and customer databases.

Before looking for an appropriate way to analyze this massive amount of data, we need to determine what problem we are trying to solve. For instance, are we interested in predicting the next stage of a satisfactory condition? Do we want to develop specific recommendations with a view to taking proactive steps in military situations? Are we interested in predicting the type of problem that is expected to occur for a particular system? From this point of view, driving the analysis in a particular direction will depend primarily on the type of problem we wish to solve. One effective way to solve this type of problem is the graph. We can take advantage of the graph in the field of computer science for such kinds of problems in three different ways: increase performance, provide flexibility, and improve the speed of movement.

1.1. GRAPHS AND NETWORKS

A graph, mathematically, is a combination of arbitrary objects called nodes/vertices connected via paths/edges. The information, or data, can be visualized using graphs to uncover their relation and facilitate handling them. Either one big graph or transaction graphs can be used to represent a large volume of data. The choice of one of these forms

depends primarily on the nature of the data that will be dealt with. Basically, various operations can be performed on the vertices and edges within the graph, for instance, partition one large graph into many of those small ones based on some requirements, adding/deletion of vertices/edges into or from the graph's collection, or checking whether two vertices share an edge (are connected as frequent). Graphs are used all over the place. Networking makes heavy intensive use of them, and they are also utilized in artificial intelligence, data mining, game development, geoinformatics, bioinformatics, and many other disciplines. Effectively, anything that contains a set of connections can be represented in a graph form. Computer scientists have developed a great deal of theory about graphs and operations on them. This is partly because graphs can be used to represent many problems in computer science that are otherwise abstract.

Graphs can be categorized into various types based on the number of vertices/edges, interconnectivity, and their overall structure. The main feature in all types of graphs is whether they are directed or undirected. The difference is the same as between one directional and bidirectional streets. In a directed graph, the direction matters, and the edge can not be used in the other direction, while in an undirected graph, the direction does not matter, and can be simulated using a directed graph by using pairs of edges in both directions. Some graphs are extreme because they are made up of vertices only with no edges to connect them. For example, a Null graph consists of only a few vertices and a Trivial graph has only one vertex. These types of graph are not our focus.

The other type of graph is called connected graphs. To be a connected graph, it must fulfill the requirement of at least one existing edge for every vertex connected to some other vertex at the other side of the edge. If there is an edge from every single vertex to

every single other vertex, this will be a fully connected graph (or it can be called a complete graph). A graph will be called just connected if there is a path to get from each vertex to each of the other vertices, not necessarily in a direct path. Many categories fall under these two types of graph, whether it is full connect or not, such that Full-clique, Near-clique, Full-bipartite, Near-bipartite, Star, and Chain. Figures 1.1, 1.2, 1.3, and 1.4 show examples of these types of graph.

Full-clique

Near-clique

Figure 1.1: Clique Graph
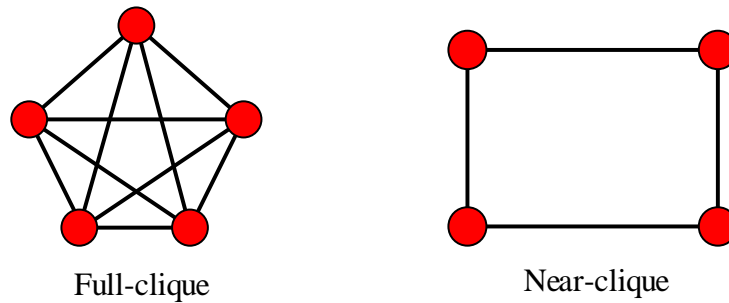
Full-bipartite
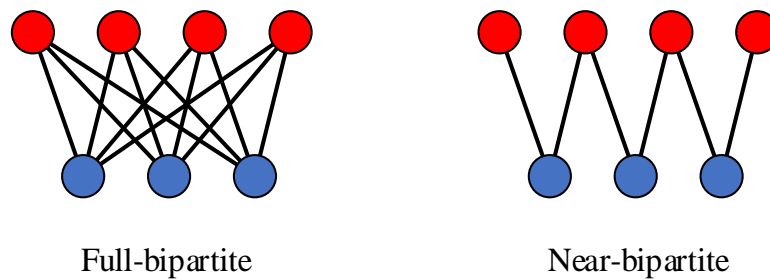
Near-bipartite

Figure 1.2: Bipartite Graph
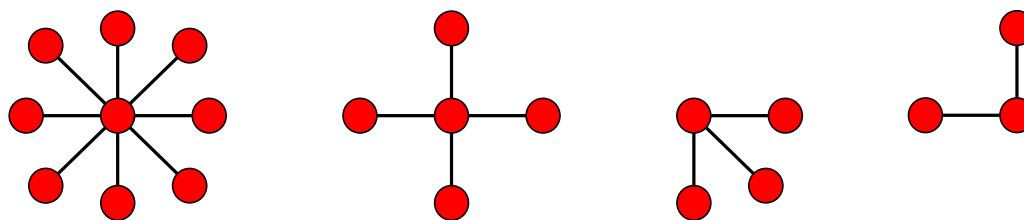
Figure 1.3: Star Graph
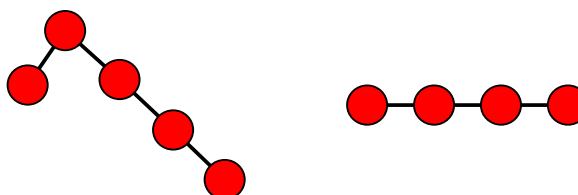
Figure 1.4: Chain Graph

1.2. FREQUENT SUBGRAPH MINING

The quintessence of graph mining is frequent subgraph mining (FSM), where the objective here is to extract all the frequent subgraphs in a given dataset whose existence counts are with/above a specified threshold. Many applications (e.g., chemoinformatics, bioinformatics, or machine learning) have utilized the FSM process in order to extract the critical knowledge [1, 2, 3, 4, 5, 6, 7] from data. The detection of beneficial hidden patterns in a very massive dataset represents its objective. This step will assist to reveal properties that identify real-world graphs from random graphs and uncover anomalies in a specific graph such as in 1) mining biochemical structures, 2) program control flow analysis, 3) mining XML structures or Web communities, 4) building blocks for graph classification, clustering, compression, comparison, and correlation analysis, or 5) fraud detection in

telecommunications networks, auction networks, social networks, or cyber-attacks and intrusion.
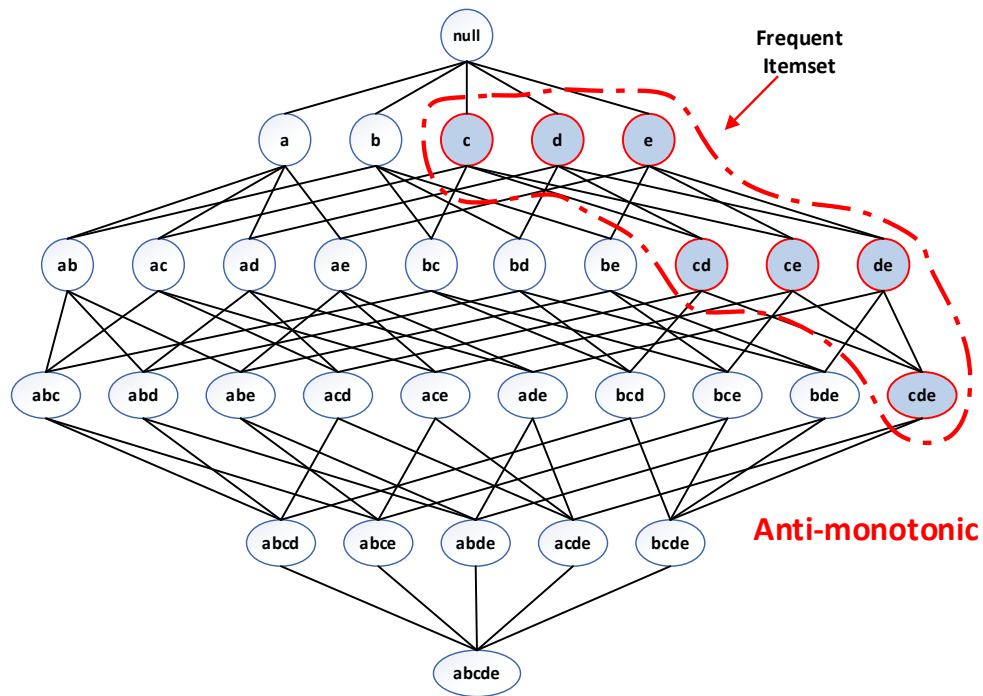
Figure 1.5: An illustration of the Anti-monotonicity property. If {c, d, e} is frequent, then all subsets of this itemset are frequent

Anti-monotonicity means that a $size - n$ subgraph is frequent only if all of its subgraphs are frequent [9]. As an illustration of the above property, consider the structure shown in Figure 1.5. If a subgraph such as {c, d, e} is found to be frequent, then the anti-monotonic property suggests that all of its $size - n$ subgraphs (i.e., the shaded itemsets in this figure) must also be frequent. The intuition behind this property is as follows: any subgraph that contains {c, d, e} must contain {c, d}, {c, e}, {d, e}, {c}, {d}, and {e}, (i.e., subgraphs of the 3-itemset). Therefore, if the support for {c, d, e} is greater than the support threshold, so are its subgraphs.

Conversely, if a subgraph such as {a, b} is infrequent, then all of its subgraphs must be infrequent too. As illustrated in Figure 1.6, the entire subgraph containing subgraphs of {a, b} can be pruned immediately once {a, b} is found to be infrequent. This strategy of trimming the exponential search space based on the support measure is known as support-based pruning. Such a pruning strategy is made possible by a key property of the support measure, namely, that the subgraph support never exceeds its original source support. Any measure that possesses an anti-monotonic property can be incorporated directly into the mining algorithm to effectively prune the candidate search space.
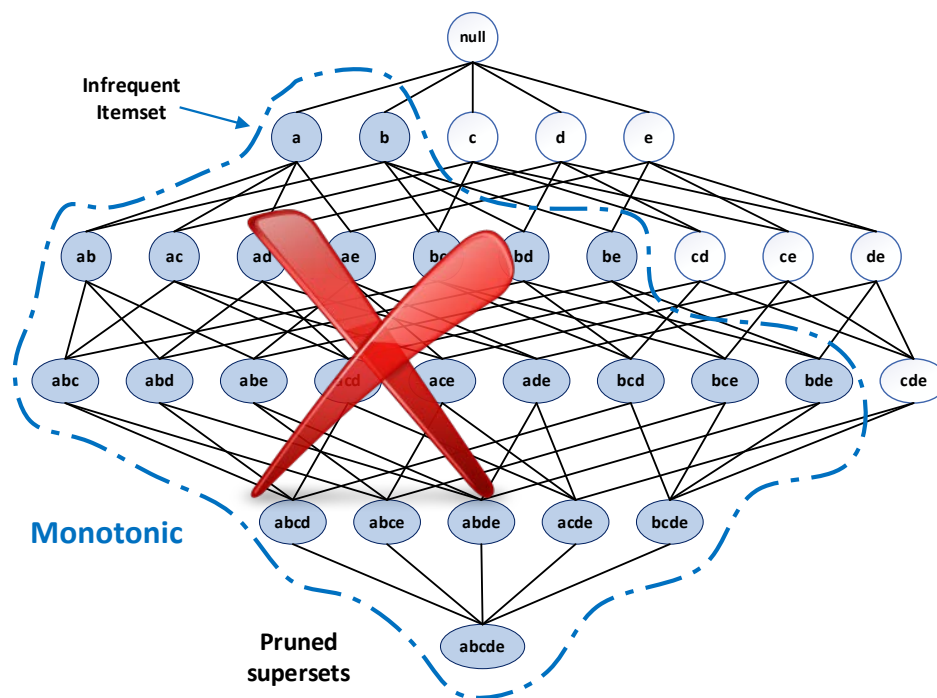
Figure 1.6: An illustration of support-based pruning. If {a, b} is infrequent, then all supersets of {a, b} are eliminated.

1.3. FREQUENT SUBGRAPH MINING COMPUTATIONAL CHALLENGES

1.3.1. Key Computational Challenges in Frequent Subgraph Mining. The main challenge in subgraph mining is efficiency [4, 6, 8], where

- The cardinality of the graph collection to be mined may be very large in both transaction and single graphs; for example, a biological network may consist of 7496 vertices ($|V| = 7496$) with 515 distinct labels and 25408 edges ($|E| = 25408$) [9].

- The number of graphs generated by the subgraph extension process is completely abundant. This abundance has a significant impact on increasing the cost of support evaluation, which is the core of the frequent subgraph mining process. The pattern-growth algorithm extends a frequent graph directly by adding a new edge in every possible position. It does not perform expensive join operations. A potential problem with the edge extension is that the same graph can be discovered multiple times. Also, a new edge to be extended could be frequent or not. The support evaluation is more complicated than subgraph extension and will cost more time during the mining process. A major challenge in mining frequent subgraphs is that the mining process often generates a huge number of patterns. This is because if a subgraph is frequent, all of its subgraphs are frequent as well. A frequent graph pattern with n edges can potentially have 2^n frequent subgraphs, which is an exponential number.

- Testing for graph isomorphisms is computationally intensive. The isomorphism challenge is that the vertices and edges in a given pair of graphs may be mapped in

a variety of ways. The number of possible mappings may be exponential in terms of the number of the vertices.

- Communication cost in distributed processing (i.e., MapReduce) is also an important concern, as large amounts of intermediate data may be generated and transferred among workers. Excessive network transmission increases the overall execution time of graph mining and may also lead to bottlenecks and failures.

1.3.2. Variation of Sequential, Distributed, and Parallel Processing in Frequent Subgraph Mining. Both parallel and distributed processing share the same challenge when they deal with either single or transaction graphs. The partitioning of a single graph or distributing of transaction graphs must satisfy the quality graph partitioning (guarantee of not losing any data), multilevel paradigm, and load balancing. One straightforward partition scheme for transaction graphs is to distribute the graphs [3], so that each partition contains the same number of graphs from the set of graphs $\boldsymbol{G} = (\boldsymbol{G_1}, \boldsymbol{G_2}, \dots, \boldsymbol{G_n})$. This works well for most of the datasets. However, for datasets where the size (edge count) of the graphs varies substantially, another splitting option occurs where the total number of edges aggregated over the graphs in a partition are close to each other. In this way, the load balancing factor of distributed processing will be improved. From another side, the number of partitions is also an important tuning parameter. The partitioning of the single graph considered can be a big challenge. The choice of the proper partitioning method will save a lot of edges from not being lost and maintain the frequency rules.

Frequent subgraph patterns from a single large graph in the distributed platform rely on computing the support of a pattern [1]. If the input graphs are partitioned over

various worker nodes, the local support of a subgraph in the respective partition at a worker node is not very useful for deciding whether the given subgraph is frequent or not. Support measures that simply count the occurrences of a pattern may violate the anti-monotonic property since occurrences of the pattern may overlap with each other. In a single graph, the challenge in mining a partitioned graph is that there can be false negative patterns (i.e., a pattern p that is globally frequent can be missed because certain edges involved in subgraph isomorphisms for p span different partitions). Communication cost (in distributed processing like MapReduce) is also an important concern as large amounts of intermediate data may be generated and transferred among workers. Excessive network transmission increases the overall execution time of graph mining, where the support computation cannot be delayed arbitrarily and may lead to bottlenecks and failures.

Both scenarios, single and transaction graph settings, share the scalability problems in the mining sequential patterns process, including 1) maximum time required for scanning the database, 2) size of the mining dataset, where large data input may exceed memory resources of a single machine, and 3) vast amounts of CPU time required to compute frequent patterns.

A huge number of possible sequential patterns may be hidden in databases. A mining algorithm should find the complete set of patterns when possible to satisfy the minimum support (frequency) threshold. This should be done in a manner that is highly efficient and scalable, involving only a small number of database scans and incorporating various kinds of user-specific constraints. Also, sequential pattern mining requires, besides the discovery of frequent itemsets, the arrangement of these itemsets in sequences and the

discovery of which of these are frequent. Moving towards a parallel or distributed environment is very important to solve the challenges in sequential pattern mining.

1.4. SUMMARY

Graphs are everywhere. The capability of a graph database to solve multiple domain problems, such as in biological networks, chemical patterns, social networks, or computer network and web data patterns, really are endless. The objective of this dissertation is to mine the historical data, whether structured or unstructured (e.g., real-time strategy (RTS) games or medical data) to discover and analyze the value hidden in their connection.

PAPER

I. THE USE OF FREQUENT SUBGRAPH MINING TO DEVELOP A RECOMMENDER SYSTEM FOR PLAYING REAL-TIME STRATEGY GAMES

Isam A. Alobaidi[1], Jennifer L. Leopold[1], and Ali A. Allami[2]

[1]Department of Computer Science, Missouri University of Science and Technology, Rolla, MO 65409

[2]Electrical Engineering & Computer Science Department, University of Missouri, Columbia, MO 65211

ABSTRACT

Machine learning and computational intelligence have facilitated the development of recommendation systems for a broad range of domains. Such recommendations are based on contextual information that is explicitly provided or pervasively collected. Recommendation systems often improve decision-making or increase the efficacy of a task. Real-time strategy (RTS) games are one domain where computationally determined recommendations for moves that a player should, and should not, make can provide a competitive advantage. The goal of our research is to develop an accurate predictive recommendation system for multiplayer strategic games that is based on frequent subgraph mining. Herein we present that approach and validate it using the historical data of one RTS game.

1. INTRODUCTION

The ever-increasing expansion of information and communications technology has initiated a new era for the development of recommendation systems for a wide variety of application domains (e.g., entertainment, E-commerce, E-health, etc.); see Figure 1. Recommendations could be for products or services that a customer might consider purchasing, treatments that a doctor might consider prescribing for a patient, or a sequence of actions that a robot should perform in a certain situation. Typically, the recommendations are based on an analysis of historical data, often characterized as positive and negative examples for the recommendation scenario. In order to be of value, recommendation systems must have high predictive accuracy.
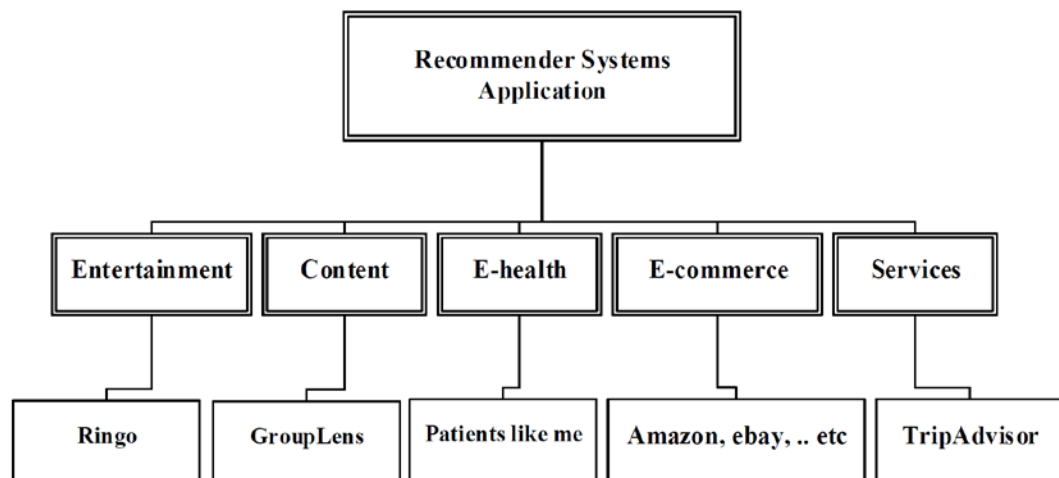
Figure 1: Recommender System Classification

Another venue where recommendation systems can be valuable is strategic games. Players have long been interested in studying previously played games to try to discern

which moves are advantageous to make and which moves should be avoided. With the current widespread interest in online, real-time strategy (RTS) games, which can involve a diverse and complex set of entities and functionality, determining which moves to make (and which not to make) can be extremely challenging. Fortunately, there are several databases of played games that can be analyzed to glean some insight.

In this study, we develop a predictive recommendation system for strategic multiplayer games that is based on graph mining. Using a database of played games, we model each of those games as a directed graph, and use frequent subgraph mining to look for patterns of moves that occurred frequently in winning games; these form the basis of our recommendations for moves that a player should make. Similarly, we look for patterns of moves that occurred frequently in losing games; those become the basis of our recommendations for moves that a player should not make. We test the accuracy of our method by repeatedly partitioning our database of played games into training and test datasets, and testing for the occurrence of true positives, true negatives, false positives, and false negatives. We also compare our method to an alternative approach, frequent sequence mining.

The organization of this paper is as follows. Section 2 provides a brief discussion of the main topics in this paper: game data mining, frequent subgraph mining, and frequent sequence mining. The particular algorithm that we used for frequent subgraph mining is explained in more depth in Section 3. A description of the RTS game data that we used for testing our method is provided in Section 4. Our experimental method and results are discussed in Section 5. A summary of this research and consideration of future work is discussed in Section 6.

2. BACKGROUND

In this section we briefly discuss some of the related work that has been done in the fields of game data mining, frequent subgraph mining, and frequent sequence mining.

2.1. GAME DATA MINING

One objective of game data mining is to analyze a collection of played games and find patterns of moves that were made in winning (and possibly losing) games. Game data mining was the main focus of research in [1, 2, 3]. In [2] a method, Playtracer, for game analysis and improvement was proposed. A multidimensional scaling strategy was applied to cluster players and game states, and a detailed visual representation of the paths taken by players during the game was provided. Specifically, Classical Multidimensional Scaling (CMDS) [4] was used in order to visualize the paths. The Playtracer method showed mutual ways that players succeeded and failed, and enabled tracking a specific player's progress across multiple levels.

Two widely used data mining techniques, Classification and Regression Trees (CART) and artificial neural networks, were utilized in [3] to analyze a collection of game data (i.e., STEAM) for predictive purposes. CART is a decision tree algorithm that aims to build a predictive model based on the values of several inputs. Artificial neural networks also attempt to discover new patterns from inputs by subjecting them to a repetitive learning process. The aim of this study was to predict what should be followed as accurately as possible. Their method relied on the analysis of the online reviews (e.g., number of screenshots, number of reviews of a specific action) to achieve their objectives.

2.2. FREQUENT SUBGRAPH MINING

Given a single (directed or undirected) graph, it can be useful to know which subgraphs occur at least n times where n is a user-specified threshold for frequency. Similarly, given a collection of graphs and a frequency threshold n, it may be important to know which subgraphs occur in at least n of those graphs. The process of answering this question is called frequent subgraph mining.

Several methods for frequent subgraph mining were presented in [5, 6, 7, 8]. An algorithm that finds only maximal frequent subgraphs from a collection of graphs was given in [5]. This method consists of two basic steps: (1) from a collection of graphs, all frequent trees (i.e., undirected graphs in which any two vertices are connected by exactly one path) are first found; (2) from the mined trees, maximal subgraphs then are constructed. This strategy can significantly reduce the size of the result set.

Another method was proposed in [6] to only find closed frequent graph patterns instead of mining all subgraphs. The main idea behind this method was to consider the graph g closed when it is not possible to find a proper supergraph of g with the same support (i.e., frequency) as g.

An algorithm named Fast Frequent Subgraph Mining (FFSM) was developed in [7]. The strategy in that work was to reduce the number of redundant candidate subgraphs that are examined by utilizing specialized operations (called FFSMJoin and FFSM-Extension) to generate the candidate subgraphs.

A technique for finding frequent subgraphs in a large sparse graph was proposed in [8]. In that work, two approaches for exploring the search space of subgraphs were examined. A breadth-first approach was employed in their first algorithm, HSI-GRAM,

examining the search space for frequent subgraphs in a horizontal way. A depth-first approach was employed in their second algorithm, VSI-GRAM, to explore the search space in a vertical fashion when looking for frequent subgraphs.

Amongst many of the frequent subgraph mining algorithms that have been developed, computationally expensive extension/joining operations (to create larger candidate subgraphs from smaller frequent subgraphs) and false positive pruning (to reduce the search space) have been the biggest challenges that researchers have tried to address. Unfortunately, most methods have been limited to only working on a single graph or a collection of graphs, but not being applicable to both settings.

Frequent subgraph mining is a reasonable approach to consider for game mining. Each played game can be represented as a directed graph, wherein a vertex represents a move made by a player in that game and an edge represents two consecutive moves. It then could prove useful to identify subgraphs (i.e., sequences of moves) that frequently occur in the collection of graphs (i.e., played games).

2.3. FREQUENT SEQUENCE MINING

Frequent sequence mining is used to find a set of patterns amongst a collection of instances that specify a sequence (e.g., a list) of items. This methodology can be used for diverse types of data; in [9] it was used to look for patterns in sequences of speech and bio-signals based on methods proposed in [10].

In [11], researchers proposed an algorithm called Sequential Pattern Discovery using Equivalence classes (SPADE). It starts by computing the frequencies of single-item sequences. In the next step, it counts the frequency of two-item sequences using a bi-

dimensional matrix to count the number of sequences for each pair of items. Subsequent n-item sequences are processed by joining $n-1$-item sequences using lists of ids representing other objects. The size of those ids lists is the number of sequences in which an item appears.

A disadvantage to frequent sequence mining algorithms is that the results (i.e., the most frequently occurring sequences) do not list the items in the same order that they may have appeared in an instance's sequence in the dataset; the method does not care about the order in which an item appeared in an instance's sequence, it simply cares about whether or not the item occurred in the instance's sequence. Nonetheless this method can potentially provide some predictive recommendations from a strategic game dataset where each game can be viewed as sequences of moves by a winner and a loser.

3. METHODOLOGY: FREQUENT SUBGRAPH MINING

The primary data mining technique that we used to develop a predictive recommendation system for strategic games was frequent subgraph mining. As mentioned in the previous section, we modeled each played game as a graph where a vertex represented a move in the game and an edge represented two consecutive moves. A game graph was not a strictly linear sequence of edges because some moves in turn generated multiple moves (e.g., a move could create a monster that would in turn propagate additional monsters, each of which would result in a new vertex and edge). We then analyzed the collection of graphs (a dataset of played games) to find frequent subgraphs: sequences of

moves that were common to several winners' games and sequences of moves that were common to several losers' games.

In this section we start by briefly providing some basic graph terminology that will facilitate discussion of the particular frequent subgraph algorithm that we utilized for our study.

3.1. PRELIMINARIES

Let $\mathcal{G} = \{G_1, G_2, \dots, G_n\}$ be a set of linear directed graphs which represents the historical data in our case. Each G_i represents a single game's moves, such that $G_i = (V_i, E_i)$ where V_i represents a node labeled as an action code of a player's move, while an edge in E_i represents two consecutive moves. A graph $T = (V_T, E_T)$ is a subgraph of $G_i = (V_i, E_i)$ iff $V_T \subseteq V_{G_i}, E_T \subseteq E_{G_i}$.

Definition 1. Let $T = (V_T, E_T)$ be a subgraph of a graph $G_i = (V_i, E_i)$. A subgraph isomorphism of T to G_i is an injective function $f : V_T \longrightarrow V_{G_i}$ satisfying $(f(u), f(v)) \in E_{G_i}$ for all edges $(u, v) \in E_T$. Intuitively, a subgraph isomorphism is a mapping from V_T to V_{G_i} such that each edge in E_{G_i} is mapped to a single edge in E_T and vice versa.

Problem 1. Given a set of graphs $\mathcal{G}$, the frequent subgraph isomorphism mining problem is defined as finding all subgraphs T in G such that $t_G(T) \geq \tau$, where $t_G(T)$ is the number of graphs in G that contain T and τ is the user-specified threshold.

Problem 2. Given a set of graphs $\mathcal{G}$ such that each G_i is divided into three phases G_{i1}, G_{i1}, G_{i3} and a frequent subgraph T, the frequent phase mining problem is defined as finding all subgraphs T in G_{ij} such that $t_{G_{ij}}(T) \geq \tau$, where τ is the user-specified threshold.

In our case, problem (2) counts the actual frequency (i.e., occurrences) of each subgraph provided that it is greater than or equal to τ. However, this may not be useful in various cases [8, 12], while others necessitate the exact number of occurrences (like graph indexing in [13].

3.2. GraMi ALGORITHM

For the purpose of generating candidate subgraphs, a variety of frequent subgraph mining and subgraph extension algorithms have been developed, as discussed in previous work [8, 14, 15]. In particular, GraMi [15] is one of the most efficient methods and is the foundation for the work presented in this paper. The key ideas behind GraMi are briefly outlined here.

Algorithm 1 is used to find a set of all frequent edges *fEdges* in the collection of graphs = $\{G_{i=1,...,n}\}$. All of these frequent edges have support greater than or equal to the assigned threshold τ. Because of the anti-monotone property, only frequent edges will be considered when finding the frequent subgraphs. Algorithm 2 is given each frequent edge to extend it to a new frequent subgraph. This is done by incorporating that edge with another subgraph. All extensions created in previous iterations are excluded by utilizing the *DFScode* canonical form that was introduced for gSpan [14]. The set *Candidate* in Algorithm 2 will include all the new subgraph extensions that had not been considered in prior iterations. In subsequent steps, any new subgraph extension within the set *Candidate* that does not meet the support threshold τ requirement will be discarded. If any of those subgraphs had been extended, they would produce a new non-frequent subgraph according to the anti-monotonic property.

Algorithm 1 Frequent Subgraph Mining - FSM

Input $\mathcal{G} = \{G_{i=1,...,n}\}$ and frequency threshold τ

Output All $fSubgraphs$ $S(G_i)$ with the support $\geq \tau$

  1:   $fSubgraphs \leftarrow \phi$

  2:   $Count = 0$

  3:   **for each** edge e_{G_i} **do**

  4:      **if** $e_{G_i} = e_{G_{i+1}}$ **then**

  5:        $Count ++$

  6:      **end-if**

  7:      **if** $Count \geq \tau$ **then**

  8:        $fEdges \leftarrow fEdges \cup e_{G_i}$

  9:      **end-if**

10:     **end-for**

11:     **for each** $e \in fEdges$ **do**

12:       $fSubgraphs \leftarrow fSubgraphs \cup SubE(e, \mathcal{G}, \tau, fEdges)$

13:       Remove e from $\mathcal{G}$ and $, fEdges$

14:     **end-for**

15:   **return** $fSubgraphs$

Algorithm 2 Subgraph Extension - $SubE$

Input $fSubgraphs$ $S, fEdges$ and frequency threshold τ

Output All Sub_{new} with the support $\geq \tau$

  1:   $Sub_{new} \leftarrow \phi$

  2:   $Candidate \leftarrow \phi$

  3:   **for each** $e \in fEdges$ and $n \in fSubgraphs$ **do**

  4:      **if** e fit to extend n **then**

  5:        Generate a new subgraph $ExtS$

  6:        **if** $ExtS$ exist in $\mathcal{G}$ and not generated before **then**

  7:          $Candidate \leftarrow Candidate \cup ExtS$

  8:        **else**

  9:          remove $ExtS$

10:        **end-if**

11:       **end-if**

12:     **end-for**

13:    **for each** $ExtS \in Candidate$ **do**

14:      **if** $ExtS$ count in $\mathcal{G} \geq \tau$ **then**

15:        $Sub_{new} \leftarrow Sub_{new} \cup SubE(ExtS, \mathcal{G}, \tau, fEdges)$

16:      **end-if**

17:    **end**

18: **return** Sub_{new}

3.3. USING FREQUENT SUBGRAPHS TO MAKE RECOMMENDATIONS

In this section we discuss the algorithms that we utilized in order to mine the game dataset for frequent subgraphs and build a recommendation system. The task of finding the number of occurrences for each subgraph was carried out using Algorithm 3. The mechanism for node-finding was used for matching the first node of a candidate subgraph with its occurrence in the original dataset. The objective of this process was to determine the starting point for conducting a depth-first search (*DFSearch*) to find all similar subgraphs in the winner (or loser) graph collection. These results were stored temporarily in a *temp* set to compute their replication in the subsequent steps, and then the final result was placed within *ExactFSG* set.

It was decided that the recommendation system might be more useful if the moves were analyzed for three phases of the game: the beginning of the game, the middle of the game, and the end of the game. This is traditionally being done for strategic games (i.g. chess) with the aim of analysis. Hence each game was divided into the first third number of moves, the second third number of moves, and the last third number of moves. Our work is not fixed to three phases; the number of phases can be easily modified by making a small change in Algorithm 4 to handle k phases. The objective of Algorithm 4 was to determine the number of occurrences of each individual subgraph considering in which phase of the

game the sequence of moves was made. Algorithm 4 takes the *ExactFSG* set that was

introduced by Algorithm 3 and facilitates the node-finding and *DFSearch* process to

determine the phase of each individual frequent subgraph in this set. The node-finding

mechanism was used a second time in subsequent steps, but only to identify the first node

identity, $node_{ID}$, of the candidate subgraph assigned to it from the original dataset this

time. It is worth mentioning that we consider the majority of appearances to decide the

phase of the frequent subgraph. It should be noted that the subgraph nodes may straddle

two consecutive phases. If so, we report that subgraph as it appeared in two phases.

Algorithm 3 Exact Subgraph Frequency

Input $\mathcal{G} = \{G_{i=1,\dots,n}\}$, $fSubgraphs\ S$ and frequency threshold τ
Output All the Exact Frequent Subgraph with their frequency

1: $Count = 0$
2: **for** $i = 1 \longrightarrow$ all graphs in $(fSubgraphs)$ **do**
3:    $frq = 0$
4:    **for** $j = 1 \longrightarrow$ all graphs in $(\mathcal{G})$ **do**
5:      **if** findnode $(G_j,\ fSubgraphs_i) \neq 0$ **do**
6:        $temp \longleftarrow$ dfsearch $(G_j,\ fSubgraphs_i)$
7:        **if** $temp \geq size(\ fSubgraphs_i)$ & $isisomorphic(\ fSubgraphs_i, G_j)$ **do**
8:          $frq + +$
9:        **end-if**
10:      **end-if**
11:    **end-for**
12:    **if** $frq \geq \tau$ **do**
13:      $count + +$
14:      $ExactFSG(count) \longleftarrow fSubgraphs_i$
15:    **end-if**
16: **end-for**
17: **return** $ExactFSG$

Algorithm 4 Majority of Subgraph Appearance

Input $\mathcal{G} = \{G_{i=1,\ldots,n}\}$ and $ExactFSG$

Output Display each $ExactFSG$ and the locate $phase$

1:   $count_1 = 0, \ count_2 = 0, \ count_3 = 0$

2:   $phase_1 = 0, \ phase_2 = 0, \ phase_3 = 0$

3:    **for** $i = 1 \longrightarrow$ all graphs in $(ExactFSG)$ **do**

4:      $frq = 0$

5:     **for** $j = 1 \longrightarrow$ all graphs in $(\mathcal{G})$ **do**

6:       $phase = \lceil size(G_i)/3 \rceil$

7:      **if** findnode $(G_j, ExactFSG) \neq 0$ **do**

8:        $temp \longleftarrow$ dfsearch $(G_j, ExactFSG)$

9:       **if** $temp \geq size(ExactFSG) \ \& \ isisomorphic(fSubgraphs_i, G_j)$ **do**

10:         **for** $k = 1 \longrightarrow \ size(ExactFSG)$ **do**

11:          $node_{ID} =$ findnode $(G_j, ExactFSG_i)$

12:          **if** $node_{ID} \leq phase$ **do**

13:           $count_1 + +$

14:          **elseif** $node_{ID} > phase \ \& \ node_{ID} \leq phase * 2$ **do**

15:           $count_2 + +$

16:          **else**

17:           $count_3 + +$

18:          **end-if**

19:         **end-for**

20:        **end-if**

21:       **end-if**

22:      **if** $count_1 \neq 0$ **do**

23:       **if** $count_1 > count_2$ **do**

24:        $phase_1 + +$

25:       **elseif** $count_1 < count_2$ **do**

26:        $phase_2 + +$

27:       **else do**

28:        $phase_{1\&2} + +$

29:       **end-if**

30:      **elseif** $count_2 \neq 0$ **do**

31:       **if** $count_2 > count_3$ **do**

32:        $phase_2 + +$

```
33:        elseif $count_2 < count_3$ do
34:            $phase_3 + +$
35:        else do
36:            $phase_{2\&3} + +$
37:        end-if
38:      else
39:            $phase_3 + +$
40:        end-if
41:    end-if
42:  end-for
43: return $phase$ result
```

4. DATA DESCRIPTION

Interloper is an online multiplayer real-time strategy (RTS) game [16]. The game allows the creation and deployment of entities, and the destruction of an opponent's entities. A player wins the game when the other player's entities/assets have been destroyed or the other player cannot create any more assets. A dataset of 19 played games involving 2 players was obtained for this study. Each player's move in the dataset was encoded with 15, 7, or 6 digits. The first two digits in a code of length 15 or 7 represented the type of action (i.e., move); only the first digit was used in a code of length 6 to represent the type of action. The last four digits in all codes were used to represent a counter of each specific action. The purpose of the counter was to produce a unique data item for each move in the game. The middle eight digits in a code of length 15 was used to represent the source and destination location when moving an entity. The player *ID* was represented with the third digit in codes of lengths 15 and 7, and with the second digit in codes of length 6.

For this study the dataset was separated into the winner's moves and the loser's moves for each game. Because of the limited size of the dataset we obtained (i.e., 19 games), a program was written to increase the number of games to 90 and 120 by randomly duplicating games. Our method was tested on both the original dataset of size 19 and the larger datasets of sizes 90 and 120.

5. EXPERIMENTAL EVALUATION

In this section we discuss the criteria by which we evaluated the performance of our recommendation system. As noted above, we analyzed the game in terms of three phases (i.e., beginning game, middle game, and end game) by dividing each game into three equal parts; the total number of moves in a game (by both the winner and the loser) ranged from 183 to 5,338. For each of the 3 phases analyzed, 60% of the data were used for training and the remaining 40% were used for testing with k-fold cross-validation [17]. We measured precision and recall, which are viewed as metrics of exactness and completeness of testing, respectively. Equations 1 and 2 are the mathematical formulas for precision and recall, respectively.

$$Precision = \frac{True\ Positive}{True\ Positive + False\ Positive} \tag{1}$$

$$Recall = \frac{True\ Positive}{True\ Positive + False\ Negative} \tag{2}$$

The trade-off between precision and recall was measured by using for another metric named F-measure [17], which represents the harmonious mean between precision and recall. The accuracy scale was applied to measure the closeness of the measured value to the true value. Equations 3 and 4 are the mathematical formulas of F-measure and accuracy, respectively.

$$F - measure = \ 2 * \frac{Recall * Precision}{Recall + Precision} \tag{3}$$

$$Accuracy = \frac{True\ Positive + True\ Negative}{True\ Positive + False\ Positive + False\ Negative + True\ Negative} \tag{4}$$

5.1. EXPERIMENT AND RESULTS

In this section we present the results of analyzing the Interloper game dataset using both frequent subgraph mining and frequent sequence mining. The algorithms presented in Section 3 were (collectively) implemented in Matlab and Java. SPADE (discussed in Section 2.3) was implemented in R. Our experiments were executed on an Intel(R) Core (TM) i7-6700 CPU@3.40GHz computer with 32GB memory.

Tables 1 and 2 show some of the experimental results of frequent subgraph mining using a threshold of 2 for the winner and loser datasets consisting of 19 games. The first and second columns show the actions in the frequent subgraphs with their number of occurrences from the entire dataset, respectively. The tables also list the phase of the game in which each frequent subgraph most often was found. The fourth column in each table is

a classification of the majority of that subgraph's actions; we classified that game's actions as either offensive, defensive, or movement (of an entity in the game space).

Table 1: Winner Data

Winner Subgraph	Frequency	Majority of Appearance	Classification
2810040 2810041 2810042	3	Second phase	offensive
2810035 2810036 2810037	4	First phase	offensive
2300171 2300172 2300173 2300174	3	Second and Third phase	move
2300171 2300172 2300173	6	Second phase	move
2810084 2810085 2810086 2810087	3	First and Second phase	offensive
2810084 2810085 2810086	3	Third phase	offensive
2500010 2500011 2500012	2	Third phase	offensive

Table 2: Loser Data

Winner Subgraph	Frequency	Majority of Appearance	Classification
2710003 2810015 2810017	3	First phase	offensive
2810003 2810005 2710001	2	First phase	offensive
2810024 2810026 2810028	6	First phase	offensive
2810008 2810010 2810012	3	Third phase	offensive
2510000 2510001 2510002	3	Third phase	offensive
2310187 2310188 2310189	3	Second phase	move
2310419 2310420 2310421	4	Third phase	move

These results were obtained by performing 3-fold cross-validation, repeated five times. Each time, for the 19-game dataset, 12 games were selected randomly (without duplication) for training, and the remaining 7 games were used for testing. The size of the resulting frequent subgraphs ranged from two nodes with one edge to four nodes with three edges. All of the two-node subgraphs were ignored because of the limited information they

provide for the recommendation objective (i.e., only two moves) compared to larger subgraphs.

Frequent subgraphs that were found in the winner graphs indicate actions that are recommended for a player to do, whereas frequent subgraphs that were found in the loser graphs indicate actions that are recommended that a player should not do. The benefit of the counter attached to each action reflects the relative number of times the player had made that type of move in that game. Characterizing the actions, such as offensive or defensive, gives a general notion of the strategy the player is employing in that sequence and would facilitate mapping one game's actions to another's (e.g., mapping Interloper's offensive actions to StarCraft's offensive actions).

Figure 2: Comparison of Average Precision, Recall and F-measure for Different Number of Games

Figure 2 shows the precision, recall, and F-measure scores obtained for each phase of the game that was analyzed. In order to ensure the fineness of the results, three different sizes of datasets were tested: 19, 90, and 120 games. All of these tests were subject to the same conditions of the 3-fold cross-validation with five repetitions. Averages for these five repetitions were calculated to determine the final results of these metrics. Precision and recall can be affected when the size of the input dataset increases. Despite this, our system did not experience a significant difference in those results. Figure 3 shows the comparison of average accuracy for the different sized datasets.
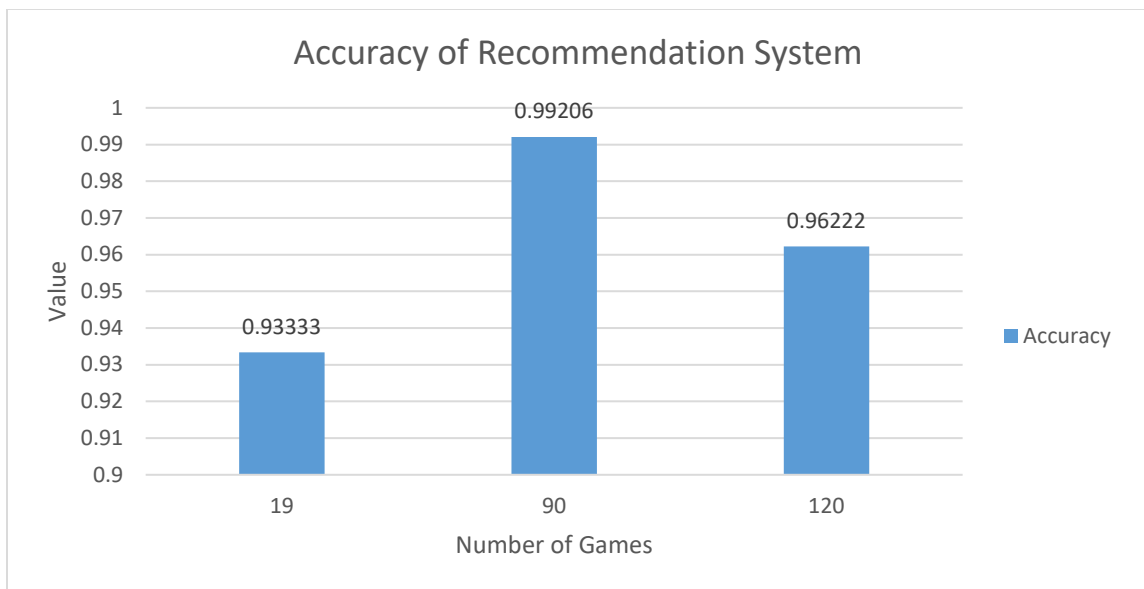
Figure 3: Comparison of Average Accuracy for Different Number of Games

We also utilized frequent sequence mining to analyze the Interloper game data; specifically, we used an implementation of the SPADE algorithm discussed in Section 2.3. The majority of the SPADE results (some of which are shown in Table 3 for the 19-game dataset) were not consecutive sequences of actions from games; they were simply lists of

individual actions that had occurred in some order in a majority of winners' or losers' games. While this was somewhat informative, it was equivalent to if we had limited our frequent subgraph mining to subgraphs of single vertices (no edges). Unfortunately, the highest support for the results returned by SPADE was 0.5, meaning that only 50% of the games in the tested dataset contained the reported list of actions. This was the case for not only the 19-game dataset, but also the larger 90- and 120-game datasets. Consequently, we did not feel that the predictive accuracy of the recommendations we could make from these results would be high, and did not pursue cross-validation testing.

Table 3: Portion of the SPADE Output for the 19-Games Dataset

Dataset	Phase	Support	Subgraph
Winner	1	0.5	2300005 2300006 2300000 2300005 2300006 2300000 2300004 2300005 2300006
Winner	2	0.3	2700018
Winner	3	0	No result
Loser	1	0.4	2810003 2710007 2810003
Loser	2	0.2	2710017 2810064 2810066 2710017 2810064 2810066
Loser	3	0.2	600011 500010 600011 500010 600009 600011

6. CONCLUSION AND FUTURE WORK

The use of recommendation systems has become widespread in our society. In general, they examine historical data and try to predict what should be done in the future. Herein we have applied a graph data mining technique, frequent subgraph mining, to a

strategy game dataset to develop a system that can provide recommendations about moves that a player should and should not make in order to improve his/her chances of winning the game. As proof of concept, we tested our system on a real-time strategy (RTS) game dataset, and achieved very accurate results when we tested our recommendations. We also attempted to apply another technique, frequent sequence mining, but did not find that it provided as useful or accurate recommendations.

In the future we plan on testing our approach on other RTS games such as StarCraft, and will try to develop a generalized mapping scheme for action types that will be applicable for the broader genre of RTS games. We then hope to apply this approach to other problem domains that can map their entities and actions to those of a strategic game in a broad semantic sense, where resources are effectively created and destroyed, and where it would be beneficial to have recommendations for optimal management of those resources.

REFERENCES

[1]   Drachen, C. Thurau, J. Togelius, G. N. Yannakakis, and C. Bauckhage, "Game Data Mining," in *Game Analytics: Maximizing the Value of Player Data,* (London, UK), pp. 205–253, Springer, 2013.

[2]   E. Andersen, Y.-E. Liu, E. Apter, F. Boucher-Genesse, and Z. Popović, "Gameplay Analysis Through State Projection," in *Proceedings of the 5$^{th.}$ International Conference on the Foundations of Digital Games*, (Monterey, CA, USA), pp. 1–8, ACM, 2010.

[3]   H.-N. Kang, H.-R. Yong, and H.-S. Hwang, "A Study of Analyzing on Online Game Reviews using a Data Mining Approach: STEAM Community Data," *International Journal of Innovation, Management and Technology*, vol. 8, no. 2, pp. 90, 2017.

[4]   M. A. A. Cox and T. F. Cox, "Multidimensional Scaling," *Handbook of Data Visualization*, pp. 315–347. Berlin, Heidelberg: Springer, Berlin Heidelberg, 2008.

[5]   J. Huan, W. Wang, J. Prins, and J. Yang, "SPIN: Mining Maximal Frequent Subgraphs from Graph Databases," in *Proceedings of the 10th. ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD `04, (Seattle, WA, USA), pp. 581–586, ACM, 2004.

[6]   X. Yan and J. Han, "CloseGraph: Mining Closed Frequent Graph Patterns," in *Proceedings of the 9th. ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '03, (Washington DC., USA), pp. 286–295, ACM, 2003.

[7]   J. Huan, W. Wang, and J. Prins, "Efficient Mining of Frequent Subgraphs in the Presence of Isomorphism," in *Proceedings of the 3rd. IEEE International Conference on Data Mining*, ICDM '03, pp. 549–552, IEEE, 2003.

[8]   M. Kuramochi and G. Karypis, "Finding Frequent Patterns in a Large Sparse Graph," in *Proceedings of the 2004 SIAM International Conference on Data Mining*, pp. 345–356, SIAM, 2004.

[9]   H. P. Martinez and G. N. Yannakakis, "Mining Multimodal Sequential Patterns: A Case Study on Affect Detection," in *Proceedings of the 13th. International Conference on Multimodal Interfaces*, ICMI '11, (Alicante, Spain), pp. 3–10, ACM, 2011.

[10]   R. Agrawal and R. Srikant, "Mining Sequential Patterns," in *Proceedings of the 11th. International Conference on Data Engineering*, (Taipei, Taiwan), pp. 3–14, IEEE, 1995.

[11]   M. J. Zaki, "SPADE: An Efficient Algorithm for Mining Frequent Sequences," *Machine Learning*, vol. 42, no. 1, pp. 31–60, 2001.

[12]   W.-T. Chu and M.-H. Tsai, "Visual Pattern Discovery for Architecture Image Classi_cation and Product Image Search," in *Proceedings of the 2nd. ACM International Conference on Multimedia Retrieval*, ICMR '12, (Hong Kong, China), pp. 1–27, ACM, 2012.

[13]   X. Yan, P. S. Yu, and J. Han, "Graph Indexing: A Frequent Structure-based Approach," in *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, (Paris, France), pp. 335–346, ACM, 2004.

[14]   X. Yan and J. Han, "gSpan: Graph-based Substructure Pattern Mining," in *Proceedings of the 2002 IEEE International Conference on Data Mining*, ICDM '02, (Maebashi City, Japan), pp. 721–724, IEEE, 2002.

[15] M. Elseidy, E. Abdelhamid, S. Skiadopoulos, and P. Kalnis, "GraMi: Frequent Subgraph and Ppattern Mining in a Single Large Graph," Proc. *VLDB Endowment*, vol. 7, no. 7, pp. 517–528, 2014.

[16] "Interloper Game Description." http://interlopergame.com/. Accessed: 2018-18-12.

[17] P.-N. Tan, M. Steinbach, and V. Kumar, *Introduction to Data Mining*. Pearson Education India, 2006.

II. PREDICTIVE ANALYSIS OF REAL-TIME STRATEGY GAMES USING DISCRIMINATIVE SUBGRAPH MINING

Jennifer L. Leopold[1], Isam A. Alobaidi[1], and Nathan W. Eloe[2]

[1]Department of Computer Science, Missouri University of Science and Technology, Rolla, MO 65409

[2]School of Computer Science and Information Systems, Northwest Missouri State University, Maryville, MO, USA

ABSTRACT

Real-Time Strategy (RTS) video games are not only a popular entertainment medium, they also are an abstraction of many real-world applications where the aim is to increase your resources and decrease those of your opponent. An obvious application is a military battle; yet another example is a person's physical health where it is advantageous to increase the number of healthy cells in the body and destroy cancerous cells (wherein cancer is your opponent). Using predictive analytics, which examines past examples of success and failure, we can learn how to predict positive outcomes for such scenarios. Herein we show how discriminative subgraph mining can be employed to analyze a collection of played RTS games, and make recommendations about sequences of actions that should, as well as should not, be made to increase the chances of winning future games. As proof of concept, we present the results of an experiment that utilizes our strategy for one particular RTS game.

1. INTRODUCTION

Real-Time Strategy (RTS) games are a subgenre of strategy video games wherein the participants position and maneuver units (e.g., troops, robots, and drones) and structures under their control to secure areas and destroy their opponent's assets. In some games, the created entities can in turn create and destroy other entities. Hence the focal points of such games are: resource generation and destruction, and indirect control of units and structures (via other units and structures). RTS games typically have a diverse set of resources which the player can deploy, basically offensive or defensive in nature, and a large variety of environments/storylines from which to select, often with a military science fiction theme; a popular and sophisticated example is StarCraft. The games are usually multi-player, with the winner determined by some criterion such as the player with the most assets at the end of a certain time period or by the last player remaining after all other players' assets have been depleted. Although the RTS game scenario is used for entertainment purposes, it can be abstracted as a model for real-world applications such as military battles, cyberinfrastructure networks that may need to be managed as they come under malicious attack, and even disease history/diagnosis systems which track a patient's symptoms, treatments, and disease progression over time.

Herein we test the hypothesis that predictive analytics can be employed to examine a collection of played games and make recommendations as to what a player should do and what a player should not do in order to increase the chances of winning the game the next time s/he plays. As proof of concept, this method will be tested for one particular RTS game; however, the method that we employ should be applicable to any multi-player RTS

game and possibly could be generalized to sequences of categorically offensive versus defensive moves for any RTS game. Specifically, we will model the moves of each played game as directed graphs for the winner's and loser's moves, respectively, and apply discriminative subgraph mining to identify our game strategy recommendations.

The organization of this paper is as follows. Section 2 provides a brief overview of game data mining, data mining techniques used in predictive analytics, and discriminative subgraph mining. Section 3 explains the discriminative subgraph mining algorithm that we utilized for our study. Section 4 outlines the experiment that we conducted to test our hypothesis and the results that we obtained. A summary and conclusions of our research are given in Section 5. Future work is discussed in Section 6.

2. RELATED WORK

2.1. GAME DATA MINING

For years there has been interest in analyzing games played by others in order to become a more competitive player. In its earliest form, people sought to identify the moves in the game that led to desirable, rather than undesirable, outcomes. For many games it is not only the quantity of assets, but particular features of the assets in the game that must be considered (e.g., an asset's functionality and location). For example, in the game of chess, given the choice, it is usually better to have one bishop than three pawns; position of a piece on the game board is also important as a bishop that is blocked by other pieces may not be able to attack. A number of studies have been conducted wherein a database of played games is analyzed to determine the winning percentage under various scenarios

such as games in which one player has two bishops and no knights and the other player has two knights and no bishops after some point in a chess game; see [1, 2] for examples of such studies. Contemporary genres of games, such as RTS video games, have a much more sophisticated collection of assets (e.g., game pieces) than traditional games such as chess and the characteristics of the assets can be much more diverse. Accordingly, analysis of desirable asset acquisition and deployment throughout a game has become more complex and computationally expensive.

Another branch of game data mining, also known as game telemetry, involves analysis of the people who play the game and/or the personas they may create. There are databases of this information for various online games and mining software to analyze data such as the players' skill level and time spent having played the game; see [3] for an example of such software. Some analyses may try to relate features from a player's profile to his/her winning percentage and odds of winning future games. This area of study is not the focus of the research pursued herein; we do not consider any data related to a player's profile.

As is discussed in [4], the intentions of game data mining should be made clear. *Description* describes patterns found in the game data; similarly, *characterization* is a summation of some general features associated with the data. These patterns could be independent of whether they occurred in the winners' games or the losers' games, or whether the patterns occurred in a majority or a minority of the games in the dataset. Description and characterization are the fundamental, general goals of most data mining efforts. *Classification* (and *clustering*) are used to compare and organize some features of the data into classes; with game data this usually isn't necessary since we are most

interested in classifications as winning and losing games, information which is already known. *Discrimination* seeks to identify the differences between groups of instances in the game data beyond just the classification of winning and losing. *Prediction* has the goal of providing a rule (or some form of guideline) that can be used as guidance for playing or forecasting the outcome of future games. The work presented in this study focuses on discrimination and prediction of game data.

2.2. DATA MINING TECHNIQUES USED IN PREDICTIVE ANALYTICS

Utilizing mathematical modeling, the field of predictive analytics examines past examples of success and failure to determine the variables that lead to successful outcomes and can be used to make predictions about future events. It has been used widely in the financial and insurance sectors. Here we briefly discuss some of the most common types of data mining methods used for predictive analytics.

Regression analysis: Linear regression is one type of regression analysis commonly used for predictive analytics. This method analyzes the relationship between a dependent variable and a set of independent variables. The relationship is expressed as an equation that predicts the value of the dependent variable as a linear function of the independent variables. For game data the dependent variable is typically the outcome of the game (i.e., win or lose) and the independent variables can be the various possible moves. Given the number of possible moves in an RTS game and the number of possible sequences of moves, this method could be computationally prohibitive.

Rule induction: Rule induction methods such as association rule mining seek to find relationships between variables in the dataset. For example, it could be determined that

when the player does actions A and B, s/he also does actions C and D. By applying association rule mining on only the winners' games, we could identify some actions that winning players did. Similarly, by mining the losers' games, we could find some actions common to losing players. However, we then would have to examine the differences between those rule sets to gain knowledge about what winners did that losers did not do, and vice-versa. It should be noted that rule mining typically generates a considerable number of rules because of its combinatorial approach; typically, only rules meeting a certain support threshold are retained.

Decision trees: Decision trees are most often used for classification and can be thought of as a graphical depiction of a rule; each branch of a decision tree can be thought of as a separate rule consisting of a conjunction of the attribute predicates of nodes along that branch. One approach would be to construct decision trees from the winning games and losing games, respectively. Resultingly, the issues previously mentioned for association rule mining of the RTS game data would apply for decision tree methods as well.

Clustering: Clustering is a way to categorize a collection of instances in order to look for patterns; groups are formed to maximize similarity between the instances within a group and to maximize dissimilarly between instances in different groups. Game data are already clustered into two groups: winners and losers. For the purpose of analyzing successful (and unsuccessful) actions, we would likely attempt to form clusters of action sequences. As with linear regression, given the number of possible moves in an RTS game and the number of possible sequences of moves, this method would be computationally prohibitive, and likely would result in an uninformative number of clusters, unless some

type of feature reduction mapping method was employed (i.e., mapping specific actions and their time of occurrence in the game to more generalized representations).

Neural networks: Neural networks are composed of a series of interconnected nodes that map a set of inputs into one or more outputs. The interconnections between inputs (which, for the game data, could be actions in the game) are determined based on an analysis of the played games. As with clustering, this method likely would be computationally prohibitive, and would probably not yield useful results, for the RTS game data unless we employed some type of data reduction mapping, which subsequently could result in loss of useful, specific information.

2.3. DISCRIMINATIVE SUBGRAPH MINING

Many problems can be modeled with graphs, wherein entities are represented as vertices and relationships between entities are represented as edges. When the relationship between two vertices has some semantic distinction of a predecessor and a successor, the edges are directed and hence the graph is considered directed. A played RTS game can be modeled as a directed graph where each action (e.g., move) is represented by a vertex and an edge represents two consecutive actions that were made in the game. By necessity, each vertex also must be identified by which player performed that action. The moves for one player do not form a strictly linear sequence because an action can generate multiple actions; for example, the player may create a drone which in turn simultaneously spawns 5 more drones, each of which becomes a new vertex, and 5 edges are created from the propagating drone vertex.

Finding interesting patterns in graphs (both directed and undirected) has been well-researched and is applicable to many problem domains in fields such as bioinformatics, cheminformatics, and communication networks. An 'interesting' pattern in a graph could be a subgraph that appears frequently over a collection of graphs or could be a subgraph that has a particular topography (e.g., a clique). Another type of interesting pattern is a discriminative subgraph.

Discriminative subgraph mining seeks to find a subgraph that appears in one collection of graphs, but does not appear in another collection of graphs. This approach has been used to study several problems including identifying chemical functional groups that trigger side-effects in drugs [5], classifying proteins by amino acid sequence [6], and identifying bugs in software [7, 8, 9]. Here we briefly discuss some of the strategies that have been employed for discriminative subgraph mining.

In [10] the authors define global-state networks, a collection of graphs that represent a series of snapshots taken over a period of time and model some event. Each snapshot graph has the same topology, but the nodes and/or edges in each graph may have different values. The authors' technique, MINDS, is specifically designed to find minimally discriminative subgraphs in large global-state networks. The network graph search space is organized as a set of decision trees to scrutinize the changes from one snapshot to the next in the collection. To reduce an exponential subgraph search space, they employ a Monte Carlo Markov sampling strategy. While the strategies employed in MINDS were found to work well for the global-state networks, they would not be appropriate for the RTS game dataset where each game, and hence each graph's topology, can differ significantly from other games. Additionally, as will be discussed in Section 3,

game data mining should not necessarily be limited to just finding minimally discriminative subgraphs.

Discriminative subgraph mining was used in [11] to find subgraphs that would cover as many positive examples and as few negative examples as possible. The test dataset contained protein structures possessing a specific function and proteins not having that function. Each graph contained approximately 1,000 edges and was very dense (i.e., in terms of the number of edges relative to the number of vertices in the graph). Two heuristics were employed to reduce the computational complexity of the mining process. Together these heuristics were used to assign a score to each candidate discriminative subgraph; the score considered the number of positive graphs minus the number of negative graphs in which the subgraph was found. Only the smallest such subgraphs with high scores were returned in the results; any (larger) subgraph that contained one of these (smaller) subgraphs was not further examined, thereby reducing the search space. This algorithm could have been adapted for the predictive game strategy study, but would have had to have been run for both the cases of the winning games being the positive examples and the losing games being the negative examples, and the winning games being the negative examples and the losing games being the positive examples in order to find recommendations for what should and should not be done to win the game.

Another strategy for dealing with the large search space normally incurred with discriminative subgraph mining was presented in [12]. As discussed above, a scoring scheme was used to evaluate the discrimination potential of candidate subgraphs. However, The Learning To Search (LTS) algorithm of [12] differed from the work of [11] by combining the scoring scheme with a sampling strategy to select candidate subgraphs.

Candidates deemed promising (in terms of their score) were added to a list and further extended with edges for additional consideration; non-promising candidates were discarded, thereby implementing a branch-and-bound search. This method was tested on protein datasets with good prediction accuracy and a faster runtime than some other discriminative mining methods. As with the algorithm in [12], this approach possibly could be used to analyze a strategy game dataset.

Discriminative subgraph mining also has been used to find bugs in software in [7, 8, 9]. For this application, a program is modeled as a graph based on its control flow graph. In brief, a control flow graph is a directed graph made up of nodes representing basic blocks. Each basic block contains one or more statements from the program. There is an edge from basic block B_i to basic block B_j if program execution can flow from B_i to B_j. Traces through the control flow graph for inputs that produce correct results forms one collection of graphs and traces for inputs that produce incorrect results forms a second collection of graphs. The idea is to look for a discriminative subgraph between the two collections of graphs; this represents the lines of code that are, or are not, being executed when the bug occurs. The algorithm presented in [7] utilizes the LEAP algorithm [13] as a branch-and-bound heuristic on the search space of graphs that it examines; it is based on the observation that subgraphs with higher frequency are more likely to be discriminative. This algorithm was modified slightly to specifically scrutinize certain programming constructs and subsequently was tested in [8, 9]. This general approach to discriminative subgraph mining is applicable to the RTS game dataset and is discussed in more detail in the next section.

3. METHODOLOGY: DISCRIMINATIVE SUBGRAPH MINING

The algorithm we employed for discriminative subgraph mining is similar to the approach taken in [8, 9], but does not employ any heuristics specific to game data. Although we ran it sequentially, it easily lends itself to parallel or distributed processing.

Let C^+ and C^- represent two sets of (undirected or directed) graphs for which we want to find a discriminative subgraph; that is, we want to find a subgraph that appears in the graphs in C^- and does not appear in the graphs in C^+, or vice-versa. We shall refer to C^+ as the positive graphs and C^- as the negative graphs although this naming convention has no direct semantic correlation to the classification of the graphs in those respective sets (e.g., 'winner' does not necessarily mean positive). The *function FindDiscriminativeGraph* (Algorithm 1) first removes non-discriminative edges from the graphs in both sets; since such edges appear in the graphs in both sets, they cannot be used to differentiate the graphs in the those sets. *FindDiscriminativeGraph* then calls *CreateDiscriminativeGraph* (Algorithm 2) to try to find a subgraph that is common to all graphs in C^-, but not common to all the graphs in C^+. If we are unable to find such a graph, then the function *RelaxedCreateDiscriminativeGraph* (Algorithm 3) is called, which relaxes the requirement that the subgraph we seek not be present in all of the C^+ graphs; instead the subgraph only has to not be present in $\alpha * |C^+|$ of the C^+ graphs, where α is a user-specified parameter (our default is $\alpha = 0.5$).

FindDiscriminativeGraph and *CreateDiscriminativeGraph* use a function called *Augment*; this function takes a subgraph G and adds to it an edge (and possibly a node) such that the source vertex exists in G, and the edge (and destination node) exists in all

graphs in subgraph collection S_1. In this way, a subgraph with an additional edge that exists in all elements of S_1 is created and considered by the algorithm.

Algorithm 1 $FindDiscriminativeGraph\ (C^+,\ C^-,\ \alpha,\ \beta)$

C^+: set of positive graphs

C^-: set of negative graphs

α : percentage of graphs that discriminative subgraph need not be present in C^+
   when relaxing conditions

β : percentage of graphs that discriminative subgraph need not be present in C^-
   when relaxing conditions

1: remove non-discriminative edges from graphs in C^+ and C^-;

2: $G = CreateDiscriminativeGraph(C^-, C^+)$;

3: **if** G is empty **then**

4:    $G = RelaxedCreateDiscriminativeGraph(C^-, C^+, |C^+| * \alpha)$;

5:    **if** G is empty **then**

6:      $G = CreateDiscriminativeGraph(C^+, C^-)$;

7:      **if** G is empty **then**

8:        $G = RelaxedCreateDiscriminativeGraph(C^+, C^-, |C^-| * \beta)$;

9:      **end-if**

10:    **end-if**

11: **end-if**

12: **return** G

If we still fail to find a discriminative subgraph, then the difference likely does not involve edges that are in all graphs in C^- and not in graphs in C^+, but rather involves edges in the C^+ graphs that are not in the C^- graphs. Thus, we again call $CreateDiscriminativeGraph$, but reverse the order of the parameters (C^+ and C^-) from our previous call. If we still fail to find a discriminative subgraph, we again call $RelaxedCreateDiscriminativeGraph$ and look for a subgraph that only has to not be

present in $\beta * |C^-|$ of the C^- graphs, where β is a user-specified parameter (our default

is $\beta = 0.5$).

Algorithm 2 $CreateDiscriminativeGraph$ (S_1, S_2)

S_1: set of graphs

S_2: set of graphs

  1:   $FreqSG$ = queue of 1-edge subgraphs in S_1;

  2:   **while** $FreqSG$ is not empty **do**

  3:       $G = FreqSG$.dequeue ( );

  4:       **if** G is not in any graph in S_2 **then**

  5:         return (G);

  6:       **end-if**

  7:     NewGraphs = Augment (G);

  8:       **for** each graph G' in NewGraphs **do**

  9:         $FreqSG$.enqueue (G');

10:       **end-for**

11:    **end-while**

12: **return** (empty graph)

It is possible that the resulting discriminative graph will be disconnected. Additionally, it could be the case that multiple subgraphs could qualify as a discriminative subgraph. The algorithm addresses both of these cases by returning the maximal discriminative subgraph; this result may be disconnected and will include all possible discriminative edges. It should be noted that it also is possible that our algorithm will not find any subgraph that meets the discriminative conditions. This could occur if the requirement that at least α (β) of the graphs in $C^- (C^+)$ must have at least one edge in common has not been satisfied.

The computational complexity of the process is dependent upon the number of graphs in each collection and the number of edges in each graph. As specified in line 1 of

CreateDiscriminativeGraph, we begin by examining each single edge from each graph in one of the graph collections. However, in lines 7-9 of that algorithm, we potentially build larger subgraphs that must be searched for; this is the subgraph isomorphism problem, which is *NP*-complete.

Algorithm 3 $RelaxedCreateDiscriminativeGraph$ (S_1, S_2, γ)

S_1: set of graphs

S_2: set of graphs

γ : threshold for number of graphs discriminative subgraph must be present in

  1:    $FreqSG$ = queue of 1-edge subgraphs in S_1;

  2:    **while** $FreqSG$ is not empty **do**

  3:        $G = FreqSG$.dequeue ( );

  4:        **if** G is in $< \gamma$ graph in S_2**then**

  5:          return (G);

  6:        **end-if**

  7:      NewGraphs = Augment (G);

  8:        **for** each graph G' in NewGraphs **do**

  9:          $FreqSG$.enqueue (G');

10:        **end-for**

11:     **end-while**

12:   **return** (empty graph)

---

4. EXPERIMENT AND RESULTS

In this section we discuss the details of an experiment we conducted to test the hypothesis that predictive analytics, specifically discriminative subgraph mining, can be employed to examine a collection of played strategy games and make recommendations as to what a player should do, and should not do, in order to increase the chances of winning the game in the future.

4.1. EXPERIMENTAL SETUP

The game that we selected is an online, multi-player RTS game called Interloper [14]. Interloper was chosen over more sophisticated RTS games like StarCraft because of its relatively limited set of action types which include: creating territory tiles, spawning drones, spawning blockades, creating units (e.g., sentinels, drones, defenders, destroyers, markers, bombs, blockades, and snipers), building structures, destroying targets, moving and positioning characters, removing characters, hitting characters, and exploding characters. We obtained a database of 19 played Interloper games from the game's developer. Each of these games contained the sequence of actions performed by each of two players, with a designation of which player won the game. Each action type in the data file had a documented integer encoding. The total number of moves (for both players) in a game in the dataset ranged from 183 to 5,338.

For each game in the dataset we created two individual files: one for the winner's moves and one for the loser's moves. The format for each of the data files that we created was modeled as a directed graph, one edge per line, where each vertex was an action, and an edge represented a consecutive sequence of (two) actions made in that game. As with games such as chess, we thought it would be interesting to analyze (and make recommendations for) the game in three phases: the beginning game, the middle game, and the end game. In chess there is no clear definition of when the middle game begins and ends, or when the end game begins. Similarly, we had no such guidelines for Interloper. Therefore, we simply divided each game file into the first third number of moves, the middle third number of moves, and the last third number of moves, and referred to these as phases 1, 2, and 3 of the games, respectively. Each phase was analyzed separately.

As described in the previous section, our discriminative subgraph mining algorithm would not find a discriminative subgraph unless a certain percentage of the graphs in each (C^- or C^+) "collection" had at least a certain percentage of edges in common. Therefore, we had to test small groups of games at a time. To make sure that we did not miss any possible common edges, we tested every combination of two winning and two losing graphs; that is, a pair of winning graphs played the role of C^+ in *FindDiscriminativeGraph* and a pair of losing graphs played the role of C^-. We then reversed the roles (i.e., a pair of losing graphs played the role of C^+ and a pair of winning graphs played the role of C^-). Depending on whether the discriminative subgraph was found in C^+ or C^- for the particular assignment to those parameters told us whether the moves should be recommended as something that should be done in order to increase the chance of winning (because it was a difference found in the winning graphs) or something that should not be done (because it was a difference found in the losing graphs).

To test the predictive accuracy of our method, we performed cross validation on the dataset of 19 played games. For phase 1, we used 5-fold cross validation. Five partitions were created, 4 of which contained 4 games and 1 of which contained 3 games; by 'game' we mean both the winner and loser for that game. A random number generator (www.random.org/lists/) was used to determine which games were assigned to each partition (with no duplication). For each of the 5 iterations of the 5-fold cross validation, the "training" dataset was formed from 4 of the partitions and the "test" dataset was the remaining partition; the roles of the partitions were rotated through each iteration of the 5-fold cross validation. Discriminative subgraphs were determined from all possible pairs of winning and losing games in the "training" dataset (i.e., 4 of the 5 partitions). This resulted

in a set of subgraphs that formed the recommendations for actions that should be done and a set of subgraphs which formed the recommendations for actions that should not be done in order to win the game.

The error rate was calculated as follows. If a recommendation for what should be done (subgraph) was found in one of the winning graphs in the test partition, it was counted as a true positive (TP); if instead that recommendation (subgraph) was found in one of the losing graphs in the test partition, it was counted as a false positive (FP). If a recommendation for what should not be done (subgraph) was found in one of the losing graphs in the test partition, it was counted as a true negative (TN); if instead that recommendation (subgraph) was found in one of the winning graphs in the test partition, it was counted as a false negative. The error rate was calculated as $1 - ((TP + TN) / (TP + TN + FP + FN))$, and was averaged over the five iterations of the 5-fold cross validation.

For phases 2 and 3 of the game, significantly fewer discriminative subgraphs were found than for phase 1; this will be discussed in the next section. Therefore, instead of creating 5 partitions for cross-validation, we only created 3 partitions: 2 partitions contained 6 games and 1 partition contained 7 games. Consequently, only 3 iterations were run in those cross validations instead of 5. As was done for phase 1, games still were randomly chosen for each partition for each test. All cross-validation tests (for all phases) were repeated 5 times.

It should be noted that the discriminative subgraph mining algorithm was implemented in Python 3.7. A combination of Python programs and bash scripts were

created for data file conversions and batch program executions. All programs were executed on a Dell Intel i7-7700 3.60 GHz 64 GB RAM Windows 10 PC.

4.2. EXPERIMENTAL RESULTS

Each of the three phases of the game was analyzed separately using cross-validation, with each cross-validation test repeated 5 times with randomized data (game) assignment for training and test data from the 19-game dataset. Table 1 shows the average error rate for each of the cross-validation tests for each phase, as well as the average error rate over each phase's 5 tests. The resulting predictive accuracy was good, considering that, in general, discriminative subgraphs can have very low frequencies. The collective recommendations (for moves that should be made and moves that should not be made) were accurate approximately 86.5%, 92.4%, and 98.7% of the time for phases 1, 2, and 3 of the game, respectively.

Table 1. Cross-Validation Test Results

Test No.	Phase 1 Avg. Error Rate	Phase 2 Avg. Error Rate	Phase 3 Avg. Error Rate
1	14.40%	10.60%	1.00%
2	13.40%	0.08%	1.60%
3	13.00%	9.10%	0.70%
4	13.50%	9.80%	2.00%
5	13.30%	8.50%	1.40%
Avg.	**13.32%**	**7.62%**	**1.34%**

It should be noted that the accuracy for phases 2 and 3 were likely much higher than for phase 1 in part because 3-fold (rather than 5-fold) cross validation testing was used

for those phases and because there were significantly fewer discriminative subgraphs to test in those phases.

For phase 1 of the game, when testing all pairs of 2 winning and 2 losing graphs, 2,333 discriminative subgraphs were found that constituted "should do" recommendations and 2,270 discriminative subgraphs were found that represented "should not do" recommendations. The average size of the "should do" recommendation subgraphs was 28 edges; the smallest had 1 edge and the largest had 170 edges. The average size of the "should not do" recommendation subgraphs was 22 edges; the smallest had 1 edge and the largest had 168 edges.

Of the ten most frequently recommended "should do" subgraphs, 3 contained 3 edges (i.e., 4 moves) and 4 contained 4-5 edges (i.e., 5-6 moves). In contrast, 5 of the 10 most frequently recommended "should not do" subgraphs contained only 1 edge (i.e., 2 moves) and 5 contained 2-3 edges (i.e., 3-4 moves). Thus, for this phase of the game, we are not able to provide quite as much information about what a player should not do as we can say about what a player should do.

The types of actions in the phase 1 discriminative subgraphs were predominantly only two types: creation of territory tiles and (fast) moves of a game character. In the Interloper game, creation of territory files can be considered an offensive action against one's opponent. Movement of a game character could be either an offensive or defensive action; the player's intent (e.g., moving away from danger versus moving to a more strategic position in the game space) cannot be deduced from the game data. Another observation that can be made from these particular discriminative subgraphs is a counter that is associated with both of these types of moves. For each game, the counter for each

type of action begins at 1 and is incremented by 1 each time that type of action occurs. The phase 1 discriminative subgraphs differed not only in sequences of territory tile creation and character movement, but also in how relatively early (or late) those actions occurred and in what succession. For example, an edge (2800029, 2800030) represents two tile creations with counters 29 and 30, indicating that these were tile creations that occurred well after the game had started (i.e., they were the 29th and 30th tile creations that this player made). Their occurrence in a discriminative subgraph would indicate that it either is or is not advisable to create so many tiles (back to back) in the first phase of the game.

For phase 2 of the game, when testing all pairs of 2 winning and 2 losing graphs, 250 discriminative subgraphs were found that represented "should do" recommendations and 213 discriminative subgraphs were found that characterized "should not do" recommendations. These were about 90% less than the respective numbers of subgraphs found in phase 1. This is not surprising as the number (and order) of different moves that a player could (and likely did) make increased at this point in the game, thereby reducing the number of graphs that had edges in common and could meet the criteria of *FindDiscriminativeGraph*. The average size of the "should do" recommendation subgraphs was 25 edges; the smallest had 1 edge and the largest had 274 edges. The average size of the "should not do" recommendation subgraphs was 14 edges; the smallest had 1 edge and the largest had 155 edges.

The most frequently recommended "should not do" subgraphs in phase 2 only contained a single edge (i.e., 2 moves); thus, there was a further decrease in the amount of information we could provide a player in terms of what not to do in order to win the game. In contrast, 3 of the top 6 most frequently recommended "should do" subgraphs contained

at least 11 edges (i.e., 12 moves). Overall, compared to phase 1, this can be seen as the ability to provide much more information about what a player should do in order to win the game during this phase. Unfortunately, again the types of actions that occurred in the discriminative subgraphs were limited, mostly moving a game character (although now at a slower speed than in phase 1); we had anticipated seeing more offensive actions during this phase of the game.

For the final phase of the game, 68 discriminative subgraphs were found that characterized "should do" recommendations; this was a 97% decrease from the number found in phase 1 and a 72% decrease from the number found in phase 2. In this phase, 36 discriminative subgraphs were found that represented "should not do" recommendations; this was a 98.4% decrease from the number of such subgraphs found in phase 1 and an 83% decrease from the number found in phase 2. As mentioned previously, the moves in this phase of the game likely varied more from game to game, and, as such, it became more difficult to meet the criteria stipulated in *FindDiscriminativeGraph*. The average size of the "should do" recommendation subgraphs was 22 edges, which was only slightly smaller than what had been seen in the other two phases; the smallest had 1 edge and the largest had 115 edges, which was by far the smallest of the three phases. The average size of the "should not do" recommendation subgraphs was 18 edges, which is the average size between what was seen for phases 1 and 2; the smallest had 1 edge and the largest had 145 edges, which was slightly smaller than in phase 2. There were 92% fewer discriminative subgraphs found in phase 3 than had been found in phase 1.

For phase 3, we finally saw some of the most frequently recommended "should not do" subgraphs have multiple edges (i.e., more than 2 moves); of the top 7 such subgraphs,

3 contained more than 4 edges, and 2 of those contained 9-10 edges. Amongst the top 7 most frequently recommended "should do" subgraphs, only 1 had a single edge; the average number of edges for the others in this list was 7 edges (i.e., 8 moves). Although we could provide more recommendations about what 'not to do' in phase 3 than for phases 1 and 2, we still could provide much more information about what 'to do' during this phase of the game.

The majority of the actions in the "should do" subgraphs still involved (slow) movement of a game character whereas the actions in the "should not do" subgraphs predominantly involved territory tile creation, removal of a game character, and/or positioning of a game character. Territory tile creation and removal of a game character can be considered offensive actions in Interloper; as mentioned previously, positioning of a game character could be for offensive or defensive purposes, which cannot be determined from the game data. We were surprised that none of the discriminative subgraphs (for any of the phases) included any defensive actions (e.g., spawning a blockade); however, there are by far more offensive types of actions in the game than defensive actions.

Of all the pairs of 2 winner and 2 loser graphs tested, only a few failed to produce a discriminative subgraph. There were no contradictory results; that is, it was never the case that a sequence of actions in essence would be both recommended and not recommended. Some test pairs produced the same results as other pairs; duplicates were not included in the counts of discriminative subgraphs reported for each phase. Some test pairs produced discriminative subgraphs that were subgraphs of other reported discriminative subgraphs; this was not unexpected since some test (game) pairs had edges in common.

5. SUMMARY AND CONCLUSIONS

Herein we tested the hypothesis that a form of predictive analytics, namely discriminative subgraph mining, can be used to examine a set of played strategy games and generate a set of recommendations that could be used to predict the chances of winning the game in the future. Using a dataset of played games of a multi-player, Real-Time Strategy (RTS) video game, Interloper, we modeled each game as a graph and found a collection of subgraphs that specified sequences of actions that players should, and should not, make in each of three phases of the game. Although the dataset only contained 19 games, the experimental results showed that the accuracy of our recommendations was high. Overall, our recommendations for our test game, Interloper, were more informative in terms of what a player should do at each of three phases of the game in order to win; however, we also were able to provide some information about what the player should not do. Most importantly, this study has served as a proof of concept that this approach may be a promising strategy for not only game predictive analytics, but also for other problem domains that involve direct and indirect resource generation and destruction.

6. FUTURE WORK

We plan to test our discriminative subgraph mining approach on other types of RTS games. If we have the success that we had with Interloper, we hope to establish a mapping between action types and assets in this genre of games so that a more generalized recommendation system can be developed. We also hope to explore ways to make the algorithms more efficient, perhaps applying some heuristics to reduce the search space that

are inherent to the nature of game data. Ultimately, we intend to abstract this strategy to other problem domains such as a health care disease tracking and prediction systems using the same foundation of analyzing examples of success and failure in order to make recommendations for future positive outcomes.

REFERENCES

[1] L. Kaufman, "The Relative Value of the Pieces," *Computer Chess Reports*, 4:2, pp. 33–34, 1994.

[2] M. Sturman, "Beware the Bishop Pair," *Computer Chess Reports*, 5:2, pp. 58–59, 1995.

[3] P. Braun, A. Cuzzocrea, T. Keding, C. Leung, A. Padzor, and D. Sayson, "Game Data Mining: Clustering and Visualization of Online Game Data in Cyber-Physical Worlds," in *proceedings of International Conference on Knowledge Based and Intelligent Information and Engineering Systems* KES '17, (Marseille, France), vol. 112, no. 2, pp. 2259–2268, 2017.

[4] A. Drachen, C. Thurau, J. Togelius, G. Yannakakis, and C. Bauckhage, "Game Data Mining," *Game Analytics: Maximizing the Value of Player Data*, (London, UK), pp. 205–253, Springer London, 2013.

[5] Z. Shao, Y. Hirayama, Y. Yamanishi, and H. Saigo, "Mining Discriminative Patterns from Graph Data with Multiple Labels and Its Application to Quantitative Structure-Activity (QSAR) Models," *Journal of Chemical Information Models*, vol. 55, no. 12, pp. 2519–2527, 2015.

[6] N. Jin, C. Young, and W. Wang, "Discriminative Subgraph Mining for Protein Classification," in *Computational Knowledge Discovery for Bioinformatics Research*, pp. 279–295, 2012.

[7] H. Cheng, D. Lo, Y. Zhou, X. Wang, and X. Yan, "Identifying Bug Signatures Using Discriminative Graph Mining," in *Proceedings of the 18th. International Symposium on Software Testing and Analysis* ISSTA '09, (Chicago, IL USA), pp. 141–151, ACM, 2009.

[8] J. Leopold, N. Eloe, and P. Taylor, "BugHint: A Visual Debugger Based on Graph Mining," in *Proceedings of the 24th International Conference on Visualization and Visual Languages* ICVVL '18, (San Francisco, CA, USA), pp. 109–118, 2018.

[9] J. Leopold, N. Eloe, J. Gould, and E. Willard, "A Visual Debugging Aid Based on Discriminative Graph Mining," in *Journal of Visual Languages and Computing*, to appear February 2019.

[10] S. Ranu, M. Hoang, and A. Singh, "Mining Discriminative Subgraphs from Global-state Networks," in *Proceedings of the 19th. ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* KDD '13, (Chicago, IL, USA), pp. 509– 517, ACM, 2013.

[11] A. Fuksova, O. Kuzelka, and A. Szaboova, "A Method for Mining Discriminative Graph Patterns," in *Proceedings of NIPS Machine Learning in Computational Biology Workshop,* 2013.

[12] N. Jin, and W. Wang, "LTS: Discriminative Subgraph Mining by Learning from Search History," in *Proceedings of IEEE 27th. International Conference on Data Engineering* ICDE '11, (Hannover, Germany), pp. 207–218, 2011.

[13] X. Yan, H. Cheng, J. Han, and P. Yu, "Mining Significant Graph Patterns by Leap Search," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* SIGMOD '08, (Vancouver, BC, Canada), pp. 433–444, ACM, 2008.

[14] "Interloper Game Description." http://interlopergame.com/. Accessed: 2019-12-01.

III. PREDICTIVE ANALYSIS OF REAL-TIME STRATEGY GAMES: A GRAPH MINING APPROACH

Isam A. Alobaidi[1], Jennifer L. Leopold[1], Ali A. Allami[2], Nathan W. Eloe[3], and Dustin Tanksley[4]

[1]Department of Computer Science, Missouri University of Science and Technology, Rolla, MO 65409

[2]Electrical Engineering & Computer Science Department, University of Missouri, Columbia, MO 65211

[3]School of Computer Science and Information Systems, Northwest Missouri State University, Maryville, MO, USA

[4]Department of Computer & Electrical Engineering, Missouri University of Science and Technology, Rolla, MO 65409

ABSTRACT

Machine learning and computational intelligence have facilitated the development of recommendation systems for a broad range of domains. Such recommendations are based on contextual information that is explicitly provided or pervasively collected. Recommendation systems often improve decision-making or increase the efficacy of a task. Real-Time Strategy (RTS) video games are not only a popular entertainment medium, they also are an abstraction of many real-world applications where the aim is to increase your resources and decrease those of your opponent. Using predictive analytics, which examines past examples of success and failure, we can learn how to predict positive outcomes for such scenarios. The goal of our research is to develop an accurate predictive recommendation system for multiplayer strategic games to determine recommendations for moves that a player should, and should not, make and thereby provide a competitive

advantage. Herein we compare two techniques, frequent and discriminative subgraph mining, in terms of the error rates associated with their predictions in this context. As proof of concept, we present the results of an experiment that utilizes our strategies for two particular RTS games.

1. INTRODUCTION

The ever-increasing expansion of information and communications technology has initiated a new era for the development of recommendation systems for a wide variety of application domains (e.g., entertainment, E-commerce, E-health, etc.). Recommendations could be for products or services that a customer might consider purchasing, treatments that a doctor might consider prescribing for a patient, or a sequence of actions that a robot should perform in a certain situation. Typically, the recommendations are based on an analysis of historical data, often characterized as positive and negative examples for the recommendation scenario. In order to be of value, recommendation systems must have high predictive accuracy.

Another venue where recommendation systems can be valuable is strategic games. Real-Time Strategy (RTS) games are a subgenre of strategy video games wherein the participants position and maneuver units (e.g., troops, robots, and drones) and structures under their control to secure areas and destroy their opponent's assets. In some games, the created entities can in turn create and destroy other entities. Hence the focal points of such games are: resource generation and destruction, and indirect control of units and structures (via other units and structures).

RTS games typically have a diverse set of resources which the player can deploy, basically offensive or defensive in nature, and a large variety of environments/storylines from which to select, often with a military science fiction theme; a popular and sophisticated example is StarCraft. The games are usually multi-player, with the winner determined by some criterion such as the player with the most assets at the end of a certain time period or by the last player remaining after all other players' assets have been depleted. Although the RTS game scenario is used for entertainment purposes, it can be abstracted as a model for real-world applications such as military battles, cyberinfrastructure networks that may need to be managed as they come under malicious attack, and even disease history/diagnosis systems which track a patient's symptoms, treatments, and disease progression over time.

In this study, we test the hypothesis that predictive analytics can be employed to examine a collection of played games and make recommendations to increase the chances of winning the game the next time a person plays. Using a database of played games, we model each of those games as a directed graph, and use frequent subgraph mining and discriminative subgraph mining, respectively, to look for patterns of moves that occurred in winning games; these form the basis of our recommendations for moves that a player should make. Similarly, we look for patterns of moves that occurred in losing games; those become the basis of our recommendations for moves that a player should not make. We test the accuracy of our two methods by partitioning our database of played games into training and test datasets, and testing for the occurrence of true positives, true negatives, false positives, and false negatives. We also compare these two methods against each other, in terms of error rate of predictions.

The organization of this paper is as follows. Section 2 provides a brief discussion of the main topics in this paper, including game data mining and data mining techniques used in predictive analytics. The particular algorithms that we used for frequent subgraph mining and discriminative subgraph mining are explained in more depth in Section 3. A description of the RTS game data that we used for testing our method is provided in Section 4. Our experimental method and results are discussed in Section 5. A summary of this research and consideration of future work is discussed in Section 6.

2. BACKGROUND

In this section we briefly discuss some of the related work that has been done in the fields of game data mining, frequent subgraph mining, and discriminative subgraph mining.

2.1. GAME DATA MINING

For years there has been interest in analyzing games played by others in order to become a more competitive player. In its earliest form, people sought to identify the moves in the game that led to desirable, rather than undesirable, outcomes. For many games it is not only the quantity of assets, but particular features of the assets in the game that must be considered (e.g., an asset's functionality and location). For example, in the game of chess, given the choice, it is usually better to have one bishop than three pawns; position of a piece on the game board is also important as a bishop that is blocked by other pieces may not be able to attack. A number of studies have been conducted wherein a database of played games is analyzed to determine the winning percentage under various scenarios

such as games in which one player has two bishops and no knights and the other player has two knights and no bishops after some point in a chess game; see [1, 2] for examples of such studies. Contemporary genres of games, such as RTS video games, have a much more sophisticated collection of assets (e.g., game pieces) than traditional games such as chess and the characteristics of the assets can be much more diverse. Accordingly, analysis of desirable asset acquisition and deployment throughout a game has become more complex and computationally expensive.

One objective of game data mining is to analyze a collection of played games and find patterns of moves that were made in winning (and possibly losing) games. Game data mining was the main focus of research in [3, 4, 5]. In [4] a method, Playtracer, for game analysis and improvement was proposed. A multidimensional scaling strategy was applied to cluster players and game states, and a detailed visual representation of the paths taken by players during the game was provided. Specifically, Classical Multidimensional Scaling (CMDS) [6] was used in order to visualize the paths. The Playtracer method showed mutual ways that players succeeded and failed, and enabled tracking a specific player's progress across multiple levels.

Two widely used data mining techniques, Classification and Regression Trees (CART) and artificial neural networks, were utilized in [5] to analyze a collection of game data (i.e., STEAM) for predictive purposes. CART is a decision tree algorithm that aims to build a predictive model based on the values of several inputs. Artificial neural networks also attempt to discover new patterns from inputs by subjecting them to a repetitive learning process. That method relied on the analysis of online reviews (e.g., number of screenshots and number of reviews of a specific action) to predict what should be done

next in a game. Another branch of game data mining, also known as game telemetry, involves analysis of the people who play the game and/or the personas they may create. There are databases of this information for various online games and mining software to analyze data such as the players' skill level and time spent having played the game; see [7] for an example of such software. Some analyses may try to relate features from a player's profile to his/her winning percentage and odds of winning future games. This area of study is not the focus of the research pursued herein; we do not consider any data related to a player's profile.

As is discussed in [8], the intentions of game data mining should be made clear. *Description* describes patterns found in the game data; similarly, *characterization* is a summation of some general features associated with the data. These patterns could be independent of whether they occurred in the winners' games or the losers' games, or whether the patterns occurred in a majority or a minority of the games in the dataset. Description and characterization are the fundamental, general goals of most data mining efforts. *Classification* (and *clustering*) are used to compare and organize some features of the data into classes; with game data this usually isn't necessary since we are most interested in classifications as winning and losing games, information which is already known. *Discrimination* seeks to identify the differences between groups of instances in the game data beyond just the classification of winning and losing. *Prediction* has the goal of providing a rule (or some form of guideline) that can be used as guidance for playing or forecasting the outcome of future games. The work presented in this study focuses on discrimination and prediction of game data.

2.2. DATA MINING TECHNIQUES USED IN PREDICTIVE ANALYTICS

Utilizing mathematical modeling, the field of predictive analytics examines past examples of success and failure to determine the variables that lead to successful outcomes and can be used to make predictions about future events. It has been used widely in the financial and insurance sectors. Here we briefly discuss some of the most common types of data mining methods used for predictive analytics.

Regression analysis: This method analyzes the relationship between a dependent variable and a set of independent variables. For game data the dependent variable would likely be the outcome of the game (i.e., win or lose) and the independent variables would be the various possible moves.

Rule induction: Rule induction methods such as association rule mining seek to find relationships between variables in the dataset. By applying association rule mining on only the winners' games, we could identify some actions that winning players did. Similarly, by mining the losers' games, we could find some actions common to losing players.

Decision trees: Decision trees are most often used for classification and can be thought of as a graphical depiction of a rule; each branch of a decision tree can be thought of as a separate rule consisting of a conjunction of the attribute predicates of nodes along that branch. One approach would be to construct decision trees from the winning games and losing games, respectively.

Clustering: Clustering is a way to categorize a collection of instances in order to look for patterns; groups are formed to maximize similarity between the instances within a group and to maximize dissimilarly between instances in different groups. Game data are already clustered into two groups: winners and losers. For the purpose of analyzing

successful (and unsuccessful) actions, we would likely attempt to form clusters of action sequences.

Neural networks: Neural networks are composed of a series of interconnected nodes that map a set of inputs into one or more outputs. The interconnections between inputs (which, for the game data, could be actions in the game) could be determined based on an analysis of the played games.

Most of the above methods would be computationally prohibitive, and would probably not yield useful results, for the RTS game data unless we employed some type of data reduction mapping, which subsequently could result in loss of useful, specific information.

2.3. SUBGRAPH MINING

Many problems can be modeled with graphs, wherein entities are represented as vertices and relationships between entities are represented as edges. When the relationship between two vertices has some semantic distinction of a predecessor and a successor, the edges are directed and hence the graph is considered directed. A played RTS game can be modeled as a directed graph where each action (e.g., move) is represented by a vertex and an edge represents two consecutive actions that were made in the game. By necessity, each vertex also must be identified by which player performed that action. The moves for one player do not form a strictly linear sequence because an action can generate multiple actions; for example, the player may create a drone which in turn simultaneously spawns 5 more drones, each of which becomes a new vertex, and 5 edges are created from the propagating drone vertex.

Subgraph mining is a technique used to discover a particular pattern in graphs. Two techniques will discuss here.

2.3.1. Frequent Subgraph Mining. Given a single (directed or undirected) graph, it can be useful to know which subgraphs occur at least n times where n is a user-specified threshold for frequency. Similarly, given a collection of graphs and a frequency threshold n, it may be important to know which subgraphs occur in at least n of those graphs. The process of answering this question is called frequent subgraph mining.

Several methods for frequent subgraph mining were presented in [9, 10, 11, 12]. Amongst many of the frequent subgraph mining algorithms that have been developed, computationally expensive extension/joining operations (to create larger candidate subgraphs from smaller frequent subgraphs) and false positive pruning (to reduce the search space) have been the biggest challenges that researchers have tried to address.

2.3.2. Discriminative Subgraph Mining. Discriminative subgraph mining seeks to find a subgraph that appears in one collection of graphs but does not appear in another collection of graphs. This approach has been used to study several problems including identifying chemical functional groups that trigger side-effects in drugs [13], classifying proteins by amino acid sequence [14], and identifying bugs in software [15, 16, 17]. Various discriminative subgraph mining algorithms are given in [15, 16, 17, 18, 19], some of which are tailored for particular problems; due to space limitations, they are not discussed in detail here.

3. METHODOLOGY

In this section we discuss the two graph mining methods that we utilized for a predictive recommendation system for strategic games.

3.1. FREQUENT SUBGRAPH MINING

One of the data mining techniques that we used to develop a predictive recommendation system for strategic games was frequent subgraph mining. As mentioned in the previous section, we modeled each played game as a graph where a vertex represented a move in the game and an edge represented two consecutive moves. A game graph was not a strictly linear sequence of edges because some moves in turn generated multiple moves (e.g., a move could create a monster that would in turn propagate additional monsters, each of which would result in a new vertex and edge). We then analyzed the collection of graphs (a dataset of played games) to find frequent subgraphs: sequences of moves that were common to several winners' games and sequences of moves that were common to several losers' games. In this section we first briefly provide some basic graph terminology that will facilitate discussion of the particular frequent subgraph algorithm that we utilized for our study.

3.1.1. Preliminaries. Let $\mathcal{G} = \{G_1, G_2, ..., G_n\}$ be a set of linear directed graphs which represents the historical data. Each G_i represents a single game's moves, such that $G_i = (V_i, E_i)$ where V_i represents a node labeled as an action code of a player's move, while an edge in E_i represents two consecutive moves. A graph $T = (V_T, E_T)$ is a subgraph of $G_i = (V_i, E_i)$ iff $V_T \subseteq V_{G_i}, E_T \subseteq E_{G_i}$.

Definition 1. Let $T = (V_T, E_T)$ be a subgraph of a graph $G_i = (V_i, E_i)$. A subgraph isomorphism of T to G_i is an injective function $f: V_T \longrightarrow V_{G_i}$ satisfying $(f(u), f(v)) \in E_{G_i}$ for all edges $(u, v) \in E_T$. Intuitively, a subgraph isomorphism is a mapping from V_T to V_{G_i} such that each edge in E_{G_i} is mapped to a single edge in E_T and vice versa.

Problem 1. Given a set of graphs $\mathcal{G}$, the frequent subgraph isomorphism mining problem is defined as finding all subgraphs T in G such that $t_G(T) \geq \tau$, where $t_G(T)$ is the number of graphs in G that contain T and τ is the user-specified threshold.

Problem 2. Given a set of graphs $\mathcal{G}$ such that each G_i is divided into three phases G_{i1}, G_{i1}, G_{i3} and a frequent subgraph T, the frequent phase mining problem is defined as finding all subgraphs T in G_{ij} such that $t_{G_{ij}}(T) \geq \tau$, where τ is the user-specified threshold.

In our case, problem (2) counts the actual frequency (i.e., occurrences) of each subgraph provided that it is greater than or equal to τ. However, this may not be useful in various cases [12, 20], while others necessitate the exact number of occurrences (like graph indexing in [21]. The choice of three for number of phases was an arbitrary decision influenced by board games such as chess that have tradictionally been analyzed in terms of the moves made in the beginning, middle, and end of the game.

3.1.2. GraMi Algorithm. For the purpose of generating candidate subgraphs, a variety of frequent subgraph mining and subgraph extension algorithms have been developed, as discussed in previous work [12, 22, 23]. In particular, GraMi [23] is one of the most efficient methods and is the foundation for the work presented in this paper. The key ideas behind GraMi are briefly outlined here. Algorithm 1 is used to find a set of all frequent edges fEdges in the collection of graphs $= \{\boldsymbol{G_{i=1,...,n}}\}$. All of these frequent edges have support greater than or equal to the assigned threshold $\boldsymbol{\tau}$. Because of the anti-

monotone property, only frequent edges will be considered when finding the frequent subgraphs.

Algorithm 1 Frequent Subgraph Mining - FSM

Input $\mathcal{G} = \{G_{i=1,\ldots,n}\}$ and frequency threshold τ
Output All $fSubgraphs\ S(G_i)$ with the support $\geq \tau$

1:   $fSubgraphs \leftarrow \phi$
2:   $Count = 0$
3:    **for each** edge e_{G_i} **do**
4:     **if** $e_{G_i} = e_{G_{i+1}}$ **then**
5:      $Count + +$
6:     **end-if**
7:     **if** $Count \geq \tau$ **then**
8:      $fEdges \leftarrow fEdges \cup e_{G_i}$
9:     **end-if**
10:   **end-for**
11:   **for each** $e \in fEdges$ **do**
12:    $fSubgraphs \leftarrow fSubgraphs \cup SubE(e,\mathcal{G},\tau,fEdges)$
13:    Remove e from $\mathcal{G}$ and $,fEdges$
14:   **end-for**
15: **return** $fSubgraphs$

Algorithm 2 is given each frequent edge to extend it to a new frequent subgraph. This is done by incorporating that edge with another subgraph. All extensions created in previous iterations are excluded by utilizing the *DFScode* canonical form that was introduced for gSpan [22]. The set *Candidate* in Algorithm 2 will include all the new subgraph extensions that had not been considered in prior iterations.

In subsequent steps, any new subgraph extension within the set *Candidate* that does not meet the support threshold τ requirement will be discarded. If any of those subgraphs

had been extended, it would produce a new non-frequent subgraph according to the anti-monotonic property.

Algorithm 2 Subgraph Extension - $SubE$

Input $fSubgraphs\ S, fEdges$ and frequency threshold τ

Output All Sub_{new} with the support $\geq \tau$

1:   $Sub_{new} \leftarrow \phi$

2:   $Candidate \leftarrow \phi$

3:    **for each** $e \in fEdges$ and $n \in fSubgraphs$ **do**

4:      **if** e fit to extend n **then**

5:       Generate a new subgraph $ExtS$

6:       **if** $ExtS$ exist in $\mathcal{G}$ and not generated before **then**

7:        $Candidate \leftarrow Candidate \cup ExtS$

8:       **else**

9:        remove $ExtS$

10:       **end-if**

11:      **end-if**

12:    **end-for**

13:    **for each** $ExtS \in Candidate$ **do**

14:      **if** $ExtS$ count in $\mathcal{G} \geq \tau$ **then**

15:       $Sub_{new} \leftarrow Sub_{new} \cup SubE(ExtS, \mathcal{G}, \tau, fEdges)$

16:      **end-if**

17:    **end**

18: **return** Sub_{new}

3.1.3. Using Frequent Subgraphs to Make Recommendations. In this section we discuss the algorithms that we utilized in order to mine the game dataset for frequent subgraphs and build a recommendation system. The task of finding the number of occurrences for each subgraph was carried out using Algorithm 3.

The mechanism for node-finding was used for matching the first node of a candidate subgraph with its occurrence in the original dataset. The objective of this process

was to determine the starting point for conducting a depth-first search (*DFSearch*) to find all similar subgraphs in the winner (or loser) graph collection. These results were stored temporarily in a *temp* set to compute their replication in the subsequent steps, and then the final result was placed within *ExactFSG* set.

Algorithm 3 Exact Subgraph Frequency

Input $\mathcal{G} = \{G_{i=1,\ldots,n}\}$, $fSubgraphs$ S and frequency threshold τ
Output All the Exact Frequent Subgraph with their frequency

1:    $Count = 0$
2:    **for** $i = 1 \rightarrow$ all graphs in ($fSubgraphs$) **do**
3:      $frq = 0$
4:      **for** $j = 1 \rightarrow$ all graphs in ($\mathcal{G}$) **do**
5:        **if** findnode (G_j, $fSubgraphs_i$) $\neq 0$ **do**
6:          $temp \leftarrow$ dfsearch (G_j, $fSubgraphs_i$)
7:          **if** $temp \geq size(fSubgraphs_i)$ & $isisomorphic(fSubgraphs_i, G_j)$ **do**
8:            $frq++$
9:          **end-if**
10:        **end-if**
11:       **end-for**
12:      **if** $frq \geq \tau$ **do**
13:       $count++$
14:       $ExactFSG(count) \leftarrow fSubgraphs_i$
15:      **end-if**
16:    **end-for**
17:   **return** $ExactFSG$

3.2. DISCRIMINATIVE SUBGRAPH MINING

The algorithm we employed for discriminative subgraph mining is similar to the approach taken in [16, 17], but does not employ any heuristics specific to game data. Although we ran it sequentially, it easily lends itself to parallel or distributed processing.

Let C^+ and C^- represent two sets of (undirected or directed) graphs for which

we want to find a discriminative subgraph; that is, we want to find a subgraph that appears

in the graphs in C^- and does not appear in the graphs in C^+, or vice-versa. We shall refer

to C^+ as the positive graphs and C^- as the negative graphs although this naming convention

has no direct semantic correlation to the classification of the graphs in those respective sets

(e.g., 'winner' does not necessarily mean positive).

Algorithm 4 $FindDiscriminativeGraph\ (C^+,\ C^-,\ \alpha,\ \beta)$

C^+: set of positive graphs

C^-: set of negative graphs

$\alpha\ $: percentage of graphs that discriminative subgraph need not be present in C^+
      when relaxing conditions

$\beta\ $: percentage of graphs that discriminative subgraph need not be present in C^-
      when relaxing conditions

  1:   remove non-discriminative edges from graphs in C^+ and C^-;

  2:   $G\ =\ CreateDiscriminativeGraph(C^-,C^+)$;

  3:   **if** G is empty **then**

  4:     $G\ =\ RelaxedCreateDiscriminativeGraph(C^-,C^+,|C^+|*\alpha)$;

  5:     **if** G is empty **then**

  6:       $G\ =\ CreateDiscriminativeGraph(C^+,C^-)$;

  7:       **if** G is empty **then**

  8:         $G\ =\ RelaxedCreateDiscriminativeGraph(C^+,C^-,|C^-|*\beta)$;

  9:       **end-if**

10:     **end-if**

11:   **end-if**

12:   **return** G

The function *FindDiscriminativeGraph* (Algorithm 4) first removes non-

discriminative edges from the graphs in both sets; since such edges appear in the graphs in

both sets, they cannot be used to differentiate the graphs in the those sets.

FindDiscriminativeGraph then calls *CreateDiscriminativeGraph* (Algorithm 5) to try to find a subgraph that is common to all graphs in C^-, but not common to all the graphs in C^+. If we are unable to find such a graph, then the function *RelaxedCreateDiscriminativeGraph* (Algorithm 6) is called, which relaxes the requirement that the subgraph we seek not be present in all of the C^+ graphs; instead the subgraph only has to not be present in $\alpha * |C^+|$ of the C^+ graphs, where α is a user-specified parameter (our default is $\alpha = 0.5$).

Algorithm 5 $CreateDiscriminativeGraph\ (S_1, S_2)$

S_1: set of graphs
S_2: set of graphs

  1: $FreqSG$ = queue of 1-edge subgraphs in S_1;
  2:   **while** $FreqSG$ is not empty **do**
  3:       $G = FreqSG$.dequeue ( );
  4:       **if** G is not in any graph in S_2 **then**
  5:         return (G);
  6:       **end-if**
  7:     NewGraphs = Augment (G);
  8:       **for** each graph G' in NewGraphs **do**
  9:         $FreqSG$.enqueue (G');
10:       **end-for**
11:    **end-while**
12: **return** (empty graph)

     FindDiscriminativeGraph and *CreateDiscriminativeGraph* use a function called *Augment*; this function takes a subgraph G and adds to it an edge (and possibly a node) such that the source vertex exists in G, and the edge (and destination node) exists in all graphs in subgraph collection S_1. In this way, a subgraph with an additional edge that exists in all elements of S_1 is created and considered by the algorithm.

If we still fail to find a discriminative subgraph, then the difference likely does not involve edges that are in all graphs in C^- and not in graphs in C^+, but rather involves edges in the C^+ graphs that are not in the C^- graphs. Thus, we again call $CreateDiscriminativeGraph$, but reverse the order of the parameters (C^+ and C^-) from our previous call. If we still fail to find a discriminative subgraph, we again call $RelaxedCreateDiscriminativeGraph$ and look for a subgraph that only has to not be present in $\beta * |C^-|$ of the C^- graphs, where β is a user-specified parameter (our default is $\beta = 0.5$).

Algorithm 6 $RelaxedCreateDiscriminativeGraph$ (S_1, S_2, γ)

S_1: set of graphs
S_2: set of graphs
γ : threshold for number of graphs discriminative subgraph must be present in
  1:   $FreqSG$ = queue of 1-edge subgraphs in S_1;
  2:   **while** $FreqSG$ is not empty **do**
  3:       $G = FreqSG$.dequeue ( );
  4:       **if** G is in $< \gamma$ graph in S_2 **then**
  5:         return (G);
  6:       **end-if**
  7:     NewGraphs = Augment (G);
  8:       **for** each graph G' in NewGraphs **do**
  9:         $FreqSG$.enqueue (G');
10:       **end-for**
11:    **end-while**
12:   **return** (empty graph)

It is possible that the resulting discriminative graph will be disconnected. Additionally, it could be the case that multiple subgraphs could qualify as a discriminative subgraph. The algorithm addresses both of these cases by returning the maximal

discriminative subgraph; this result may be disconnected and will include all possible discriminative edges. It should be noted that it also is possible that our algorithm will not find any subgraph that meets the discriminative conditions. This could occur if the requirement that at least α (β) of the graphs in C^- (C^+) must have at least one edge in common has not been satisfied.

The computational complexity of the process is dependent upon the number of graphs in each collection and the number of edges in each graph. As specified in line 1 of *CreateDiscriminativeGraph*, we begin by examining each single edge from each graph in one of the graph collections. However, in lines 7-9 of that algorithm, we potentially build larger subgraphs that must be searched for; this is the subgraph isomorphism problem, which is *NP*-complete.

4. DATA DESCRIPTION

Interloper [24] and StarCraft II [25] are online multiplayer real-time strategy (RTS) games. These games allow the creation and deployment of entities, and the destruction of an opponent's entities. A player wins the game when the other player's entities/assets have been destroyed or the other player cannot create any more assets. In this study, a dataset of 19 played games involving 2 players was obtained for Interloper, and a dataset of 228 played games involving 2 players was obtained for StarCraft II. Each of these games contained the sequence of actions performed by each of two players, with a designation of which player won the game. Each move in the Interloper's/StarCraft II dataset was encoded with 6-7 digits. Certain digits represented the action type, other digits represented the

player *ID*, and other digits represented a counter (distinguishing how many times a particular action had been executed by a particular player).

As with games such as chess, we thought it would be interesting to analyze (and make recommendations for) the games in three phases: the beginning game, the middle game, and the end game. In chess there is no clear definition of when the middle game begins and ends, or when the end game begins. Similarly, we had no such guidelines for Interloper and StarCraft. Therefore, we simply divided each game file into the first third number of moves, the middle third number of moves, and the last third number of moves, and referred to these as phases 1, 2, and 3 of the games, respectively. Each phase was analyzed separately.

5. EXPERIMENTAL EVALUATION

In this section we discuss the details of an experiment we conducted to test the hypothesis that predictive analytics, specifically frequent and discriminative subgraph mining, can be employed to examine a collection of played strategy games and make recommendations as to what a player should do, and should not do, in order to increase the chances of winning the game in the future.

5.1. EXPERIMENTAL SETUP

We analyzed the game in terms of three phases (i.e., beginning game, middle game, and end game) by dividing each game into three equal parts; the total number of moves in a game (by both the winner and the loser) ranged from 183 to 5,338 in Interloper and from 595 to 5,245 in StarCraft. For each of the three phases analyzed, 80% of the data were used

for training and the remaining 20% were used for testing with 5-fold cross validation. A random number generator (www.random.org/lists/) was used to determine which games were assigned to each partition (with no duplication). This process was repeated five times for each phase in order to avoid any bias during the measure of error rate. Accuracy was used to evaluate the closeness of the measured value to the true value. Equation 1 is the mathematical formula of accuracy where TP is true positive, TN is true negative, FP is false positive, and FN is false negative.

$$Accuracy = \frac{TP + TN}{TP + FP + FN + TN} \tag{1}$$

If a recommendation for what should be done (subgraph) was found in one of the winning graphs in the test partition, it was counted as a true positive (TP); if instead that recommendation (subgraph) was found in one of the losing graphs in the test partition, it was counted as a false positive (FP). If a recommendation for what should not be done (subgraph) was found in one of the losing graphs in the test partition, it was counted as a true negative (TN); if instead that recommendation (subgraph) was found in one of the winning graphs in the test partition, it was counted as a false negative. The error rate was calculated as $1 - Accuracy$, and was averaged over the five iterations of the 5-fold cross validation.

For phase 1 of the Interloper game, we used 5-fold cross validation. Five partitions were created, 4 of which contained 4 games and 1 of which contained 3 games; by 'game' we mean both the winner and loser for that game. For phases 2 and 3 of the Interloper game, significantly fewer discriminative subgraphs were found than for phase 1; this will

be discussed in the next section. Therefore, instead of creating 5 partitions for cross-validation, we only created 3 partitions: 2 partitions contained 6 games and 1 partition contained 7 games. Consequently, only 3 iterations were run in those cross validations instead of 5. As was done for phase 1, games still were randomly chosen for each partition for each test. The same procedure of 5-fold cross validation was utilized for the three phases of StarCraft game. In each phase of this game five partitions were created, 4 of which contained 46 games and 1 of which contained 44 games; 5 iterations were run in those cross validations. This was unlike what happened with phase 2 and 3 of Interloper game because this time enough discriminative subgraphs were found for this purpose.

5.2. EXPERIMENT RESULTS

In this section we present the results of analyzing the Interloper and StarCraft games dataset using both frequent subgraph mining and discriminative subgraph mining. The algorithms of frequent subgraph mining presented in Section 3.1 were (collectively) implemented in Matlab and Java. The algorithms of discriminative subgraph mining presented in Section 3.2 were implemented in Python 3.7. A combination of Python programs and bash scripts were created for data file conversions and batch program executions. Our experiments were executed on an Intel(R) Core (TM) i7-6700 CPU@3.40GHz computer with 32GB memory.

5.2.1. FSM - Experimental Results. Tables 1, 2, 3, and 4 show some of the experimental results of frequent subgraph mining using a threshold of 2 for the winner and loser datasets consisting of 19 Interloper's games and a threshold of 10 for the winner and loser datasets consisting of 228 StarCraft's games.

Table 1: Winner Data of FSM – Interloper Game

Winner Subgraph	Frequency	Classification
2810003 2810004 2810005 2810006	13	offensive
2800002 2400005 2400006 2800003 2400007 2400008	28	move
2110001 2610001 2110002	21	defensive
2400001 2400002 2400003 2800001	16	move

Table 2: Loser Data of FSM – Interloper Game

Winner Subgraph	Frequency	Classification
2810010 2810011 2810012 2810013	15	offensive
2810002 2710001 2810003 2810004 2810005	22	offensive
2110011 2110012 2110013 2810014 2110015 2810016	26	defensive
2410008 2810010 2410012	33	move

Table 3: Winner Data of FSM – StarCraft II Game

Winner Subgraph	Frequency	Classification
1262215 1262216 1262217 1262218	73	offensive
1272171 1272172 1272173 1272174 1272175	26	offensive
1572171 1572172 1272173 1572174	58	move
1772219 1272220 1772221 1772222 1272223 1772224 1772225	63	defensive

Table 4: Loser Data of FSM – StarCraft II Game

Winner Subgraph	Frequency	Classification
1292114 1772278 1772279 1772280 1772281 1772282	85	offensive
1592115 1292116 1592117 1592118	32	move
1762199 1262200 1762201 1262202 1762203 1762204	46	defensive
3622407 3622408 3622409	17	offensive

The first and second columns show the actions in the frequent subgraphs with their number of occurrences from the entire dataset, respectively. The third column in each table is a classification of the majority of that subgraph's actions; we classified that game's

actions as either offensive, defensive, or movement (of an entity in the game space). These results were obtained by performing 5-fold cross-validation, repeated five times. Each time, for the 19-game Interloper dataset, 15 games were selected randomly (without duplication) for training, and the remaining 4 games were used for testing. For the 228-game of StarCraft's dataset, 182 games were selected randomly (without duplication) for training, and the remaining 46 games were used for testing.

The size of the resulting frequent subgraphs ranged from two nodes with one edge to twenty-eight nodes with twenty-seven edges. All of the two-node subgraphs were ignored because of the limited information they provide for the recommendation objective (i.e., only two moves) compared to larger subgraphs. Frequent subgraphs that were found in the winner graphs indicate actions that are recommended for a player to do, whereas frequent subgraphs that were found in the loser graphs indicate actions that are recommended that a player should not do. The benefit of the counter attached to each action reflects the relative number of times the player had made that type of move in that game. Characterizing the actions, such as offensive or defensive, gives a general notion of the strategy the player is employing in that sequence and would facilitate mapping one game's actions to another's (e.g., mapping Interloper's offensive actions to StarCraft's offensive actions).

Tables 5 and 6 show the average error rate for each of the cross-validation tests for each phase, as well as the average error rate over each phase's 5 tests for Interloper and StarCraft, respectively. The resulting predictive accuracy was not good for frequent subgraph mining; in general, frequent subgraphs can have very low frequencies at times and high frequencies at other times. The collective recommendations (for moves that

should be made and moves that should not be made) were accurate approximately

50.28%, 39.67%, and 14.32% of the time for phases 1, 2, and 3 of Interloper, respectively.

It should be noted that this ratio improved slightly to 45.13.% when measuring the error

rate for phase 1 of StarCraft but increased to 40.1% and 24.1% for phase 2 and 3. We

attribute this increase in the error rate to the increase in the number of winner frequent

subgraphs found in the loser dataset and the loser frequent subgraphs found in the winner

dataset.

Table 5. Cross-Validation Test Results of FSM – Interloper Game

Test No.	Phase 1 Avg. Error Rate	Phase 2 Avg. Error Rate	Phase 3 Avg. Error Rate
1	50.21%	36.67%	7.87%
2	45.91%	41.92%	24.08%
3	47.94%	42.6%	14.33%
4	54.83%	38.86%	11.23%
5	52.52%	38.27%	13.99%
Avg.	**50.28%**	**39.67%**	**14.32%**

Table 6. Cross-Validation Test Results of FSM - StarCraft II Game

Test No.	Phase 1 Avg. Error Rate	Phase 2 Avg. Error Rate	Phase 3 Avg. Error Rate
1	45.53%	44.83%	24.26%
2	45.71%	41.62%	22.26%
3	45.42%	38.33%	29.02%
4	43.6%	40.71%	22.71%
5	45.42%	34.55%	22.1%
Avg.	**45.13%**	**40.1%**	**24.1%**

5.2.2. DSM - Experimental Results. Tables 7 and 8 show the average error rate

for each of the cross-validation tests for each phase, as well as the average error rate over

each of the phase 5 tests for Interloper and StarCraft, respectively. The resulting predictive accuracy was good, considering that, in general, discriminative subgraphs can have very low frequencies. The collective recommendations (for moves that should be made and moves that should not be made) had error rates of approximately 13.32%, 7.62%, and 1.34% of the time for phases 1, 2, and 3 of Interloper, respectively. It should be noted that this ratio improved to 10.52%, 7.38%, and 2.52% when measuring the error rate for the first, second, and third phase of StarCraft. We attribute this decrease in the error rate to the decrease in the number of winner discriminative subgraphs found in the loser dataset and the loser discriminative subgraphs found in the winner dataset.

Table 7. Cross-Validation Test Results of DSM – Interloper Game

Test No.	Phase 1 Avg. Error Rate	Phase 2 Avg. Error Rate	Phase 3 Avg. Error Rate
1	14.40%	10.60%	1.00%
2	13.40%	0.08%	1.60%
3	13.00%	9.10%	0.70%
4	13.50%	9.80%	2.00%
5	13.30%	8.50%	1.40%
Avg.	**13.32%**	**7.62%**	**1.34%**

Table 8. Cross-Validation Test Results of DSM - StarCraft II Game

Test No.	Phase 1 Avg. Error Rate	Phase 2 Avg. Error Rate	Phase 3 Avg. Error Rate
1	10.70%	7.70%	1.90%
2	9.80%	6.70%	3.90%
3	10.10%	8.40%	1.70%
4	10.90%	7.60%	1.20%
5	11.10%	6.50%	3.90%
Avg.	**10.52%**	**7.38%**	**2.52%**

For phase 1 of the Interloper game, when testing all pairs of 2 winning and 2 losing graphs, 2,333 discriminative subgraphs were found that constituted "should do" recommendations and 2,270 discriminative subgraphs were found that represented "should not do" recommendations. The average size of the "should do" recommendation subgraphs was 28 edges; the smallest had 1 edge and the largest had 170 edges. The average size of the "should not do" recommendation subgraphs was 22 edges; the smallest had 1 edge and the largest had 168 edges.

For phase 1 of StarCraft, when testing all pairs of 2 winning and 2 losing graphs, 33,981 discriminative subgraphs were found that constituted "should do" recommendations and 28,503 discriminative subgraphs were found that represented "should not do" recommendations. The average size of the "should do" recommendation subgraphs was 146 edges; the smallest had 1 edge and the largest had 297 edges. The average size of the "should not do" recommendation subgraphs was 82 edges; the smallest had 1 edge and the largest had 268 edges.

For phase 2 of Interloper, when testing all pairs of 2 winning and 2 losing graphs, 250 discriminative subgraphs were found that represented "should do" recommendations and 213 discriminative subgraphs were found that characterized "should not do" recommendations. These were about 90% less than the respective numbers of subgraphs found in phase 1. This is not surprising as the number (and order) of different moves that a player could (and likely did) make increased at this point in the game, thereby reducing the number of graphs that had edges in common and could meet the criteria of *FindDiscriminativeGraph*. The average size of the "should do" recommendation subgraphs was 25 edges; the smallest had 1 edge and the largest had 274 edges. The average

size of the "should not do" recommendation subgraphs was 14 edges; the smallest had 1 edge and the largest had 155 edges.

The situation was not similar in phase 2 of StarCraft, where there were about 42% less than the respective numbers of subgraphs found in phase 1. When testing all pairs of 2 winning and 2 losing graphs, 14,264 discriminative subgraphs were found that represented "should do" recommendations and 12,656 discriminative subgraphs were found that characterized "should not do" recommendations. The average size of the "should do" recommendation subgraphs was 109 edges, which was only slightly smaller than what had been found in phase 1 and 3; the smallest had 1 edge and the largest had 384 edges. The average size of the "should not do" recommendation subgraphs was 94 edges; the smallest had 1 edge and the largest had 285 edges.

For the final phase of Interloper, 68 discriminative subgraphs were found that characterized "should do" recommendations; this was a 97% decrease from the number found in phase 1 and a 72% decrease from the number found in phase 2. In this phase, 36 discriminative subgraphs were found that represented "should not do" recommendations; this was a 98.4% decrease from the number of such subgraphs found in phase 1 and an 83% decrease from the number found in phase 2. As mentioned previously, the moves in this phase of the game likely varied more from game to game, and, as such, it became more difficult to meet the criteria stipulated in $FindDiscriminativeGraph$. The average size of the "should do" recommendation subgraphs was 22 edges, which was only slightly smaller than what had been seen in the other two phases; the smallest had 1 edge and the largest had 115 edges, which was by far the smallest of the three phases. The average size of the "should not do" recommendation subgraphs was 18 edges, which is the average size

between what was seen for phases 1 and 2; the smallest had 1 edge and the largest had 145 edges, which was slightly smaller than in phase 2. There were 92% fewer discriminative subgraphs found in phase 3 than had been found in phase 1. In the final phase of StarCraft, 7,047 discriminative subgraphs were found that characterized "should do" recommendations; this was a 75% decrease from the number found in phase 1 and a 39% decrease from the number found in phase 2. In this phase, 4,550 discriminative subgraphs were found that represented "should not do" recommendations; this was an 86% decrease from the number of such subgraphs found in phase 1 and a 64% decrease from the number found in phase 2. The average size of the "should do" recommendation subgraphs was 119 edges; the smallest had 1 edge and the largest had 255 edges. The average size of the "should not do" recommendation subgraphs was 73 edges, which is close to the average size between what was seen for phases 1 and 2; the smallest had 1 edge and the largest had 246 edges.

Instead of looking at all the result subgraph (recommendations), the user should be able to view only the top k "should" and "should not do" subgraphs, where k is a user-specified parameter. For example, among the top ten frequently recommended "should do" subgraphs in phase 1 of Interloper, 3 had 3 edges (i.e., 4 moves) and 4 contained 4-5 edges (i.e., 5-6 moves). In contrast, 5 of the 10 most frequent "should not do" subgraphs contained only 1 edge (i.e., 2 moves) and 5 contained 2-3 edges (i.e., 3-4 moves). It should be noted that the "should" and "should not do" subgraphs can vary in the number of edges they contain; thus, we may not be able to provide as much information about what a player should not do as we can say about what a player should do (or vice versa). The type of action can have an important role in characterizing a recommended subgraph (i.e.,

predominantly offensive, defensive, or movement). In the Interloper or StarCraft game, creation of territory files likely is considered an offensive action against one's opponent. Movement of a game character could be either an offensive or defensive action; the player's intent (e.g., moving away from danger versus moving to a more strategic position in the game space) cannot be deduced from the game data.

Another observation that can be made from discriminative subgraphs is a counter that is associated with both of these types of moves. For each game, the counter for each type of action begins at 1 and is incremented by 1 each time that type of action occurs. For example, edges (4921156, 3881100, 4921157, 4921158) in phase 2 of a StarCraft game represent three factory creations (actions beginning 492) with counters 156, 157, and 158 (where the counter is initialized to 100), indicating that these particular factories were built well after the game had started. Their occurrence in a discriminative subgraph would indicate that it either is or is not advisable to build so many factories early in the game.

6. CONCLUSION AND FUTURE WORK

The use of recommendation systems has become widespread in our society. In general, they examine historical data and try to predict what should be done in the future. Herein we have applied graph data mining techniques, frequent and discriminative subgraph mining, to multiplayer, Real-Time Strategy (RTS) video games, Interloper and StarCraft, to develop a system that can provide recommendations in order to improve a player's chances of winning a future game. We modeled each game as a graph and found a collection of subgraphs that specified sequences of actions that players should, and should

not, make in each of three phases of the game. When testing datasets of both games, experimental results of discriminative subgraph mining showed that the accuracy of our recommendations was high (an average of 93% accuracy for all three phases of both games), and better than when using frequent subgraph mining. Overall, our recommendations for our test games were more informative in terms of what a player should do at each of three phases of the game in order to win; however, we also were able to provide some information about what the player should not do. Most importantly, this study has served as a proof of concept that the discriminative subgraph approach may be a promising strategy for not only game predictive analytics, but also for other problem domains that involve direct and indirect resource generation and destruction.

In the future we plan to establish a mapping between action types and assets in this genre of games so that a more generalized recommendation system can be developed. We also hope to explore ways to make the algorithms more efficient, perhaps applying some heuristics to reduce the search space that are inherent to the nature of game data. Ultimately, we intend to abstract this strategy to other problem domains such as a health care disease tracking and prediction systems using the same foundation of analyzing examples of success and failure in order to make recommendations for future positive outcomes.

REFERENCES

[1]   L. Kaufman, "The Relative Value of the Pieces," *Computer Chess Reports*, 4:2, pp. 33–34, 1994.

[2]   M. Sturman, "Beware the Bishop Pair," *Computer Chess Reports*, 5:2, pp. 58–59, 1995.

[3]   A. Drachen, C. Thurau, J. Togelius, G. N. Yannakakis, and C. Bauckhage, "Game Data Mining," in *Game Analytics: Maximizing the Value of Player Data,* (London, UK), pp. 205–253, Springer, 2013.

[4]   E. Andersen, Y.-E. Liu, E. Apter, F. Boucher-Genesse, and Z. Popović, "Gameplay Analysis Through State Projection," in *Proceedings of the 5th. International Conference on the Foundations of Digital Games*, (Monterey, CA, USA), pp. 1–8, ACM, 2010.

[5]   H.-N. Kang, H.-R. Yong, and H.-S. Hwang, "A Study of Analyzing on Online Game Reviews using a Data Mining Approach: STEAM Community Data," *International Journal of Innovation, Management and Technology*, vol. 8, no. 2, pp. 90, 2017.

[6]   M. A. A. Cox and T. F. Cox, "Multidimensional Scaling," *Handbook of Data Visualization*, pp. 315–347. Berlin, Heidelberg: Springer, Berlin Heidelberg, 2008.

[7]   P. Braun, A. Cuzzocrea, T. Keding, C. Leung, A. Padzor, and D. Sayson, "Game Data Mining: Clustering and Visualization of Online Game Data in Cyber-Physical Worlds," in *proceedings of International Conference on Knowledge Based and Intelligent Information and Engineering Systems* KES '17, (Marseille, France), vol. 112, no. 2, pp. 2259–2268, 2017.

[8]   A. Drachen, C. Thurau, J. Togelius, G. Yannakakis, and C. Bauckhage, "Game Data Mining," *Game Analytics: Maximizing the Value of Player Data*, (London, UK), pp. 205–253, Springer London, 2013.

[9]   J. Huan, W. Wang, J. Prins, and J. Yang, "SPIN: Mining Maximal Frequent Subgraphs from Graph Databases," in *Proceedings of the 10th. ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD `04, (Seattle, WA, USA), pp. 581–586, ACM, 2004.

[10]   X. Yan and J. Han, "CloseGraph: Mining Closed Frequent Graph Patterns," in *Proceedings of the 9th. ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '03, (Washington DC., USA), pp. 286–295, ACM, 2003.

[11]   J. Huan, W. Wang, and J. Prins, "Efficient Mining of Frequent Subgraphs in the Presence of Isomorphism," in *Proceedings of the 3rd. IEEE International Conference on Data Mining*, ICDM '03, pp. 549–552, IEEE, 2003.

[12]   M. Kuramochi and G. Karypis, "Finding Frequent Patterns in a Large Sparse Graph," in *Proceedings of the 2004 SIAM International Conference on Data Mining*, pp. 345–356, SIAM, 2004.

[13]   Z. Shao, Y. Hirayama, Y. Yamanishi, and H. Saigo, "Mining Discriminative Patterns from Graph Data with Multiple Labels and Its Application to Quantitative Structure-Activity (QSAR) Models," *Journal of Chemical Information Models*, vol. 55, no. 12, pp. 2519–2527, 2015.

[14]   N. Jin, C. Young, and W. Wang, "Discriminative Subgraph Mining for Protein Classification," in *Computational Knowledge Discovery for Bioinformatics Research*, pp. 279–295, 2012.

[15]   H. Cheng, D. Lo, Y. Zhou, X. Wang, and X. Yan, "Identifying Bug Signatures Using Discriminative Graph Mining," in *Proceedings of the 18th. International Symposium on Software Testing and Analysis* ISSTA '09, (Chicago, IL USA), pp. 141–151, ACM, 2009.

[16]   J. Leopold, N. Eloe, and P. Taylor, "BugHint: A Visual Debugger Based on Graph Mining," in *Proceedings of the 24th International Conference on Visualization and Visual Languages* ICVVL '18, (San Francisco, CA, USA), pp. 109–118, 2018.

[17]   J. Leopold, N. Eloe, J. Gould, and E. Willard, "A Visual Debugging Aid Based on Discriminative Graph Mining," in *Journal of Visual Languages and Computing*, to appear February 2019.

[18]   N. Jin, and W. Wang, "LTS: Discriminative Subgraph Mining by Learning from Search History," in *Proceedings of IEEE 27th. International Conference on Data Engineering* ICDE '11, (Hannover, Germany), pp. 207–218, 2011.

[19]   X. Yan, H. Cheng, J. Han, and P. Yu, "Mining Significant Graph Patterns by Leap Search," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* SIGMOD '08, (Vancouver, BC, Canada), pp. 433–444, ACM, 2008.

[20]   W.-T. Chu and M.-H. Tsai, "Visual Pattern Discovery for Architecture Image Classi_cation and Product Image Search," in *Proceedings of the 2nd. ACM International Conference on Multimedia Retrieval*, ICMR '12, (Hong Kong, China), pp. 1–27, ACM, 2012.

[21]   X. Yan, P. S. Yu, and J. Han, "Graph Indexing: A Frequent Structure-based Approach," in *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, (Paris, France), pp. 335–346, ACM, 2004.

[22] X. Yan and J. Han, "gSpan: Graph-based Substructure Pattern Mining," in *Proceedings of the 2002 IEEE International Conference on Data Mining*, ICDM '02, (Maebashi City, Japan), pp. 721–724, IEEE, 2002.

[23] M. Elseidy, E. Abdelhamid, S. Skiadopoulos, and P. Kalnis, "GraMi: Frequent Subgraph and Ppattern Mining in a Single Large Graph," Proc. *VLDB Endowment*, vol. 7, no. 7, pp. 517–528, 2014.

[24] "Interloper Game Description." http://interlopergame.com/. Accessed: 2018-18-12.

[25] "StarCraft II Game Description." https://starcraft2.com/en-us/. Accessed: 2019-05-09.

IV. *GraPH*: GRAPH PARTITIONING BASED ON HOTSPOTS

Isam A. Alobaidi[1], Jennifer L. Leopold[1], and Andrea E. Smith[1]

[1]Department of Computer Science, Missouri University of Science and Technology, Rolla, MO 65409

ABSTRACT

Graphs have long been used to model relationships between entities. For some applications, a single graph is sufficient; for other problems, a collection of graphs may be more appropriate to represent the underlying data. Many contemporary problem domains, for which graphs are an ideal data model, contain an enormous amount of data (e.g., social networks). Hence, researchers frequently employ parallelized or distributed processing. But first the graph data must be partitioned and assigned to the multiple processors in such a way that the work load will be balanced, and inter-processor communication will be minimized. The latter problem may be complicated by the existence of edges between vertices in a graph that have been assigned to different processors. Herein we introduce a strategy that combines vocabulary-based summarization of graphs (VoG) and detection of hotspots (i.e., vertices of high degree) to determine how a single undirected graph should be partitioned to optimize multi-processor load balancing and minimize the number of edges that exist between the partitioned subgraphs. We benchmark our method against another well-known partitioning algorithm ($METIS$) to demonstrate the benefits of our approach.

1. INTRODUCTION

Graphs have long been used to model relationships between entities. For some applications, a single graph is sufficient; for other problems, a collection of graphs may be more appropriate to represent the underlying data. Some of these graphs may contain an enormous amount of data (e.g., social networks). Hence, parallelized or distributed processing often is employed. Before the analysis commences, typically the graph dataset is partitioned, and a subset of data is assigned to each processor. The partitioning should be done in such a way that the ensuing work load will be balanced and inter-processor communication will be minimized. These tasks can be particularly challenging for a single graph; consideration must be given to which vertices are assigned to which partitions (i.e., processors) and what edges originally existed between those vertices.

Ideally, partitions should be of approximately equal size, and the number of edges between vertices that are in different partitions should be minimized. The problem of finding good partitions in these respects has been studied in graph theory. Despite the numerous algorithms that have been proposed and implemented, the complexity of this problem is still considered *NP*-complete.

In general, most graph partitioning algorithms utilize either edge-cut partitioning or vertex-cut partitioning. Edge-cut partitioning splits the vertices of a graph into disjoint sets of approximately equal size considering the minimum number of cut-edges (e.g., PowerGraph [3], Spark GraphX [4], and Chaos [13]). In contrast, vertex-cut partitioning splits the edges of a graph into equal-sized sets. In this approach, the partitioning of a single graph must satisfy two requirements: the quality graph partitioning criterion (which

guarantees no lost data) and load balancing. Many studies have shown that edge-cut partitioning produces more accurate results on large real-world graphs [3, 4].

Herein we introduce a novel vertex-cut partitioning strategy that determines how a single, undirected graph should be partitioned to optimize multi-processor load balancing and minimize the number of edges that exist between the partitioned subgraphs. Our approach, $GraPH$, first uses vocabulary-based summarization [9] to identify the most highly connected structures that exist in the graph (e.g., cliques, stars, and chains). We then find the vertices in those structures that have the highest degree; these are called hotspots. The hotspots become the starting points from which subgraph partitions are formed.

This paper is organized as follows. In Section 2 we briefly discuss some of the related work in graph partitioning. We present the $GraPH$ algorithm in Section 3, and include a discussion of the VoG summarization algorithm. In Section 4 we benchmark our method against another well-known partitioning algorithm ($METIS$) to demonstrate the benefits of our approach. Concluding remarks and a discussion of future work are provided in Section 5.

2. RELATED WORK

In this section, we briefly review some of the research that has been done in graph partitioning. Despite the numerous sequential, distributed, and parallel algorithms that have been developed, the complexity of this problem is still considered to be *NP*-complete. One of the most significant challenges of the problem continues to be minimizing the loss of information (from the original graph dataset) when the partitions are formed; that is, the goal is to minimize the number of edges (from the original graph) that exists between

vertices that are in different partitions, a situation which is more likely to occur as the number of partitions increases.

Some heuristic methods for sequential graph partitioning of a single graph are discussed in [2, 6]. One offline method (wherein the entire graph is resident in memory), *METIS*, is proposed in [6]. This method produces high-quality partitions in terms of uniformity of partition size and minimization of "lost" edges. However, because of the offline setting, it cannot handle large graphs. The *METIS* algorithm consists of three phases: coarsening, partitioning, and refinement. During each phase, a sequence of specialized algorithms is applied. These algorithms help in selecting the maximal matchings in the coarsening phase, partitioning of the coarse graph in the partitioning phase, and projecting the graph back to the original graph in the refinement phase. An extension to *METIS* (Streaming *METIS* Partitioning method (*SMP*)) is proposed in [2], replacing the offline setting of *METIS* by an online setting. *SMP* provides the ability to adjust the memory capacity, and subsequently decrease computational requirements by applying the partitioning method to small subgraphs.

Some graph partitioning techniques are designed for specific application problems. Another technique for local (i.e., memory-resident, sequential processing) graph partitioning [1] specifically targets fixed cardinality problems such as k-densest subgraph and max k-vertex cover. The authors developed a fixed parameter algorithm using a greediness-for-parameterization technique. Clustering systems are used as a base in [16]. In this research, the authors propose a heuristic graph edge partitioning strategy, Neighbor Expansion (NE), with polynomial running time. Their goal was to reduce the running time

and communication cost for some specific applications such as triangle counting and PageRank.

The graph partitioning problem in a distributed environment is addressed in [7, 8, 11, 12, 14]. The authors in [12] propose a fully distributed algorithm called JA-BE-JA. This algorithm is built on two types of partitioning: vertex-cut and edge-cut partitioning; the absence of central coordination and the processing of each vertex independently make this algorithm well-designed for distributed processing. Another distributed algorithm, *PACC* (Partition-Aware Connected Components), based on graph partitioning for edge-filtering and load-balancing, is proposed in [11]. The authors of [14] propose a multi-level label propagation (*MLP*) method that uses distributed memory of several machines for partitioning the graphs. Another distributed partitioning algorithm is discussed in [10], PARallel Submodular Approximation algorithm (*Parsa*), also configures the partitions to fit the storage and computation ability of each machine.

One important characteristic of graph partitioning algorithms is the strategy employed for selecting the vertex around which the subgraph will be built for each partition. Many algorithms select such vertices randomly. Our approach was motivated by *MELT* [15], MapReduce-based Efficient Large-scale Trajectory anonymization. The main objective of that work was to examine paths traveled by people in a geographical space, and then partition the space into regions around popular locations (e.g., a coffee house, an exercise center, etc.); those locations are referred to as hotspots. As will be discussed later in this paper, the utilization of hotspots as a basis for forming partitions is a novel feature of our partitioning strategy.

3. METHODOLOGY

In this section, we present the $GraPH$ strategy for partitioning a single, undirected graph. We begin with some preliminary definitions that will facilitate this discussion. An explanation of the vocabulary-based summarization of graphs (VoG) technique developed in [9] then follows; this is a key component for our approach as it is used to determine subgraphs of high connectivity (e.g., cliques, stars, and chains). Finally, our complete set of algorithms is presented, detailing how the vocabulary-based summarization and identification of hotspots lead to the creation of optimal partitioning.

3.1. PRELIMINARIES

Definition 1. Graph: A graph G is a tuple (V, E, L) where V is a finite set of nodes called the vertex set of G, and E is a set of 2-element subsets of $V (E \subseteq V \times V)$ called the edge set of G. The nodes and edges are labeled by the function L.

Definition 2. Graph partitioning: A graph $G = (V, E)$ will be partitioned into k subgraphs $G'_{sub} = (V', E')$, $sub = 1, \dots, k$. Each $V'_{subset} \subset V_{set}$ where $V_i \cap V_j = 0$ for $i \neq j$, and each $E'_{subset} \subset E_{set}$.

Definition 3. Full-clique: Let $G = (V, E)$ be an undirected graph. A set FC of vertices in G is called a Full-clique if any two distinct vertices in FC are adjacent in G, when $k \geq 1$. The Full-clique term may refer to the subgraph in some cases. If several edges are missing, this will be defined as a **Near-clique**.

Definition 4. Full bipartite core: Let $G = (V, E)$ be an undirected graph. A set Fb of vertices in G is called Full-bipartite if two sets of vertices S_1 and S_2, $S_1 \cap S_2 = \phi$, have

edges between them, where each vertex in S_1 will be connected to every edge in S_2 but not within the same set. When the core is not fully connected this will be defined as a **Near-bipartite core**.

Definition 5. Star: A Star consists of one internal vertex in set S_1 connected to k edges of other sets S_{i+1} (spokes). A Star is considered as a special case of a Full bipartite core.

Definition 6. Chain: A Chain is a sequence of vertices such that all vertices have degree 2, except two of them have degree 1.

Figure 1 shows examples of these structure types.

(a) Full-Clique   (b) Near-Clique   (c) Full-bipartite
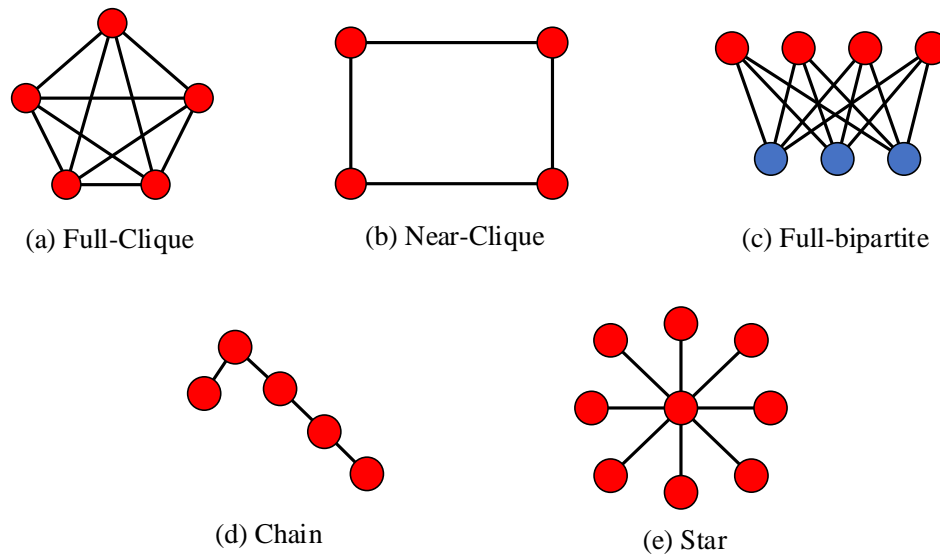
(d) Chain   (e) Star

Figure 1: Types of Structures

3.2. VOG GRAPH SUMMARIZATION

The ability to summarize information about highly connected subgraphs contained within a large graph can greatly facilitate understanding of the graph as a whole. Vocabulary-based summarization of Graphs (VoG) [9] is a formal methodology developed

for this purpose. Using a set of terms (i.e., a vocabulary) like full-cliques, near-cliques, full-bipartite core, near-bipartite core, stars, and chains, VoG provides a summary of the most highly connected and frequently occurring structures in a graph. For problem domains like social networks and communication networks, these are typically the structures of most interest.

Algorithm 1 VoG

Input Graph G
Output Graph summary M, encoding cost.
   1: **Subgraph Generation**. Using graph decomposition methods, produce a set of candidate subgraphs, which may overlap with each other.
   2: **Subgraph Labeling**. Characterize each subgraph as one of the vocabulary structure types.
   3: **Summary Assembly**. From the candidate structures, select a non-redundant subset to instantiate the graph model M. Utilizing a heuristic model (e.g., PLAIN, TOP10, TOP100, GREEDY'nFORGET), the set of structures with the lowest description cost will be selected.

Algorithm 1 outlines the main steps that are performed in VoG; see [9] for a more detailed discussion. Using graph decomposition methods, candidate subgraphs are first generated. They are then classified as various connected structures such as cliques, stars, and chains; if a subgraph qualifies as more than one of these structure types, a scoring method (based on minimum description length (*MDL*)) is used to determine which structure type that subgraph best fits. VoG then uses another scoring system to determine which collection of those structures best characterizes the graph as a whole. This is called the summary model, and could include all of the structures (PLAIN), just the k structures

with the best scores (TOP10, TOP100), or a combination of structures whose total score

is best (GREEDY'nFORGET).

3.3. PROPOSED ALGORITHM

In $GraPH$, we first use VoG to identify the most highly connected, and frequently

occurring, subgraphs. That produces a set of structures (i.e., the model summary), S.

Algorithm 2 is then used to select a subset of S which we call the majority structures, $MajS$.

The number of majority structures depends on the desired number of partitions, n. The n

structures in S that have the largest number of vertices become the majority structures.

Algorithm 2 Select the Majority Structures

Input S is set of structures produced by VoG,

      n is number of desired partitions

Output $MajS$ contains n structures in S that have the largest number of vertices

  1: $SortedS$ = Sort structures in S in descending order by number of vertices

  2:   **for** i = 1 to n **do**

  3:      $MajS[i] = SortedS[i]$

  4:   **end-for**

  5: **return** $MajS$

For each majority structure, Algorithm 3 is applied to identify the vertex that has

the highest degree; in the case of a tie, an arbitrary choice between those qualifying vertices

is made. These vertices of highest degree are called hotspots.

After assigning the hotspots, the actual partitioning commences. The subgraph that

will be assigned to a partition will consist of all the vertices in a hotspot's structure unless

that number of vertices exceeds the total number of vertices in the graph divided by the

number of desired partitions; that is considered the ideal partition size. In Algorithm 4, we

start a depth-first search from a hotspot vertex (denoted as $Hotspot$). The $MajS$ denoted in the algorithm is the set of structures from which the hotspot was selected.

Algorithm 3 Assign the HotSpot

Input $S = (V_S, E_S)$ is a structure

Output $HotSpots$ is a vertex in V_S that is the hotspot vertex for structure $S = (V_S, E_S)$

1:    **for** $i = 1$ to $|V_S|$ **do**

2:       $degree[i] = 0$

3:    **end-for**

4:    **for** $i = 1$ to $|V_S|$ **do**

5:     **for** $j = 1$ to $|V_S|$ **do**

6:       **if** there is an $edge(i, j)$ in E_S **then**

7:        $degree[i] = degree[i] + 1$

8:       **end-if**

9:     **end-for**

10:   **end-for**

11: $HotSpot = 1$

12:   **for** $i = 2$ to $|V_S|$ **do**

13:    **if** $degree[HotSpot] \le degree[i]$ **then**

14:     $HotSpot = i$

15:    **end-if**

16:   **end-for**

17: **return** $HotSpot$

There are two discontinuation criteria for building a subgraph partition; the expansion will stop when either of those conditions is satisfied:

1. The current size of a partition subgraph has reached the ideal partition size.

2. The path length from the current vertex to the hotspot has reached a maximum threshold (i.e., the total number of desired partitions).

Some vertices from the original graph may not be included in any partition using these conditions. To handle those cases, we perform a breadth-first search starting from each hotspot until all nodes are included in some partition.

Algorithm 4 Graph Partitioning

Input Graph $G = (V, E)$, $HotSpot$, and $MajS$

      $HotSpot$ is a vertex in the structure connected to the largest number of edges

      $MajS$ is a set containing structures that have the largest number of vertices

      n is the number of partitions

Output All $SubGraph$ of G, where $|V|$ of each subgraph $\geq PartitionSize$

  1: $PartitionSize = |V| / n$

  2:   **if** $|MajS_i| \leq PartitionSize$ **then**

  3:     Include all nodes of $MajS_i$ in $Partition_i$

  4:   **end-if**

  5:   Perform DFS starting from each $HotSpot$

  6:     $SubGraph_{DFS} \leftarrow DFS$

  7:   Perform BFS starting from each $HotSpot$

  8:     $SubGraph_{BFS} \leftarrow BFS$

  9: $SubGraph \leftarrow SubGraph_{DFS} \cup SubGraph_{BFS}$

10: **return** $SubGraph$

3.4. COMPUTATIONAL COMPLEXITY

The complexity of one well-known partitioning method that is considered to produce high-quality partitions, $METIS$ [6] (implemented as kmetis), is approximately $O(V + E + k \log k)$ where V is the number of nodes, E the number of edges, and k is the number of partitions [5]. In contrast, the complexity of $GraPH$ is approximately $O(V + E + n \log n)$ where V is the number of nodes, E is the number of edges, and n is the number of structures. Contributing to the overall complexity of $GraPH$ is the complexity of BFS

and DFS, which are $O(V + E)$, and the complexity of sorting n structures, which is $O(n \log n)$. We are not including the complexity of the VoG processing, which has not been published by its authors.

4. RESULTS AND ANALYSIS

In this section we compare the results of partitioning three datasets using $GraPH$ and another well-known partitioning method, $METIS$, which was discussed in Section 2. The $GraPH$ algorithms presented in Section 3.2 and 3.3 were (collectively) implemented in Matlab and C++. A C++ implementation of $METIS$ was downloaded from the Karypis Lab website [5]. Our experiments were executed on an Intel(R) Core(TM) i7-6700 CPU@3.40GHz computer with 32 GB memory.

4.1. DATA DESCRIPTION

Three single undirected graphs were used to evaluate our approach. Table 1 lists descriptive information about the graphs. One graph was synthetically generated; a second graph represented a two-dimensional finite element mesh; the third graph represented a three-dimensional finite element mesh.

Table 1: Description of the Graphs Tested

Graph Name	Number of Nodes	Number of Edges	Description
Synthetic	1565	3561	Synthetically generated
4ELT	15606	45878	2D Finite element mesh
COPTER2	55476	352238	3D Finite element mesh

4.2. EXPERIMENT AND RESULTS

We executed *GraPH* and *METIS* on each of the graphs listed in Table 1, testing seven different numbers of partitions for each graph. The results from each test were analyzed in terms of three different metrics: the number of *interior edges* per partition (i.e., edges in a partition's graph), the number of *exterior edges* per partition (i.e., edges between vertices in a partition and vertices assigned to other partitions), and the total number of *edges lost* (i.e., edges from the original graph that were not represented in any of the partition graphs).

Seven tests were conducted to create 10, 20, 30, 40, 50, 60, and 70 partitions, respectively, of the Synthetic graph. *METIS* failed to partition this graph into either 20 or 40 partitions; the program simply failed to return any results. *GraPH* produced results for all of the tested numbers of partitions for this graph. The representation of edges amongst partitions was not well distributed when 10 partitions were requested. Specifically, the number of interior edges in one of those partitions was much higher than in the other partitions, which was not an optimal partitioning. This was likely due to the fact that when a hotspot is selected from a structure, if the structure can fit entirely into a partition, all nodes from that structure automatically will be added to the partition before the depth-first search algorithm is run. This can then prevent other partitions from growing during depth-first search (as would be the case in unconnected components), encouraging disproportionate partition sizes. Because the 4ELT and COPTER2 graphs were much larger than the Synthetic graph, we tested larger numbers of partitions for those graphs, namely: 100, 200, 300, 400, 500, 600, and 700.

1565 Nodes - 3561 Edges

	10	20	30	40	50	60	70
GraPH	234.2	57.9	13.1	6.7	3.5	2.4	1.8
METIS	38.4	0	6.1	0	1.84	1.4	1.1

15606 Nodes - 45878 Edges

	100	200	300	400	500	600	700
GraPH	269.1	119.5	73.7	51.7	37.6	29	23.9
METIS	185.9	63.1	33.8	21.4	13.6	9.8	5.6

55476 Nodes - 352238 Edges

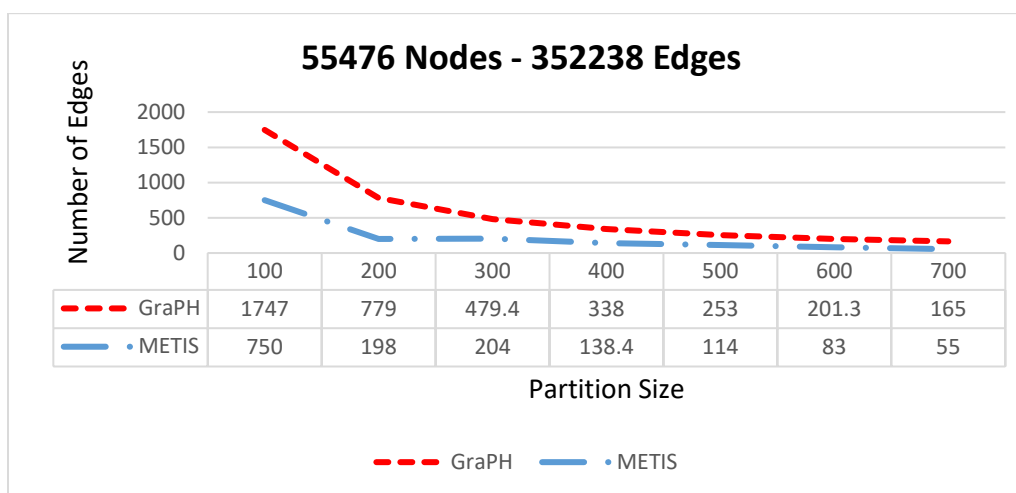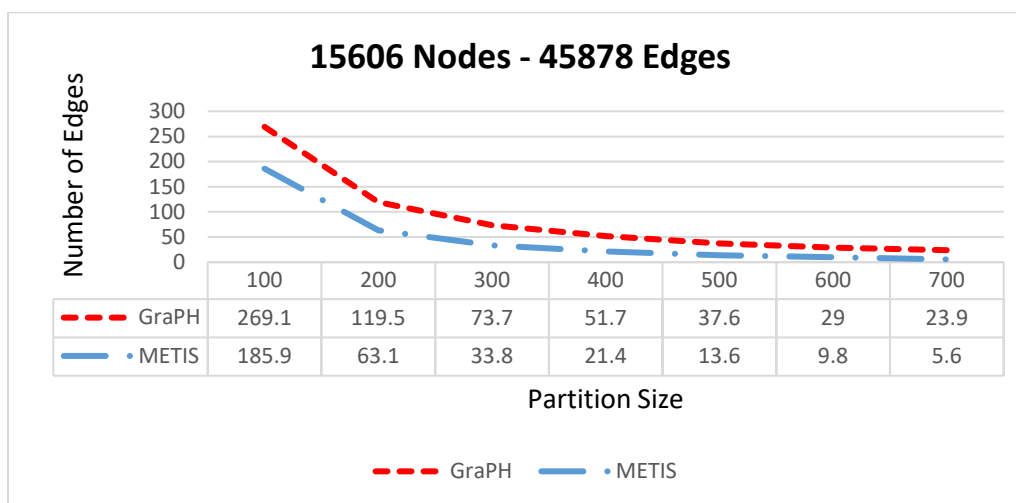|	100	200	300	400	500	600	700
GraPH	1747	779	479.4	338	253	201.3	165
METIS	750	198	204	138.4	114	83	55
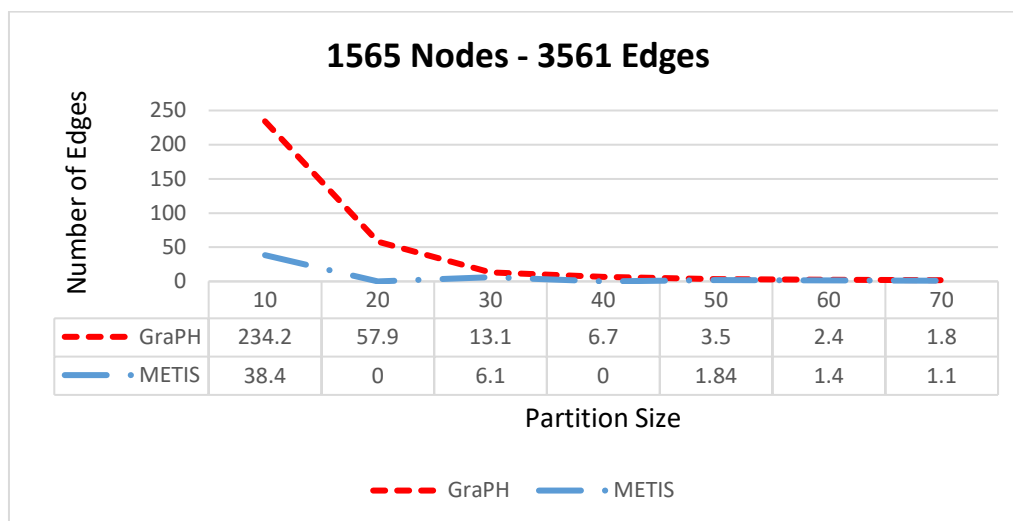
Figure 2: Interior Edges per Partition

1565 Nodes - 3561 Edges

	10	20	30	40	50	60	70
GraPH	197.5	217.2	195.8	153	126.3	106.2	91.6
METIS	635.4	0	225.2	0	138.76	115.8	99.6

15606 Nodes - 45878 Edges

	100	200	300	400	500	600	700
GraPH	1246.3	653.2	447.4	342.6	281.7	239	207
METIS	1412.6	765.6	527.3	403.3	329.6	277.7	243.6

55476 Nodes - 352238 Edges

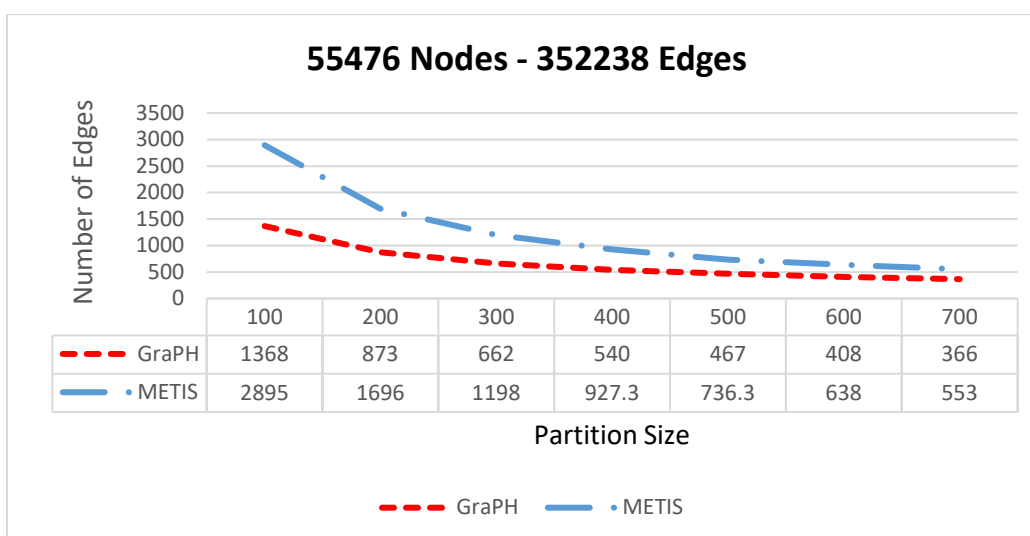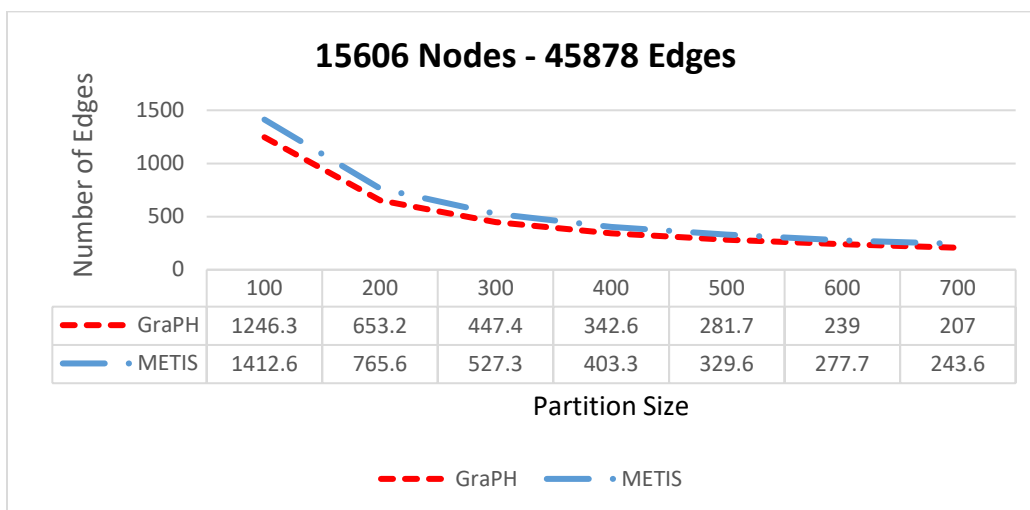|	100	200	300	400	500	600	700
GraPH	1368	873	662	540	467	408	366
METIS	2895	1696	1198	927.3	736.3	638	553
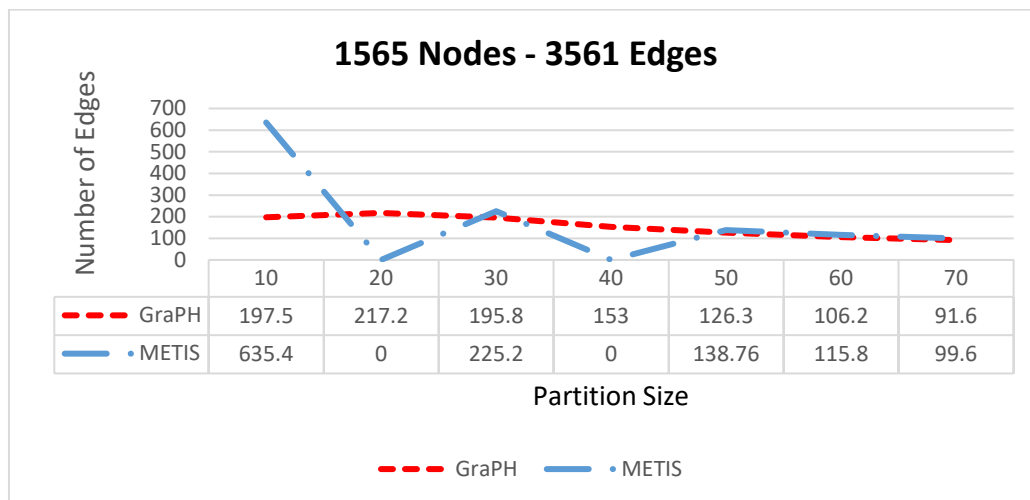
Figure 3: Exterior Edges per Partition

1565 Nodes - 3561 Edges

	10	20	30	40	50	60	70
GraPH	987	2171	2937	3060	3156	3183	3206
METIS	3177	0	3378	0	3469	3475	3487

Partition Size

GraPH   METIS

15606 Nodes - 45878 Edges

	100	200	300	400	500	600	700
GraPH	62312	65314	67104	68519	70422	71807	72498
METIS	70630	76561	79087	80659	82407	83322	85268

Partition Size

GraPH   METIS

55476 Nodes - 352238 Edges

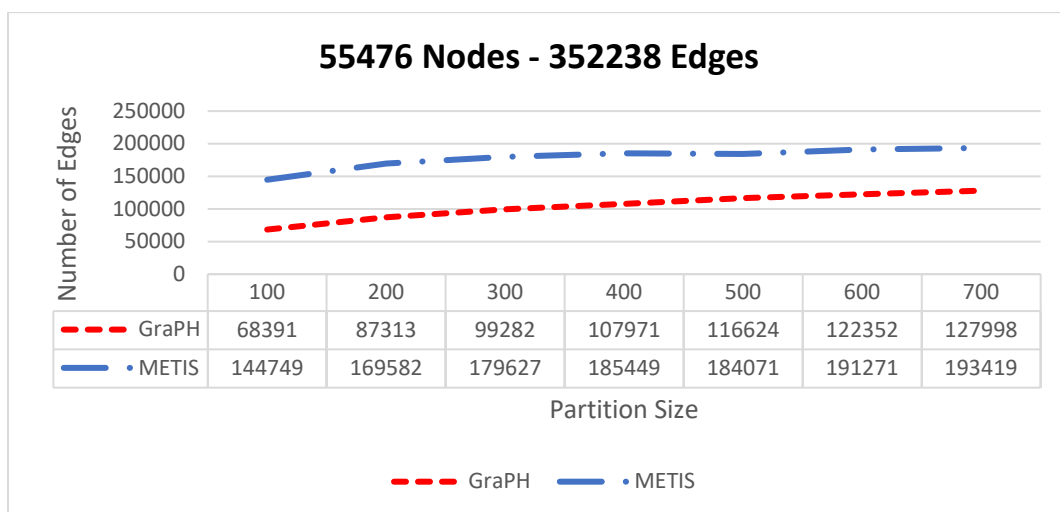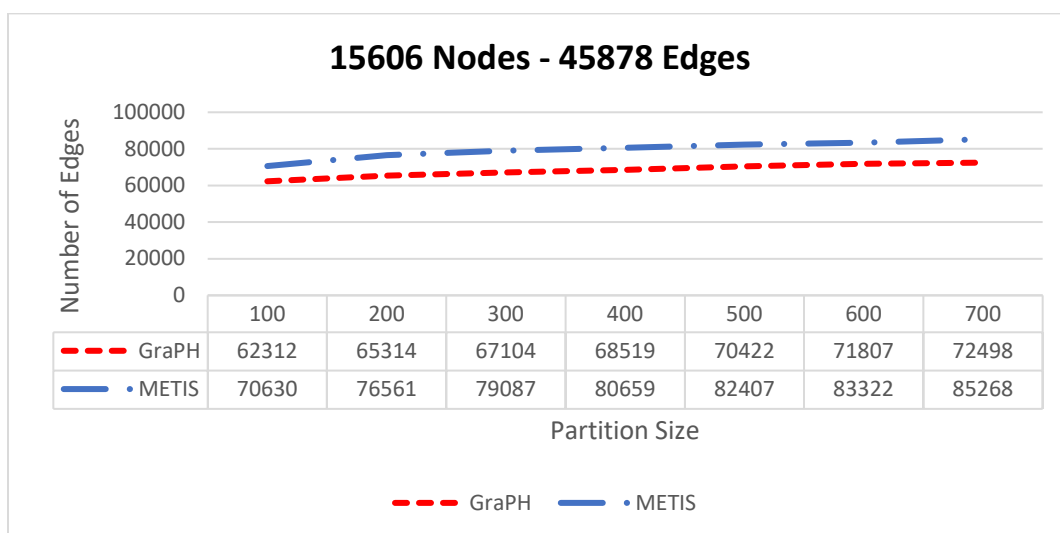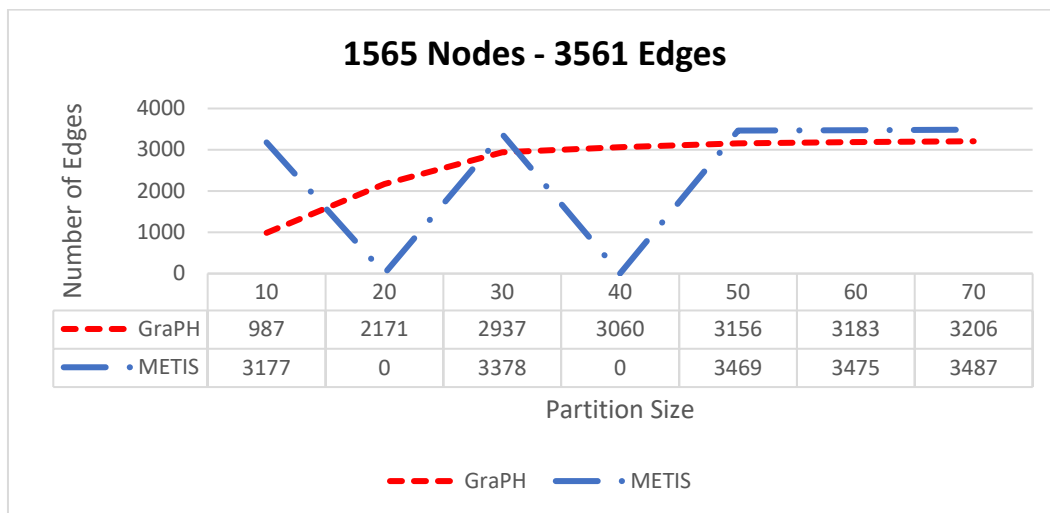|	100	200	300	400	500	600	700
GraPH	68391	87313	99282	107971	116624	122352	127998
METIS	144749	169582	179627	185449	184071	191271	193419

Partition Size

GraPH   METIS

Figure 4: Total Edges Lost

For all three of the graphs listed in Table 1, in the majority of the tests, the partitions produced by $GraPH$ had a higher number of interior edges in each partition than the partitions produced by $METIS$. It can be seen in Figure 2 that more edges from the original graph were retained within the partitions produced by $GraPH$. As shown in Figure 3, the $GraPH$ partitioning resulted in fewer exterior edges (between partitions) than what occurred in the $METIS$ partitioning. Additionally, as shown in Figure 4, $GraPH$ outperformed $METIS$ in terms of reducing the total number of edges lost from the original graph. It should be noted that as the desired number of partitions grew, the difference in partition quality (in terms of the three metrics) obtained from both methods became less distinct.

Because of the use of two methods (depth-first/breadth-first search) in $GraPH$ for the extension process that include vertices in/out of partition boundaries, we also evaluated different variations of our method. We ran $GraPH$ on the three test graphs using four different orders of processing:

1. Depth-first search extension for vertices inside the partition boundaries followed by breadth-first search extension for vertices outside the partition boundaries.

2. Breadth-first search extension for vertices inside the partition boundaries followed by depth-first search extension for vertices outside the partition boundaries.

3. Depth-first search extension for vertices inside the partition boundaries followed by depth-first search extension for vertices outside the partition boundaries.

4. Breadth-first search extension for vertices inside the partition boundaries followed by breadth-first search extension for vertices outside the partition boundaries.

We found that more consistent partitions were obtained (in terms of more interior edges and fewer external edges per partition) when we utilized the depth-first search extension process for vertices inside the boundaries followed by breadth-first search extension processing for vertices outside the boundaries. We also tested random assignment of hotspots. This was found to be unreliable in generating high-quality partitions. Interestingly, although the number of internal edges was not balanced across partitions utilizing randomization, $GraPH$ still outperformed $METIS$ in terms of producing partitions with more internal edges and fewer external edges.

5. CONCLUSION AND FUTURE WORK

With the proliferation of data in our technological world and the usefulness of modeling some problems using graphs, it is becoming increasingly difficult to process an entire graph dataset in memory. It is more efficient to partition a single large graph, and process multiple smaller subgraphs. However, in doing so, the partitioning of what may be highly interconnected data must be done in such as way as to balance the work load amongst the individual processes, minimize inter-process communication, and minimize loss of information from the original dataset. The latter problems can occur if, in the original graph, there is an edge that exists between vertices assigned to different partitions.

Herein we have presented an algorithm, $GraPH$, for partitioning a single, undirected graph. Our algorithm strives to produce quality partitions in terms of: uniformity of the size of each partition, maximization of the number of edges from the original graph that are included in each partition, and minimization of the number of edges from the

original graph that effectively exist between partitions. Our approach is novel; we first utilize vocabulary-based summarization (VoG) to find the most highly connected structures, and then find the vertices of highest degree (known as hotspots) within those structures. A benchmark comparison of $GraPH$ with another well-known, high-quality partitioning algorithm ($METIS$) demonstrated the benefits of our strategy.

In the future, we plan to explore ways to distribute or parallelize the $GraPH$ algorithms so that we can process even larger graphs than those tested for this study. To that end, we also may explore the use of some approximation (e.g., sampling) methods that may increase the efficiency of the assignment of vertices to partitions after identification of structures and hotspots.

REFERENCES

[1] É. Bonnet, B. Escoffier, V. Th. Paschos, and É. Tourniaire, "Multi-parameter Analysis for Local Graph Partitioning Problems: Using Greediness for Parameterization," in *Algorithmica*, (New York, NY, USA), vol. 71, no. 3, pp. 566–580, Springer, 2015.

[2] G. Echbarthi and H. Kheddouci, "Streaming METIS Partitioning," in *Proceedings of the 2016 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*, ASONAM '16, (Piscataway, NJ, USA), no. 8, pp.17–24, IEEE, 2016.

[3] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed Graph-parallel Computation on Natural Graphs," in *Proceedings of the 10$^{th.}$ USENIX Symposium on Operating Systems Design and Implementation*, OSDI '12, (Hollywood, CA, USA), no. 14, pp.17–30, USENIX Association, 2012.

[4] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "GraphX: Graph Processing in a Distributed Dataflow Framework" in *Proceedings of the 11$^{th.}$ USENIX Symposium on Operating Systems Design and Implementation,* OSDI '14, (Broomfield, CO, USA), no. 16, pp. 599–613, USENIX Association, 2014.

[5] G. Karypis, "Complexity of pmetis and kmetis Algorithms." http: //glaros.dtc.umn.edu/gkhome/node/419. Accessed: 2019-22-01.

[6] G. Karypis and V. Kumar, "A Fast and High-Quality Multilevel Scheme for Partitioning Irregular Graphs," *SIAM Journal on Scientific Computing*, (Philadelphia, PA, USA), vol. 20, no. 1, pp. 359–392, SIAM, 1998.

[7] G. Karypis and V. Kumar, "A Parallel Algorithm for Multilevel Graph Partitioning and Sparse Matrix Ordering," in *Journal of Parallel and Distributed Computing*, vol. 48, no. 1, pp. 71–95, ELSEVIER, 1998.

[8] R. Kiveris, S. Lattanzi, V. Mirrokni, V. Rastogi, and S. Vassilvitskii, "Connected Components in MapReduce and Beyond", in *Proceedings of the ACM Symposium on Cloud Computing* SOCC '14, (New York, NY, USA), no. 13, pp. 1–13, ACM, 2014.

[9] D. Koutra, U. Kang, J. Vreeken, and C. Faloutsos, "Summarizing and Understanding Large Graphs," in *Statistical Analysis and Data Mining: The ASA Data Science Journal*, Wiley Periodicals, Inc., vol. 8, no. 3, pp. 183–202, 2015.

[10] M. Li, D. G. Andersen, and A. J. Smola, "Graph Partitioning via Parallel Submodular Approximation to Accelerate Distributed Machine Learning," *CoRR*, 2015.

[11] H.-M. Park, N. Park, S.-H. Myaeng, and U. Kang, "Partition Aware Connected Component Computation in Distributed Systems," in *Proceedings of the 16th. IEEE International Conference on Data Mining* ICDM '16, pp. 420–429, IEEE, 2016.

[12] F. Rahimian, A. H. Payberah, S. Girdzijauskas, M. Jelasity, and S. Haridi, "A Distributed Algorithm for Large-Scale Graph Partitioning," *ACM Trans. Auton. Adapt. Syst,* (New York, NY, USA), vol. 10, no. 2, pp. 1–12, ACM, 2015.

[13] A. Roy, L. Bindschaedler, J. Malicevic, and W. Zwaenepoel, "Chaos: Scale-out Graph Processing from Secondary Storage," in *Proceedings of the 25th. Symposium on Operating Systems Principles* SOSP '15, (New York, NY, USA), pp. 410–424, ACM, 2015.

[14] L. Wang, Y. Xiao, B. Shao, and H. Wang, "How to Partition a Billion-Node Graph," in *Proceedings of the 30th. IEEE International Conference on Data Engineering* ICDE '14, pp. 568–579, IEEE, 2014.

[15] K. Ward, D. Lin, and S. Madria, "MELT: Mapreduce-based Efficient Large-scale Trajectory Anonymization," in *Proceedings of the 29th. International Conference on Scientific and Statistical Database Management* SSDBM '17, (New York, NY, USA), pp. 1–35, ACM, 2017.

[16] C. Zhang, F. Wei, Q. Liu, Z. G. Tang, and Z. Li, "Graph Edge Partitioning via Neighborhood Heuristic," in *Proceedings of the 23rd. ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* KDD '17, (New York, NY, USA), pp. 605–614, ACM, 2017.

SECTION

2. CONCLUSIONS AND FUTURE WORK

2.1. CONCLUSIONS

This dissertation presents algorithms and methods to mine and analyze transaction graphs. Specifically, we have applied graph data mining techniques, frequent (FSM) and discriminative subgraph mining (DSM), to Real-Time Strategy (RTS) video game datasets to develop a system that can provide recommendations in order to improve one's chances of winning the game.

In paper I, frequent subgraph mining, has been applied to a strategy game dataset to develop a system that can provide recommendations about moves that a player should and should not make in order to improve his/her chances of winning the game. As proof of concept, we applied our system to a real-time strategy (RTS) game dataset during each of three phases of the game and achieved fairly accurate results when we tested using cross-fold validation. We also attempted to apply another technique, frequent sequence mining, but did not find that it provided as useful or accurate recommendations.

In paper II, we tested another graph mining technique, discriminative subgraph mining, on the same RTS game dataset. The predictions for what sequences of moves a player should and should not make in order to increase his/her chances of winning a game were found to be extremely accurate for each of three phases of the game when tested using cross-fold validation.

In paper III, a comparison between the two previously mentioned graph data mining techniques, frequent and discriminative subgraph mining, were compared and tested on a much larger dataset of played multi-player, Real-Time Strategy (RTS) video games. The earlier results were reinforced, with discriminative subgraph mining producing the more accurate recommendations than frequent subgraph mining, and those recommendations being highly accurate for all three phases of the game.

In paper IV, we proposed an algorithm, *GraPH*, for partitioning a single, undirected graph. Our algorithm strives to produce quality partitions in terms of: uniformity of the size of each partition, maximization of the number of edges from the original graph that are included in each partition, and minimization of the number of edges from the original graph that effectively exist between partitions. Our approach is novel; we first utilize vocabulary-based summarization (VoG) to find the most highly connected structures, and then find the vertices of highest degree (known as hotspots) within those structures. An implementation of the algorithm was benchmarked against a well-known partitioning algorithm (METIS) and was found to be superior in the aforementioned metrics for quality partitioning for most all test cases.

2.2. FUTURE WORK

As a part of the future research, we intend to extend the scope of the graph mining predictive analytics using real-life datasets other than game data (e.g., healthcare data). Although discriminative subgraph mining produced the best results for the game datasets, we will still compare both graph mining methods for the other types of datasets in case one method proves to be more useful than the other for different types of data.

In order to reduce the search space and speed up the computation process, we intend to work on heuristic algorithms for both FSM and DSM. It is possible, for example, that we could combine particular actions into categories based on semantic similarity (e.g., "create fort" and "create castle" into "create protective unit") inorder to more quickly find common subgraphs.

BIBLIOGRAPHY

[1] J. Yang, W. Su, S. Li, and M. M. Dalkilic, "Wigm: discovery of subgraph patterns in a large weighted graph," in *Proceedings of the 2012 SIAM International Conference on Data Mining*, pp. 1083–1094, SIAM, 2012.

[2] N. Jin, and W. Wang, "LTS: Discriminative Subgraph Mining by Learning from Search History," in *Proceedings of IEEE 27th. International Conference on Data Engineering* ICDE '11, (Hannover, Germany), pp. 207–218, 2011.

[3] M. A. Bhuiyan and M. Al Hasan, "An iterative mapreduce based frequent subgraph mining algorithm," in *Proceedings of IEEE Transactions on Knowledge and Data Engineering*, vol. 27, no. 3, pp. 608–620, IEEE, 2015.

[4] W. Lin, X. Xiao, and G. Ghinita, "Large-scale frequent subgraph mining in mapreduce," in *Proceedings of IEEE 30th. International Conference on Data Engineering* ICDE '14, pp. 844–855, IEEE, 2014.

[5] E. Wong, B. Baur, S. Quader, and C.-H. Huang, "Biological network motif detection: principles and practice," in *Proceedings of Briefings in bioinformatics*, vol. 13, no. 2, pp. 202–215, Oxford University Press, 2011.

[6] C. Jiang, F. Coenen, and M. Zito, "A survey of frequent subgraph mining algorithms," in *Proceedings of The Knowledge Engineering Review*, vol. 28, no. 1, pp. 75–105, Cambridge University Press, 2013.

[7] E. Scharwächter, E. Müller, J. Donges, M. Hassani, and T. Seidl, "Detecting change processes in dynamic networks by frequent graph evolution rule mining," in *Proceedings of IEEE 16th. International Conference on Data Mining* ICDM '16, pp. 1191–1196, IEEE, 2016.

[8] D. Koutra, U. Kang, J. Vreeken, and C. Faloutsos, "Summarizing and Understanding Large Graphs," in *Statistical Analysis and Data Mining: The ASA Data Science Journal*, Wiley Periodicals, Inc., vol. 8, no. 3, pp. 183–202, 2015.

[9] H. Cheng, X. Yan, and J. Han, "Mining graph patterns," *in Frequent Pattern Mining*, pp. 307–338, Springer, 2014.

VITA

Isam Abdulmunem Alobaidi was born in Baghdad, Iraq. He earned his bachelor's degree in software engineering from Al-Mansour University College, Iraq in 2001. He subsequently earned his higher diploma degree in software engineering from the Iraqi Commission for Computers and Information - Institute of Higher Studies in Informatics, Iraq in 2002. In December 2015, he received his master's degree in computer science from Missouri University of Science and Technology, USA. He continued his graduate study towards his Ph.D. in 2016 at the same institution.

During his time as a student, he worked as a graduate research assistant and graduate teaching assistant within the Department of Computer Science. He published conference papers as a primary and secondary author, most of which are cited in this research. In December 2019, he received his Ph.D. in computer science from Missouri University of Science and Technology, USA.