2011

Heuristics modeling optimization-oriented element behaviors in discrete, repulsive systems

David Barber

HEURISTICS MODELING OPTIMIZATION-ORIENTED ELEMENT BEHAVIORS IN DISCRETE, REPULSIVE SYSTEMS

by

DAVID BARBER

A THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in
The Department of Computer Science
to
The School of Graduate Studies
of
The University of Alabama in Huntsville

HUNTSVILLE, ALABAMA

2011

In presenting this thesis in partial fulfillment of the requirements for a master's degree from The University of Alabama in Huntsville, I agree that the Library of this University shall make it freely available for inspection. I further agree that permission for extensive copying for scholarly purposes may be granted by my advisor or, in his/her absence, by the Chair of the Department or the Dean of the School of Graduate Studies. It is also understood that due recognition shall be given to me and to The University of Alabama in Huntsville in any scholarly use which may be made of any material in this thesis.

David Barber

10/18/2011
(date)

THESIS APPROVAL FORM

Submitted by David Barber in partial fulfillment of the requirements for the degree of Master of Science in Computer Science in Computer Science and accepted on behalf of the Faculty of the School of Graduate Studies by the thesis committee.

We, the undersigned members of the Graduate Faculty of The University of Alabama in Huntsville, certify that we have advised and/or supervised the candidate of the work described in this thesis. We further certify that we have reviewed the thesis manuscript and approve it in partial fulfillment of the requirements for the degree of Master of Science in Computer Science in Computer Science.

Peter Slater 18 Oct 11 Committee Chair
Dr. Peter Slater (Date)

Mary Ellen Weisskopf 10/18/11
Dr. Mary E. Weisskopf (Date)

10-18-11
Dr. Glenn W. Cox (Date)

Heggere S. Ranganath 10-26-11 Department Chair
Dr. Heggere S. Ranganath (Date)

10/18/11 College Dean
Dr. John D. Fix (Date)

Rhonda Kay Gaede 12/2/11 Graduate Dean
Dr. Rhonda Kay Gaede (Date)

ABSTRACT

School of Graduate Studies
The University of Alabama in Huntsville

Degree Master of Science in Computer Science College/Dept. Science/Computer Science

Name of Candidate David Barber

Title Heuristics Modeling Optimization-Oriented Element Behaviors in Discrete, Repulsive Systems

Finding the maximal minimum distance between elements in a discrete system has many applications. This problem is very closely related to the NP-complete sphere packing problem. Heuristics have been developed to achieve the upper bound for an arbitrary system. Heuristic rules are compared to physical or social behaviors which could be used to simulate similar existing systems. A series of cases is proposed as a test-bed for evaluation of the executed heuristics. The heuristics perform with varied success depending on the space configuration of the system. At least two of the heuristics perform with consistently high bounds on the test-bed cases. The results suggest that assigning uniform behavior to elements in a system can achieve results very near the solution's upper bound.

Abstract Approval: Committee Chair Peter Slater 18 Oct 11
Dr. Peter Slater

Department Chair Heggere S. Ranganath 10-26-11
Dr. Heggere S. Ranganath

Graduate Dean Rhonda Kay Gaede 12-2-11
Dr. Rhonda Kay Gaede

ACKNOWLEDGMENTS

I want to thank my advisor Dr. Peter Slater who was willing to take me as a thesis student even though my knowledge of the theory behind the research was a bit shaky. He paitently answered the same questions time and time again as I was not always clear of my objective or what made research practical instead of strictly an academic exercise. Without his direction, my research would have comprised of several loosely connected ideas which served as some aimless research endeavor. I believe the most important knowledge which Dr. Slater passed on to me was the knowledge and experience of the research process, which at times had me on the peaks of mount Olympus but probably more often at the bottom of the Mariana Trench.

I want to thank my employer, System Dynamics International (SDI), for supporting my education both financially and morally, in addition to providing me meaningful work in a great environment.

I want to thank my wife for her support in every way. She helped keep me focused on the goal at hand but also soberminded enough to realize there are more important things in life.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

TABLE PAGE

This thesis is dedicated to my beautiful wife and children: Seema, Faery, and Fineen

The heavens are Yours; the earth also is Yours; the world and all that is in it, You have founded them.

—Psalm 89:11

CHAPTER 1

INTRODUCTION

He does not like his foods to touch

—Hope Barber

At a local restaurant, a man watches in terror as juices from his stewed vegetables seep into the roll sitting on his plate. As the man begins to cry, he looks across the street to a baseball field where a little league baseball game is just underway. As the children begin chasing butterflies out of boredom, they begin to converge in the center of the field. The ball is smashed to the right field where it drops uninhibited to the ground for an inside-the-park homerun. As the batter rounds the bases, across town, his brother is changing the keyboard layout on his computer. He has convinced himself that the reason he types so slowly is because the keys are not positioned in an optimal manner. Next door, a security specialist is installing a sensor network while the couple owning the house is out giving dancing lessons. Could the man's dinner roll have been saved? Why was it important that the little league outfielders stay in their respective positions? What does your keyboard layout have to do with how fast you can type? How much do the sensor positions of a security network matter? How should the dancing instructors teach their students to act to maintain a maximal distance from one another?

1.1 Spatial Relationships

Spatial relationships pervade everyday life without people even thinking about them. Every scenario above has some connection to problems of spatial relationships. Had the vegetables been farther away, their juices might not have reached the roll. Had the children playing baseball been standing in the proper position, they may have been able to make a play on the baseball. Such notions are common sense to most individuals; nobody stops to ponder, "Why don't all the baseball players stand in one spot?" Naturally occurring spatial relationships always follow behavior rules: place a dinner roll as far from all the liquids on a plate as possible, place the keys most frequently used in the English language in places that fingers can navigate the keyboard deftly. Rules can vary in complexity. Consider our initial little league scenario. It may be first assumed that an equal player spread in the outfield will effectively cover the field. Now, imagine that the coach had access to all of the current meteorological data and data on the batter against the current left-handed pitcher. Perhaps the physics of the environment and the statistics related to the batter's historical hit locations may cause the outfield to pull closer to the infield and towards one side of the field. The point is that even this simple scenario can produce a complex set of rules. This is not limited to social interactions or game theory, but has strong connections to the physical sciences. Quantum physics is one such area of physical science that is very concerned with the spatial relationships of electrons in an atom and how it relates to energy (Cohn, Kumar [5]).

There is a distinction to be made about the previously cited scenarios. A dinner roll does not decide where it should be with regard to other foods on a plate, nor do the keys on a keyboard determine how they should be arranged to help a computer user. The space location of both of these examples is designed to meet a desired attribute of the system. In some cases, there may not be an intelligent agent that knows the optimal space locations and it is the rules governing the elements of the system that determine their location. Consider the movement of electrons in an atom, or of ants in an ant farm. The structure of the whole system is determined by the "decisions" of individual elements; there is no central intelligence. Perhaps this is superficially the difference between science and engineering, or modeling and design.

In addition to what layer of the system the enforcement of these spatial behavior rules is found, one could classify these behavior rules by the resulting distance characteristics. How close do elements tend to be from one another? Perhaps such a characteristic is indicative of some rule of attraction. Or conversely, how far do elements tend to be from one another? This might suggest a rule of repulsion. Depending on the system and even the classification of the elements within the system, there is a great potential for a complex behavioral rule set within any system.

The scope of this research is comprised of two objectives concerning a smaller subset of problems.

1. The primary goal is to find a method placing an arbitrary number of elements in positions where they maintain a maximal distance from their nearest neighbors.

2. The secondary goal is whether the method of distribution induced on the elements of the system of interest produces an accurate model of social or physical element interactions.

As the elements seek to be a maximal distance from their nearest neighbors, this behavior is classified as **repulsive**, and the containing system is referred to as a repulsive system. Revisiting one of the previously cited problems, consider the dancing instructors and their dilemma. What should they teach their students in order to maintain the greatest distance from one another? This question serves to combine the two perspectives of spatial relationships that have previously been foreshadowed; some models are only concerned with the final outcome (where the dancers are located on the dance floor) while others are concerned with individual element behavior (singularly, how does each dancer behave?). Therefore, the focus of this research will be on a modeled dancing problem that serves as a study of both perspectives. The advantage of analyzing this particular model is that the behavior of the elements (dancing couples) is assumed to be repulsive in nature. Many social and physical interactions follow repulsive element behavior; it is expected that this dancing couples model will be applicable to modeling other social and physical systems. One of the practical applications that has arisen from the study of maximal distance spatial relationships is related to coding theory. By placing codewords sufficiently apart, codewords which facilitate error detecting or correcting in a code can be realized (Roth [10]). The relationship of the dancing couples placement problem to this coding theory problem is whether by following their prescribed behaviors the dancers representing

codewords will find positions corresponding to the optimal codeword positions in a code solution space. After removing the role of the instructors as intelligent designers from the original narrative, this dancing couples scenario is restated as the "**dancers' placement problem**" and becomes the problem of focus for all subsequent chapters.

1.2 Overview

The focus of this research is to analyze a system composed of repulsive elements with an emphasis on two different properties of the system:

1. The first concern is the final system state which a behavior yields. Using this as the perspective, the resulting final system state determined by the element behavior is compared to a known optimal solution. It does not matter how the system got from state A to state B, only that it did. This would be more useful in applications where modeling the behavior of a system is of no significance.

2. The second concern is the accuracy of the model of behavior and how well the behavior corresponds to real applications. The system state should be closely aligned with the simulated system at any point in time for the prescribed behavior to reflect reality. As an example using the dancers' placement problem, the heuristics used in evaluating the solution outcome was derived from the idea that dancers may simultaneously attempt to improve their position by moving in a direction away from any other dancing pair until their position cannot improve. This sort of behavior model is also very much in line with research in particle swarm optimization.

The solution to the dancers' placement problem is congruent to more general modeling problems which will be introduced later. These related decision problems are already known to be np-complete, therefore an acceptable general purpose algorithm has not been discovered for solving the dancers' placement problem. For this research, a small subset of largely theoretical problem domains has been selected to analyze the solution space and perform the proceeding **heuristics** (a sequence of steps that yields a solution which may or not be optimal in the global sense) to obtain solutions for evaluation. Having no sets of data for the known application domains exhibiting similar repulsive behaviors to compare with the dancers' modeled behavior, no attempt to validate the dancers' modeled behavior as representative of any system is made and the behavior attributed to the dancers is not verified.

Chapter I serves to introduce the significance of distance relationships in common everyday experiences, presents how the structural relationships can be abstracted and viewed with different goal perspectives, and introduces a classification for repulsive behavior.

Chapter II further distinguishes the repulsive system focus of the research by the description of the dancers' placement problem, the analysis of the problem model application areas, and the comparison of the problem model with other known systems of repulsion. The applications presented are shown to be of significance in modern science.

Chapter III lays out the mathematical foundations for the remaining chapters and explains more formally any casual descriptions which were vague or ambiguous.

Chapter IV describes the heuristics, their emergent behavior, and proves termination for each heuristic. All heuristic dancer movements and resulting locations are based on a fixed step size. The heuristics are summarized as follows:

- The original heuristic used was based on an assumed simple greedy behavioral model. For this heuristic, each dancer iteratively acts on what they think is in their best interest, getting as far away from their closest neighbors as possible. The dancers only move if the position they move to increases the distance from their nearest neighbor. All of the heuristics evolved from the initial heuristic.

- The next heuristic allows conditions for a dancing couple moving to positions of equivalent distance from their closest neighbors. A dancing couple considers a position increasing the sum of all other dancing couples' distances to the couple as improvement, provided the new position still maintains the same minimum distance from their closest neighbor.

- Then, probabilistic annealing was applied to the original greedy heuristic model. The dancing couples choose at random whether to move closer to their nearest neighbor if no position farther from their nearest neighbor exists.

- Following this approach, a heuristic reducing the step size of movement was synthetically performed by enlarging the dancing area dimensions (effectively halving the step size proportional to the original area) and repositioning the dancers' positions to corresponding positions in the enlarged area. This helps alleviate cases where a dancing position based on the fixed step movements of

the dancers cannot lead to a position equidistant from any number of neighbors in the dance area, but by doubling the area, a position equidistance could be created. Another way to describe this is to consider a single dimensional space with an integer based fixed step size of 1. If two objects' positions are mapped to integers 3 and 6 respectively, what number is an equivalent distance between the two, 4 or 5? Neither, but by doubling the position locations to numbers 6 and 12, the number 9 is found as at an equivalent distance of 3 from each.

- Finally, in efforts to find a solution for the worst case configurations, a heuristic which does not solely base movement decisions on a dancer's one step view of improvement (local) is used. If a dancing couple recognizes it is in a bad position in the graph by its close distance to other dancers, they are able to look to open areas of the system and determine if any improvements could be made by moving an arbitrary number of steps to the open area .

Chapter V presents the test-bed cases and heuristic results. The four classes of graphs the heuristics will be considered on are the path, cycle, torus, and grid graph classes. The grid graph bears the most resemblance to a physical dance floor for the dancers' placement problem. Following the presentation of the test-bed cases, the results of the heuristics' execution are illustrated and analysis is given. Several poor solutions are included for demonstrating the shortcomings of heuristics given some of the available dancers' configuration states. It is not unreasonable to expect that the behavior of individual elements could restrict the optimal solution from ever being found.

Finally, Chapter VI will conclude with a summary of the heuristics' success and a discussion of future work areas. Due to the ubiquitous nature of repulsive systems in the physical and social universe, analysis of even a simple problem such as this may find relevance in a large area of scientific research.

CHAPTER 2

DISTRIBUTED SYSTEMS OF REPULSIVE ELEMENTS

In the previous chapter, spatial relationships were introduced and the problem of focus for the research was identified. The dancers' placement problem is now discussed in much greater detail and then related to other systems exhibiting similar behaviors.

2.1 Dancers' Placement Problem

A crowd of dance partners assembles on the dance floor for a cha-cha. There is a panel of judges that evaluates each dancing couple's performance. Each dancing couple has a number pinned on their dancing uniforms for the judges to record a score against. In order to remain clearly visible to the judges, each dancing couple will try moving as far away from the other dancers as they possibly can. This movement is not done in a sequential, turn based fashion but with all the dancing couples striving to do this at the same time. What is an appropriate strategy for the dancers to attain this maximal distance from neighboring dancers, and what does their distribution on the dance floor look like once they have achieved this maximal distance?

The dancing pairs are only concerned with the other dancers closest to them; they are not concerned with the density of dancers in any particular region near or far, nor are they concerned with their average distance from other dancing couples. If another dancing couple is 1 meter away from them, they do not care if all of the other dancers are a kilometer away. For them, there is no difference between this scenario and having all other dancers only 1 meter away. Time has counted down to the beginning of the competition. The dancers assume an initial random position as they enter the dance floor; once the competition has begun, the dancers' positions on the floor resembles a sort of secondary dance as the dancers act and react to the other dancers on the floor. Sometimes a dancing couple will move away from one couple only to draw closer to another, but they are all thinking with the same strategy. This strategy may benefit the entire community once the dancers have found their preferred positions. What does one gain by even considering a floor filled with dancers? Could this be used to analyze social interactions based on models with an assumed group behavior? Would dance sponsors be interested in determining the maximum capacity for the number of dancers on the floor? Could such social interactions model other physical behaviors? By following the dancers as they greedily contend for positions on the floor, one can readily observe how the movement of one dancing couple can affect so many others.

For an example, consider Figure 2.1 - Figure 2.3. Analyzing these figures shows that dancing couple 2 takes a step away from dancing couple 1 as they enter the floor. As a result, dancing couple 3 takes a step away from dancing couple 2, avoiding moving any closer to dancing couple 4. Dancing couple 4 takes a step away

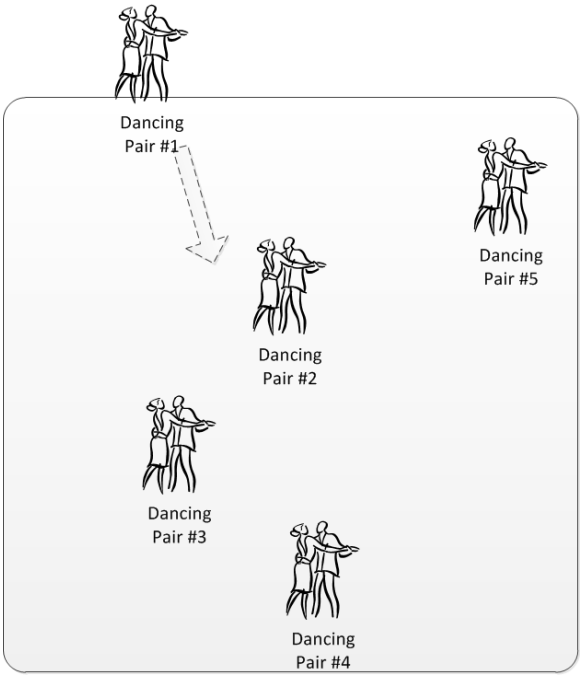

Figure 2.1: Initial dancer positioning ending with pair 1 placement

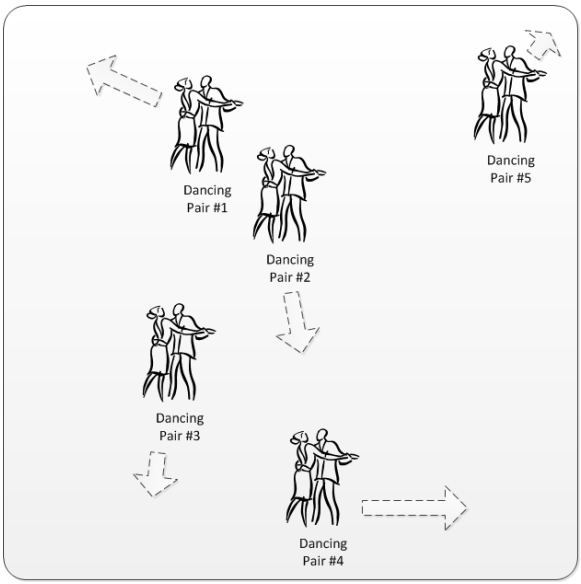

Figure 2.2: Dancers' movement following repulsive behavior, *initial position*

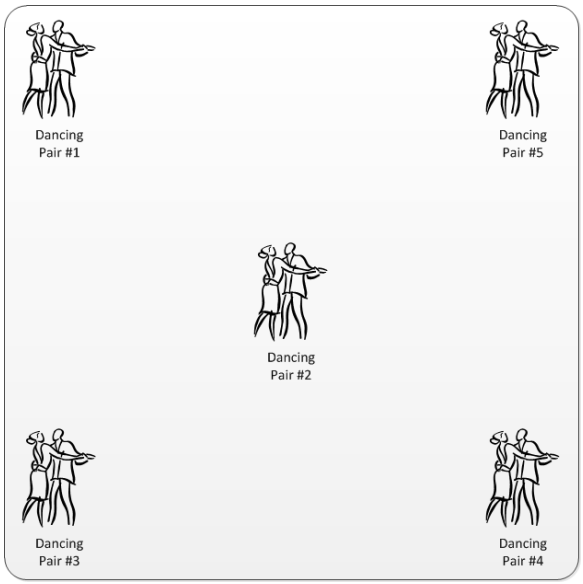

Figure 2.3: Dancers' movement following repulsive behavior, *final position*

from dancing couple 2 and dancing couple 3. Dancing couple 5 continues to isloate itself by taking a step away from dancing couple 1 and dancing couple 2, and finally, we see dancing couple 1 stepping away from dancing couple 2. For the dancers' placement problem, a **terminating position** is defined as a state where no dancing couples continue to move. For arbitrary initial dancing positions, will the dancers always reach a terminating position with their assigned behavior? If the dancers have achieved a terminating position, does a better placement exist if the dancers were initially placed differently or if they moved in a different direction sequence when they were repulsed by another dancing couple? If no better terminating positions exist for the dancers, the dancers are said to have reached a **globally optimal** position. This is because out of all the possible ways the dancers could have been placed, no improvement could have been made for maximizing the minimum distance between dancing couples. If better terminating positions exist, the dancers are said to be in a **locally optimal** position because each dancing couple cannot see how their positions could be improved based on where they currently are; but a better, globally optimal, position still exists. Without previously having considered the problem or having no recollection of previous dancers' terminating positions, the dancers may themselves be convinced that no improved positions exist.

The previous example considered does not impose any other rules or constraints on the dancers, except implicitly that they dance within a floor boundary and that they all seek to be as far from one another as possible. Now an additional constraint is added; consider a tiled dance floor arranged by aligned rows and columns of large tiles permitting dancers the space to dance within each tile. Following this,

the dancers' movement is restricted only to tiles sharing a side; that is, they are restricted to movement in a east, west, north, or south direction on the dance floor. See Figure 2.4 for an illustration of the floor layout and possible movements of dancers in such a scenario.

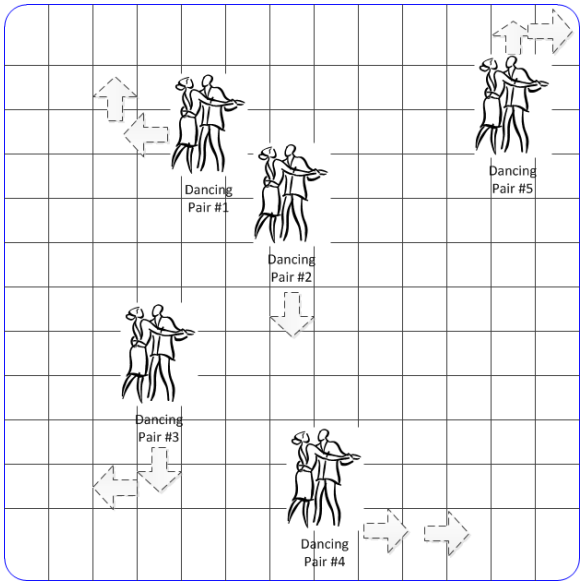

Figure 2.4: Dancers' movements on a tiled floor, *initial position*

If each dancing couple still follows the initial criteria of striving to be as far from every other dancing partner as possible, given the same dancing partners and the dimensions of the dance floor, what effect would this tiled movement have on the ability of the dancers to reach a terminating position? Statistically, how often do the dancers achieve a globally optimal terminating position in the tiled system from an arbitrary initial position? How do these statistics compare to the non-tiled example. How has the dancers' notion of distance changed? What sort of minimum distance could each couple possibly achieve from every other couple? Does the model of discussion still mirror the behavior as one might expect from human players? How

closely do the terminating positions and behavior of the dancers on the tiled floor correspond to that of the non-tiled floor? Figure 2.5 illustrates a close correlation between the outcome on a tile floor to that of a non-tiled floor.

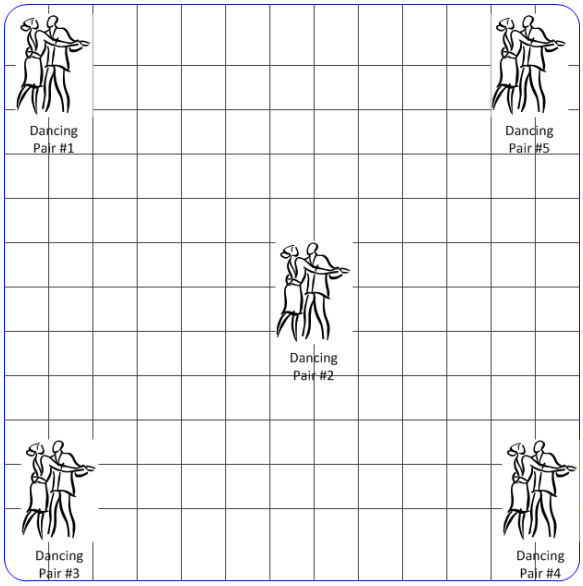

Figure 2.5: Dancers' movements on a tiled floor, *final position*

So just how can this restriction discretizing the dance floor and dancer movements affect the outcome? Perhaps not readily seen, the first impact of movement restriction is that dancers cannot move diagonally, even when it would be to their benefit. Let us examine the case, as in Figure 2.6, when all of the dancing couples line up in a straight line on the dance floor's diagonal, from the top left corner of the dance floor to the bottom right corner.

The dancers are all in close proximity to one another but there is plenty of dance floor open for them to break out and regain freedom of movement. Why would a dancing couple not take advantage of this available space? Consider the possible movements of dancing couple 3. Placed in the center of the floor, this dancing couple

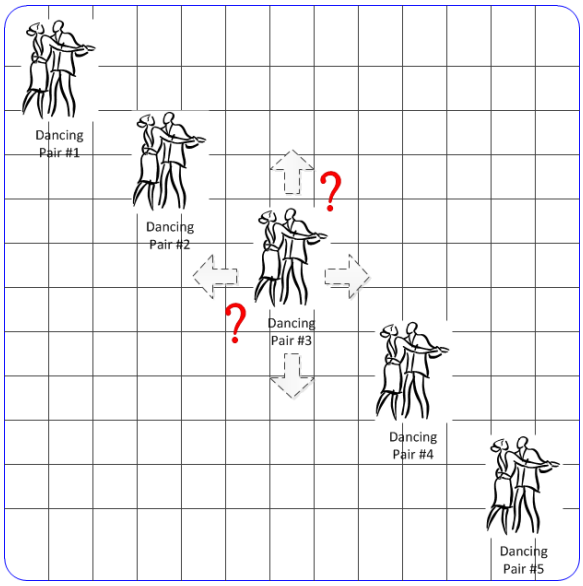

Figure 2.6: Dancers positioning themselves on the floor diagonal, *initial and final position*

can move north, south, east, or west, so why do they not move? If the dancing couple moves north, it puts them one tile closer to dancing couple 2, which is contrary to the defined behavior for the dancers. Similarly, if they move west, the dancing couple is once again one tile closer to dancing couple 2. Trying south or east directions will yield no better result, this time placing them closer to dancing couple 4. By only moving one step at a time, it is not possible for dancing couple 3 to reach the tile on the opposite diagonal to them without violating the constraints of their predefined behavior. This circumstance is not unique to dancing couple 3 but is common to all dancing couples on the floor. As a result, all of the dancers continue dancing close to one another on the dance floor's diagonal. From the example, it is clear that such circumstances exist. What sort of other constraining circumstances might there be? What if the dance floor was not a simple rectangle? Later, a more formal

mathematical model of the dance floor will be considered, and additional dancing positions which restrict movement prior to achieving an optimal configuration will be illustrated.

Now this dancing quandary will be informally restated as a goal-oriented function to be optimized. The goal of the dancers is to achieve the maximum-minimum distance from other dancing couples when considering all the different ways that dancers can be positioned on the dance floor. What should the dancers' behavior be in order to achieve this goal? To reach this goal, can one expect all the dancers to act identically, or does the behavior require a mixture of leader and follower mindsets? If the applied behavior will produce a globally optimal solution (terminating positions) for an arbitrary input (initial positions), the applied behavior will be referred to as an **algorithm**; however, if the applied behavioral approach does not always fully satisfy this goal, it is referred to as a **heuristic**. Heuristics rely on some observed attribute of the system in order to attain or at least be highly probable of attaining an outcome that is bound to a low error margin from the globally optimal outcome. The attribute used in all of the heuristics in this research is the observed relationship between a local view of each dancing couple's distances to one another and the minimum distance between each pairing of couples for the whole system. It is very possible that the behavior of the dancers may keep them from ever attaining a globally optimal solution.

2.2 Applications

Previously, there was a question asked, whether this type of behavior modeling could be relevant to any other physical applications. The previously cited dancing models all have their roots in systems with elements of repulsive behavior. For the remainder of the section, the focus will be on looking at such systems with functions of repulsion. These models can be very powerful, representing not only one practical modern day application, but numerous. The general concern of behavior in a repulsive system of elements is, given any N elements of a system and a function of repulsion (element behavior), what is the outcome for an arbitrary initial state? Inversely, often one is concerned with finding the function of repulsive that yields a distribution of the system which allows all the elements to meet a specified outcome. There are numerous models associated with systems of repulsive elements. Some are intermixed with rules of attraction; some allow for objects' repulsion to be determined by the sum of forces (Coulombic Charge) rather than individual object pairings. Given the variance in modeling repulsive systems, the focus of this research is narrowed to a smaller class of problems which have behaviors maximizing the minimum distance between any element pairing. There should be no object permitted to remain at a large distance from any other object while some dwell in close proximity. The following are examples of applications which largely deal with this class of problems.

2.2.1 Coding Theory

The first and most closely aligned problem considered external to the dancers' problem is related to coding theory. Modern technologies have led to an era of digital data; because the higher compression ratios and reliability found in a digital format allows for faster and more accurate data transmissions, data transmissions are almost always sent in the digital domain, consisting of a sequence of 1's and 0's. High frequency data transmissions often have an amount of transmission errors associated with them. In order to design equipment which is capable of handling transmission errors, the models which are used are probabilistic in nature. Using these probabilistic models, it is assumed that for every transmitted bit (a '1' or '0' of a binary sequence), the transmission error is independent of all previous or future events. Therefore, the probability of receiving 1 or n bits in error for a single data transmission can easily be calculated.

First consider the case where one is not concerned with possible data transmission errors, that is, that the message that is sent is the same as the message that is received. For such a case, the length of the whole transmission can be used for passing data because the integrity of the data is guaranteed to be preserved. Then consider the case where it is assumed that every bit has a probability p, where p is 0.97, of being received successfully and a probability 1-p of being received erroroneously (this means that a bit is received correctly 97% of the time and in error 3% of the time). Clearly, one cannot assume that the message that is transmitted contains the same information as what is received. Much worse than this, one cannot even determine

which bits are sent in error, that only the known "good" bits might be selected to interpret. What mechanisms could accommodate the transmissions of an unknown quantity and position of errors? The solution to such a problem in coding theory is to encode data in a binary sequence of greater length, which increases the chances of error in transmission, but guarantees, with a high probability, that a subset of bits size d should be transmitted correctly. Then, a decoder is able to successfully extract the transmitted information accurately or at least detect that an error occurred.

A trivial example of an error-detecting transmission is to designate a subset of the permuted binary sequences (**codewords**) with at least 2 bit positions different in value as a **code** representing true data, and consider any sequences outside this subset as erroneous transmissions. For this simple case, let the available codewords '00' and '11' represent the data values '0' and '1' respectively. In the event the receiver for any transmission receives a '01' or '10', the receiver will well know it to be in error and reject the value. Such a code is referred to as 1-bit **error detecting**. It is suggested that at a minimum the selected codewords be different in value for at least 2 bit positions. Such a condition is more commonly stated as a **Hamming distance** of 2, with Hamming distance being defined as the number of bit positions in which two binary sequences differ in value. If all pairings of the codewords maintain a minimum Hamming distance of n, this Hamming distance n determines the number of codeword transmission errors which can be detected (n-1) or corrected ($\lfloor \frac{n}{2} \rfloor$). Therefore, the method of selecting codewords for optimal error detecting and correcting is maximizing the minimum Hamming distance between codewords. The objective in this

binary sequence (codeword) design is to provide the minimum amount of redundant bits which achieves the required probability of transmitting k bits successfully.

Now consider a long sequential code transmission that must maintain a minimum probability that transmission errors can be detected or corrected upon reception. Given a transmission sequence of length n and k codewords, what is the greatest number of erroneous bits which can be detected for the collection of codewords (code) that is selected? How should such codewords be selected for an arbitrary sequence length of n? Is there a unique code that maximizes the error detection? One can expect that there is a limit to the number of bits in which the codewords of a code are different. Intuitively, for any number of codewords this number must be at least less than or equal to the length of the codeword n. As the number of codewords in a code increases, the maximum number of bits different in codewords is decreased. Without providing a more formal definition, the maximum number of codewords which can achieve a difference of x bit positions different from one another in a code is referred to as the **sphere packing bound**. A more formal defintion is provided in a later section. For an example of a code that meets its sphere packing bound (**perfect code**), see Figure 2.7. For this code, the bits p_i denote padding-bits for the data-bits, d_i, embedded in the codewords (Roth [10]).

The objective of code design in coding theory is nearly identical to the research focus problem of the dancer's placement problem. A very similar method of modeling the codewords can also be used to model dancers on the dance floor in the dancer placement problem. If dancers were used to represent codewords and the dance floor positions were modeled to reflect Hamming distance between codewords, an optimal

Hamming(7, 4)	
Data $\{d_0, d_1, d_2, d_3\}$	Codeword $\{p_0, p_1, d_0, p_2, d_1, d_2, d_3\}$
0000	0000000
1000	1110000
0100	1001100
1100	0111100
0010	0101010
1010	1011010
0110	1100110
1110	0010110
0001	1101001
1001	0011001
0101	0100101
1101	1010101
0011	1000011
1011	0110011
0111	0001111
1111	1111111

Figure 2.7: 2-Bit Error Detecting/1-Bit Error Correcting Hamming Code

solution to the dancer placement problem would perfectly map to a code meeting the sphere packing bound.

2.2.2 Atomic Modeling

Next, a physical application delving into the subatomic world is considered. Since Newton began formulating his laws of physics, scientists have been trying to gain deeper insight into object relationships and composition. Since the discovery of the atom, numerous models have been created in attempts to explain the behavior of interacting elements. Once the existence of electrons were evident, the relationship with electrons and energy became a huge research effort, as this is what plays the largest role in chemical interactions. Models were created that suggested all the

electrons in an atom are equally distributed on the surface of a sphere induced by the center (nucleus) of the atom. As the electrons would repel each other on the surface of the sphere, the problem of determing what layout configuration the electrons might arrange themselves became a matter of great scientific significance. Subsequently, a more accurate electron model was determined, but the problem is still of great interest to mathematicians and scientists alike, as there are many potential applications to which it can be mapped (Goldberg [7]). If a collection of point charges is fixed on the surface of a sphere, what is the configuration of the point charges? Is this equivalent to the minimum potential energy configuration of the system? The function of repulsion modeling point charges is not identical to the simple behavior attributed to our dancers. Charged particles are affected by the sum of the Coloumbic forces acting on them by all other charged particles, not only that of their closest neighbors (Cohn, Kumar [5]). What configuration patterns (an illustration of this is seen in Figure 2.8) emerge from the distribution of point charges?

As the force from neighboring particles is inversely related to the square of their distance, one might expect that the resulting distributions of point charges from the sum of Coloumbic charges is not so far from the dancers' eventual distribution based on a more simple function of repulsion. How similar is the model applying the dancers behavior to the charged particles on a sphere compared to the model using coloumbic forces?

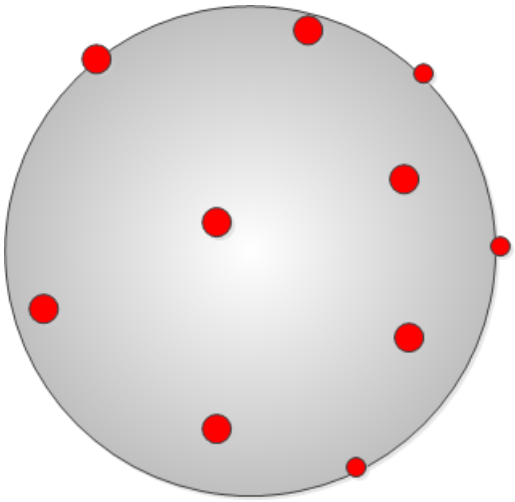

Figure 2.8: Point Charges Distributed On Sphere

2.2.3 Social Environments

Taking a step away from historical scientific observances, a model of animal habitats is suggested by the dancers' problem. Packs of wolves living in the forest try to maintain their territory by marking it. The territory lines may shift over time, but the wolves need resources, such as water and prey, to survive. The repulsion between these packs of wolves may be a complex function of topological distances and other environmental factors, in addition to the size of the pack or the strength in its number; but the wolf packs will distribute themselves by some laws of repulsion, as well as possible laws of attraction in the case of natural resources. If we assume the wolves follow a simple rule of repulsion related to straight line distance, how accurately would this reflect the real world analysis? What sort of metric is used to establish distance? Ignoring unfamiliar topological notions of distance, what if a homogenous environment with only fish was considered?

The examples stated don't reflect any real research on the particular habits of wolf packs or patterns of behavior in fishes, but there is research that seems very indicative that such modeling can give great insight into the behavior or configurations of animals. Specifically, given some rules of repulsion and attraction (artificial physics laws) representing animal behavior, how will the animals distribute themselves? The focus of such research is often not on merely determining the static placement of animals in their natural evironment, but also on what animal behaviors lead to such placements. For such cases, the focus of the research might not be on finding a particular goal configuration, but rather on addressing the behavior of individual elements in the system and how they affect the behavior of the whole system (swarm intelligence) (Amakobe, Grabczewski, Patel [1]). As this is very similar to the dual focus of the dancers' placement problem, one could consider either the static configuration of animals in their habitat or the model of animal behavior as worthy research pursuits. Restated from the dancers' perspective, the primary concern of the dancers may be the maximum space in which to dance, but they may also benefit from a pattern of thinking which helps meet the requirements of their primary concern.

2.2.4 Thomson Problem

Finally, consider a more well-known problem with a good deal of research already generated by J.J. Thomson. Thomson was a physicist who was involved in early atomic research. After creating his "Plum Pudding" model of the atom in 1904, he set out to solve the minimum energy of an n electron configuration on the surface of the sphere. While the atomic model itself has been changed several times

from its original discovery, the Thomson model is able to represent problems that are still relevant today. This problem of solving the minimum energy configuration on the surface of a sphere is applicable to many other science and theoretical math disciplines (Bowick, Cecka, Giomi, Middleton, Zielnicki [3]).

The study of multi-electron bubbles is one application of the original Thomson problem model. The study of this problem is concerned with the stable configurations of electrons on the sphere of bubbles formed by liquid helium. These bubbles are thought to be beneficial in the study of quantum effects from charges on curved surfaces. The known effects of such electronic systems are related to conductance, and the preliminary data on multi-electron bubbles shows them to be "unusual superconductors" (Silvera, Silvera Group [11]).

Virus shell assembly has been researched and potentially exudes similar configuration behavior to a problem related to distribution of favorable energies. The protein shell structures of many known viruses act as biological solutions to the Thomson problem (Center for Nonlinear Studies [9], Berger, Shor [2]).

There are obvious applications in communication services such as satellite or cell phone towers where the objective is to provide the greatest amount of coverage. As proportionality plays such a vital role in so many social and physical interactions, the Thomson problem, as maximizing a minimum parameter of a system, is found everywhere (Bowick, Cecka, Giomi, Middleton, Zielnicki [3]).

2.3 Problem Model

Now consider how this problem is to be modeled. In most real world modeling applications, the notion of distance in which the objects interact is mapped in the real number domain. This allows for an infinite number of potential points for objects to configure themselves. Real domains can also be represented by discrete domains by mapping points from the real domain into points in a discrete domain in an effort to reduce the computational complexity of a model; or in some cases, a discrete model is ideal for describing the domain, such as coding theory, where codewords are typically defined by a finite bit string. Often a graph is an appropriate model for a discrete domain. Before going further in the usage of "graph", a brief description is provided of a graph to help illustrate the model until more formal definitions are provided in a later chapter. A graph is a collection of objects (vertices) which are associated or connected with one another through edge relationships. The edges themselves do not describe any concrete notion of the relationship, but either by known context or by labeling the graph, the relationship is illustrated. A family tree is a graph example that is well known to many. Here, the edges denote a familial relationship. If a problem existing in the real number domain is not modeled accurately after mapping to a discrete domain (graph), the density of the graph can be adjusted to better reflect the domain of the real problem. If a graph is successfully mapped and oriented on a Euclidean plane, the notion of distance from one object to another is the sum of the edges between the shortest route from one object to another.

So why use a graph to represent a problem? Because a graph is composed of discrete objects, there is a finiteness to the system, and it is easier to scale the problem to meet the requirements of the method of solution. The structural nature of graphs also allows the problem model to be unaffected by the topology or metric space in which the problem exists. Graphs are easily understood representations of problem domains if they are labelled properly. In some cases, abstracting a problem to a graph provides many tools for analysis of the problem, as already known graph theorems might reveal some undiscovered truth about the original problem.

CHAPTER 3

DEFINITIONS

This chapter lays the foundation of the mathematical theory behind the applications and heuristics discussed hereafter and provides examples alongside definitions to give greater insight into how the dancers' problem model relates to its real world counterpart. The heuristics applied to the dancers' placement problem span several disciplines of mathematics including: Metric Spaces, Probability, and Graph Theory.

3.1 Metric Spaces

A metric space (X,d) is a set X and a "distance function" $d : X \times X \to \mathbb{R}$ that assigns a real number for each pair of points in X. This simply means that between any two objects of a set, a value for distance is defined and the following criteria is met:

- The distance between the two objects is non-negative. i.e. $\forall x, y \in X$, we have $d(x, y) \geq 0$.
- The only case in which the distance between two objects is zero is the case of equivalent objects, i.e. $x = y$

- The distance between two objects is symmetric, i.e. $d(x, y) = d(y, x)$
- The triangle inequality is true, i.e. $\forall x, y, z \in X,\ d(x, y) \leq d(x, z) + d(z, y)$

The metric space plays a particular role in the dancers' placement problem because maximizing the minimum distance parameter of the modeling graph is the focus of this research. A more concrete example of a metric space is a Euclidean space with the Euclidean norm (ℓ^2). This is how most individuals think of distance. Consider the general ℓ^p form $\|x\| := \sqrt[p]{|x_1|^p + ... + |x_n|^p}$ which is equivalent to the Pythagorean Theorem for a 2-dimensional Euclidean space, i.e. $\|x\| := \sqrt{|x_1|^2 + |x_2|^2}$. The distance used in modeling the dancers' placement problem does not follow the conventions of geometric graphs that one is accustomed to as it does not use a Euclidean norm, but for lack of better graph related definitions, the notion of distances between objects in a graph will be defined later.

3.2 Probability

As the dancers' placement problem involves random decision processes, in order to understand the analysis of complexity, it is necessary to introduce some definitions related to probability. First for probability, a **random variable** is defined as a function that maps an event outcome to a real number. A trivial random variable used later for modeling the heuristic complexity is the **indicator random variable**:

$$I(\xi) = \begin{cases} 1 & \text{if } \xi \text{ occured} \\ 0 & \text{if } \xi \text{ did not occur} \end{cases}$$

A **random process** is an indexed family of random variables. Further, a **discrete-time random process** is a random process whose index set is countable (for the case of computational complexity, the non-negative integers):

$$I(j, \xi) \quad j \in \mathbb{Z}_+$$

Finally, the method by which we model the probability of state occurrence in our heuristics is defined. **Markov Chains** represent random processes whose future events are independent of the all previous events but the current event (Garcia [8]). The mathematical expression for this independence is

$$P[X(t_{k+1}) = x_{k+1} | X(t_k) = x_k, ..., X(t_1) = x_1] = P[X(t_{k+1}) = x_{k+1} | X(t_k) = x_k] \ .$$

The dancers' placement problem can be modeled by Markov Chains because the probability of transitioning from one dancers' position state to another is only dependent on the current dancers' position state. There is much more to be said on the properties of random variables, processes, and Markov Chains; however, the simplicity of the model used for the complexity analysis of the heuristics should sufficiently communicate all that is necessary for understanding the analysis process.

3.3 Graph Theory

Now a more detailed description of a graph is introduced. Graphs are very versatile for modeling almost any problem because of their relational structures. The primary reason they have been selected as the model for this research was how well they map to geometric shapes and the known application in Hamming codes which are represented by graphs. A working definition of a graph for this research follows:

A **graph** G is a triple consisting of a **vertex set** V(G), and **edge set** E(G), and a relation that associates with each edge two vertices (not necessarily distinct) called **endpoints**. Edges can also be thought of as an unordered pair of vertices $\{v_1, v_2\}$. The notion of being an unordered pair is significant as swapping the first vertex in the pair with the second does not represent a different edge. This graph is said to be **non-directed**. Using a non-directed graph is an appropriate manner to model distances because of the previously stated property of the metric space: "The distance between two objects is symmetric". While they will not be used in the scope of this paper it may of interest to the reader to note that there are directed graphs which have edges represented by ordered pairs; as expected, these graph are referred to as **directed**. Another small but important concept is the **complete graph**; complete graphs are graphs with an edge relationship between every vertex pairing for unique vertices in the graph.

Next the structure used to maintain the dancers' position state is defined. A **subgraph** of a graph G is a graph H such that $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$ and the assignment of endpoints to edges in H is the same as in G. A subgraph is formally written $H \subseteq G$ and described by "G **contains** H". This means that any vertices or edges found in a subgraph are found in the original graph. In order to maintain the dancers' position state, a subset of those graph vertices which the dancers are said to occupy is kept record of throughout the heuristic. Related to subgraphs and useful for the discussion of heuristic complexity, a **clique** is a subgraph that is a complete graph.

In order to define the notion of distance for the dancers' placement problem, it is necessary to several other graph concepts. A **path** is a unique finite sequence of non-repeating, adjacent vertices in a graph (see Figure 3.1). A **u,v- path** is further identified as a path whose beginning and ending points are u and v respectively; the non-ending points of the path are referred to as **internal vertices**. A graph which consists of only the points in a path is also called a path (see Figure 3.2). In order to avoid confusion between the usages of the term path, it may be useful to consider a u,v- path sequence as a subgraph, restricting the notion of path to represent a graph consisting of only a sequence of non-repeating points. Paths are significant to understand in that they determine the distance between two vertices. The **length** of a path is its number of edges; because the vertices of a path are non-repeating, the length is equivalent to the number of vertices in the path minus one. If G has a u, v-path then the **distance** from u to v, written d(u, v) is the least length of a u, v-path. This serves as the definition for the distance between objects in the dancers' placement problem model.

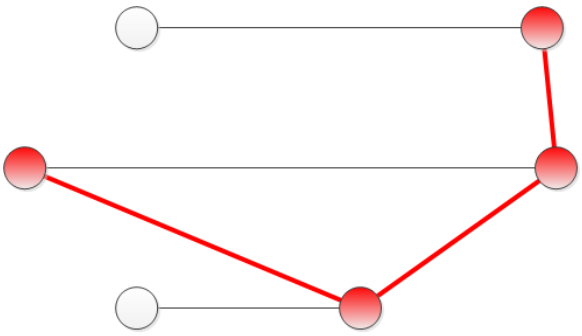

Figure 3.1: A path as a sequence in a graph

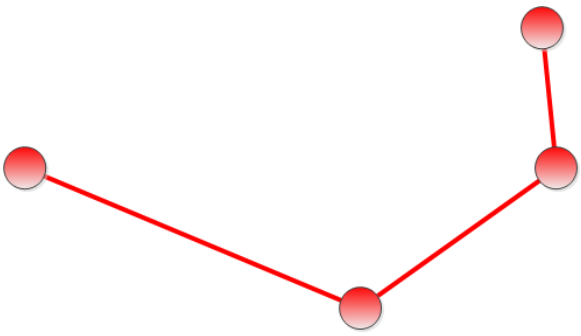

Figure 3.2: A subgraph of the preceding graph representing a path graph

The basis for selecting a graph as our model has already been addressed. The scope of the dancers' placement problem has a very large domain and while the solution we seek would be best as viable across the entire domain, we restrict the scope of the research to a smaller group of graphs to examine the behavior with the hopes of also revealing properties relevant to any system in the domain. The graphs themselves may be approximations of a local problem space. A dance floor does not consist of a finite number of discrete positions in reality, but one might expect that such a graph solution would yield an appropriate approximation. There is an amount of variance in the graph modeling that must be controlled and this is done through the **density** of the graph. A dense graph is a graph in which the number of edges is close to the maximal number of edges. The object of defining density is to establish that the quality of a real domain problem being modeled discretely can be controlled by including more points and their local distance relationships; hence, the greater number of points and their associated relationships, the greater the number of edges in the graph. The basis for selecting the particular graph models with east, west, north, south object relationships was somewhat arbitrary, but there is some similarity between this model and the models of Hamming codes using hypercubes.

3.3.1 Graph Classes

Four general graph classes were selected to apply heuristics to: the path, cycle, grid, and torus. For modeling the dancer's problem, a different metric function than the usual ℓ_2 norm (Euclidean distance) will be used, an ℓ_1 norm. The ℓ_1 norm corresponds to what is known as a **Manhattan** metric space or a **taxi car** space.

The first graph considered for our heuristic was the grid. The grid graph is what one obtains for a rectangular region if one uses a ℓ^1 norm (see Figure 3.3). The ℓ^1 norm is identical to the Hamming distance metric.

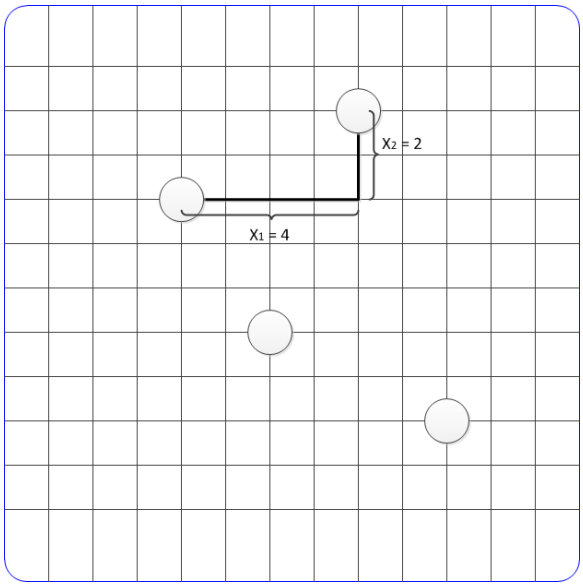

Figure 3.3: ℓ^1 norm: distance $= |\Delta x_1| + ... + |\Delta x_n|$

Of the graphs considered, this graph most closely represents a traditional dance floor. The grid graph appears as an arrangement of straight line rows and columns. Each line intersection is assigned a vertex and the line segments created by the vertex assignment represent the grid edges. The dimensions of this grid all depend on the number of vertical and horizontal lines used for the rows and columns.

Paths were selected for easy analysis of the heuristic solution to a globally optimal solution. Using the previous definition for a graph, a path graph P_n is a path with n vertices and a length of n-1.

The next natural progression from paths is to evaluate **cycles**. A **cycle** is a graph with a small number of vertices and edges whose vertices can be placed around

a circle so that two vertices are adjacent if and only if they appear consecutively along the circle. A cycle is easily created by taking a graph P_n and adding an edge relationship between the endpoints (West [12]).

Finally, just as the cycle was an extension of the graph, the grid graph was extended to create a **torus**. This is done by wrapping the boundaries of the grid by establishing edge relationships between the endpoints of every row and column.

3.3.2 Graph Parameter

A **parameter** of a graph is an attribute of the graph as determined by the evaluation of a specified function given a graph instance as an argument. The parameter that we are interested is the maximum-minimum distance achievable between each k selected vertices in our graph. We label our parameter as MaxMinDancers (**MMD**) and define it as $MMD(G,k) = MAX(\{MIN(\{d(x,y) \mid x \in S, y \in S\} \mid S \subseteq G, |S| = k\})$.

Another important aspect of our graph parameter when dealing with surface mapped graphs is **Sphere Packing**. Sphere packing is the placement of non-overlapping spheres in a metric space. Sphere packing is a form of a more general decision based packing problem. A common statement of the problem for a graph is: given graph G=(V,E), a positive integer $K \leq |V|$, and a positive integer d, is there a set $S \subseteq V(G) \ni |S| \geq K$ and $u, v \in S \implies dist(u,v) \geq d$ (is the packing number $p(G) \geq K$)? It should be noted that typically 2-dimensional spaces refer to sphere packing as circle packing, but given the potential non-planar nature of a graph and its connection to hypercubes in coding theory, sphere packing will be used when

referring to the packing problem. An important concept for the closed form bounds of our 2-dimensional graphs (Grid and Torus) is the **sphere-packing bound**; this bound is the maximum sphere packing for a given space. Typically, for ℓ^2 norms, this corresponds to the number of non-overlapping circles that can be placed in a specified space. However, because our graphs are modeled in such a way that the ℓ^1 norm represents the appropriate notion of distance, the geometry of the shape required not to overlap is not a circle, but a diamond (see Figure 3.4 and Figure 3.5).

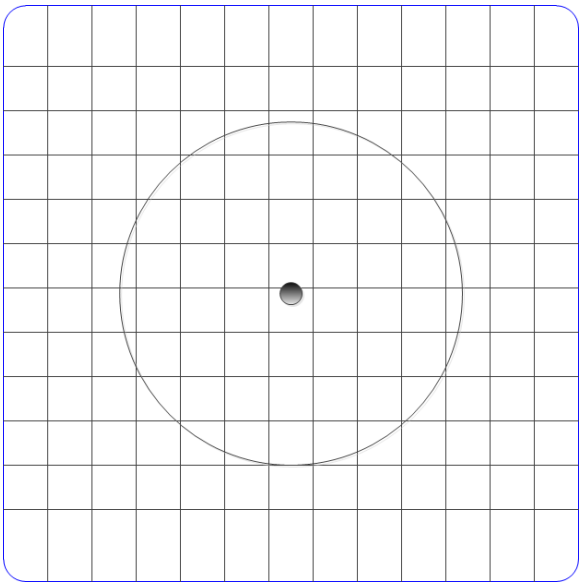

Figure 3.4: ℓ^2 norm sphere:
$r = \sqrt{|\Delta x_1|^2 + |\Delta x_2|^2}$

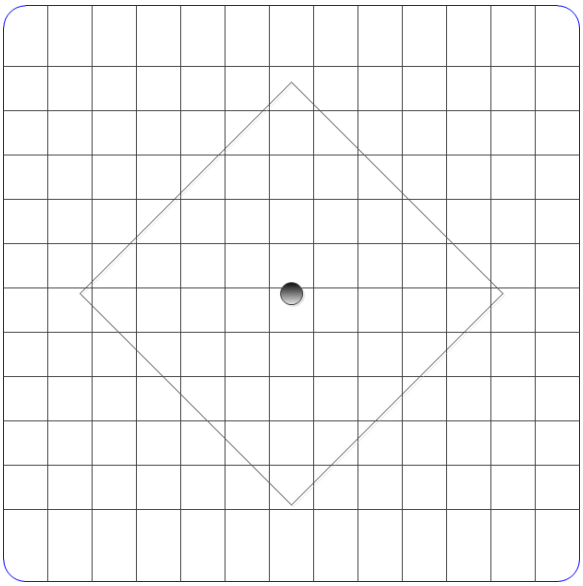

Figure 3.5: ℓ^1 norm sphere:
$r = |\Delta x_1| + |\Delta x_2|$

When we consider sphere-packing for the dancers' placement problem, the graphs are not limited to the space internal to edges in the graph; specifically, the grid graph may contain spheres centered on its corners even though the area of the sphere extends beyond the boundaries of the graph edges. In addition to their application in the dancers' placement problem, sphere-packing bounds are also the attribute

in coding theory that determine whether or not a code is 'perfect'; given the close relationship between coding theory and the dancers' placement problem, this should not come as a surprise. Let G be an arbitrary graph, the following properties are important for establishing the closed form bounds later set for the grid and torus graphs:

- All the vertices of a subset S of graph G are outside a sphere of radius r centered at an arbitrary vertex v $\iff$ All the vertices of a subset of graph G are a minimum distance of at least r from an arbitrary vertex v
- Graph G contains a sphere of radius r packing of k $\iff$ Graph G contains k vertices that maintain a minimum distance of at least 2r between any pairing of unique vertices of the k vertices
- There are k spheres of radius r centers in a graph G with a sphere of radius r packing bound of k $\iff$ The centers of k spheres of radius r maximizing the covered space in a graph G represent a globally optimal solution to the dancers' placement problem.

CHAPTER 4

HEURISTICS

In the previous chapters we have described the motivation and potential applications of solutions to the dancers' placement problem as well as briefly describing the steps of the various heuristics. Here the heuristics are described in much greater detail with their behavioral attributes. Much of the variance in approaches was the result of analyzing solutions modeled by an earlier heuristic and determining cases not well met by the current heuristic. The nature of the general heuristic performance is very case dependent and the number of graphs considered is quite limited. The central focus of evaluating the heuristics is in the solution produced when the heuristic terminates. The overall objective of heuristic design was finding a general solution for an arbitrary graph, but as with other optimization problems, there likely is not always one that best fits the entire problem domain. The basis of evaluation was limited to comparisons of the terminating state solution and their parameter value (MMD) to the parameter value of the known optimal solution (unique or non-unique).

4.1 Strict Local Improvement

The first heuristic assumed the improvement of the system parameter by strictly improving individual positions. Here follows the strict local improvement heuristic used for the maximizing the minimum distance as described in the dancer's problem.

Let G be an arbitrary graph with each vertex representing a position on the dance floor. Let S $\subseteq$ G be the set of selected vertices where the dancers are currently positioned. We randomly select a vertex from S representing a dancer who is evaluating their current position and considering movement. The dancer may take one step in any direction (to any adjacent vertex) where the new position strictly increases the minimum distance between itself and any other vertex in S. If the dancing couple finds a better position and moves, then S is modified by removing the vertex representing the dancers' previous position and appending the dancers' new position. If multiple improved positions are found, then one is selected at random. Let this modified set be denoted by S'. If S' has been modified then the process restarts by randomly selecting another dancing couple to evaluate their position. If S was not modified for the first randomly selected vertex (dancers fail to find improved position) then that vertex is flagged as visited and another vertex is selected at random from the remaining unflagged vertices. If any subsequent randomly selected vertex finds an adjacent vertex (dancers find an improved position) to increase its distance from other selected vertices then all flags are removed from vertices, and the process starts all over again with S' as the dancer's initial position. The heuristic terminates

once all vertices in S have been flagged as visited (all dancers have evaluated new positions) for finding an improved position.

4.1.1 Behavioral Perspective

This heuristic is based largely on a model of assumed human behavior. Dancers may not have the ability to distinguish which positions will ultimately lead to the greatest amount of space for themselves but instead focus on what local adjustments can be made to improve their position. Some obvious differences might be that while this heuristic considers only one dancer at a time, it is to be expected that in reality, all dancers might be maneuvering for improved positions simultaneously. In addition, all dancers are assumed to have the same rules governing their analysis of movement, one example of this would be the established fixed length of the steps for all their movements (as modeled by the discrete graph).

4.1.2 Termination

Termination is proved using a monotonic state function. To prove program termination we define the ranking state function as

$$f(S) = \sum_{1 \leq i \leq k} \frac{1}{k^{\mathrm{D}_i}} \ ,$$

where k represents the number of dancers and D_i is the minimum distance from dancing pair i to any of the other k-1 dancing partners. By the rules of the heuristic, the value D for some vertex i must be increased for each state transition. As D

increases for an arbitrary vertex i, D may decrease for some x number of vertices. The value of D_i prior to the move must be less than the value of D for the x vertices affected by the move because of the heuristic rules (otherwise the move would result in a worse position). Therefore because

$$\frac{1}{k^{n-1}} = \frac{k}{k^n} = \sum_{1 \le i \le k} \frac{1}{k^n} \ge \sum_{x+1} \frac{1}{k^n} + \sum_{\substack{1 \le i \le k \\ \mathrm{D}_i > n}} \frac{1}{k^{\mathrm{D}_i}} \ ,$$

it is apparent that the change caused by decreases of D in x vertices will never be as great as the change caused by the increase in D_i, and the function $f(S)$ decreases. Therefore, $f(S)$ is monotonic and a terminating bound exists, proving termination for the heuristic.

4.2 Weighted Equivalent Improvement

The second iteration of the heuristic, weighted equivalent improvement, takes a less strict approach to improved positions. For this heuristic, evaluation of improved positions considers an increased minimum distance between the nearest neighbors primarily but also considers the distances of other vertices that are already at greater distances than the minimum.

Let G be an arbitrary graph with each vertex representing a position on the dance floor. Let $\mathrm{S} \subseteq \mathrm{G}$ be the set of selected vertices where the dancers are currently positioned. We randomly select a vertex from S representing a dancing couple who is evaluating their current position and considering movement. A dancing couple evaluates the position they would attain by taking one step in any direction (to any

adjacent vertex) basing improvement on whether or not the minimum distance from its closest neighbor would increase. In the event that the minimum distance does not increase but is equivalent and the summed distances from all vertices in S excluding the vertex under consideration (dancing couple evaluating movement) is greater than the summation from the position where the selected dancer currently resides, this position is also considered an improvement. If the dancer finds a better position and moves, then S is modified by removing the vertex representing the dancer's previous position and appending the dancer's new position. If multiple improved positions are found, than one is selected at random. Let this modified set be denoted by S'. If S' has been modified then the process restarts by randomly selecting another dancing couple to evaluate their position. If S is not modified for the first randomly selected vertex (dancing couple fails to find improved position) then that vertex is flagged as visited and another vertex is selected at random from the remaining unflagged set S. If any subsequent randomly selected vertex finds an adjacent vertex to improve its distance from other selected vertices (dancers find an improved position) then all flags are removed, and the process starts all over again with S' as the dancers' initial position. The heuristic terminates once all vertices in S have been flagged as visited (all dancers have evaluated new positions) for finding an improved position.

4.2.1 Behavioral Perspective

This heuristic modifies the dancers' decision process to be more accommodating in their view of position improvement. If a dancing couple perceives that a move will not make their situation worse and observes that the new position will take them

farther away from a greater number of dancers, they will take the position and hope that, as the other dancers distribute themselves optimally, more space will become available and their position will again improve. All of the general behavior constraints remain present, as all of the dancers adhere to the same standard of rules and their movement step sizes remains fixed.

4.2.2 Termination

Termination is proved using a monotonic state function. To prove program termination we define a ranking function as

$$f(S) = \sum_{1 \leq i \leq k} \frac{1}{k(\mathrm{D}_i + \mathrm{DSUM})}$$

with

$$\mathrm{DSUM} = \frac{\left(\sum_{1 \leq i \leq k} \sum_{\substack{1 \leq j \leq k \\ j \neq i}} dist(i,j) \right)}{k^2 MAX\{Img(dist)\}} \ ,$$

where k represents the number of dancers, D_i is the minimum distance from dancing pair i to any of the other k-1 dancing partners, and dist represents the graph's distance function. By the rules of the heuristic, the value D for some vertex i or DSUM must be increased for each state transition. As D or DSUM increases for an arbitrary vertex i, the function $f(S)$ decreases. Therefore, $f(S)$ is monotonic and a terminating bound exists, proving termination for the heuristic.

4.3 Simulated Annealing

The simulated annealing approach attempts to add an element of randomness in searching through the global configuration solution space. In this heuristic, simulated annealing is performed by adding an additional decision point to our general distance increase rule by initializing a threshold value to be compared to a uniform pseudo-random variable sample. The threshold value is decremented after every comparison and upon a positive comparison result, the general distance increase requirement is ignored. If a stable position has not been reached by the dancers by the time that the decrementing threshold value reaches zero, the heuristic become identical to strict local improvement and algorithm will terminate in the same manner.

Let G be an arbitrary graph with each vertex representing a position on the dance floor. Let $S \subseteq G$ be the set of selected vertices where the dancers are currently positioned. We randomly select a vertex from S representing a dancer who is evaluating their current position and considering movement. A dancer evaluates the position it would attain by taking one step in any direction (to any adjacent vertex) based on whether or not the minimum distance from its closest neighbor would increase. If the distance would not increase, a random sample is taken and compared to the threshold value. If either the distance increases or the threshold value is greater than the random sample, the position is considered to be a better position. If the dancer finds a better position and moves, then S is modified by removing the vertex representing the dancer's previous position and appending the dancer's new position. If multiple improved positions are found, then one is selected at random. Let this modified set

be denoted by S'. If S' has been modified, then the process restarts by randomly selecting another dancing couple to evaluate their position. If S is not modified for the first randomly selected vertex (dancing couple fails to find improved position), then that vertex is flagged as visited and another vertex is selected at random from the remaining unflagged set S. If any subsequent randomly selected vertex finds an adjacent vertex to improve its distance from the other selected vertices (dancers find an improved position), then all flags are removed; and the process starts all over again with S' as the dancers' initial position. The heuristic terminates once all vertices in S have been flagged as visited (all dancers have evaluated new positions) for finding an improved position.

4.3.1 Behavioral Perspective

The motive for developing this heuristic was non-behavioral in nature and added as a means to design a heuristic capable of reaching more optimal terminating states. The introduction of randomness in behavior introduces an odd notion of behavioral conformance to the dancers which makes this behavioral approach seem less plausible. This could be viewed as the insight that if a dancer has no known position improvements, they can still try to move in a direction in hopes that it will eventually lead to something improved. It would be hard to imagine that the dancers collectively maintain a count of the number of times any dancing couple has moved in a random manner from which to make a decision of whether any future random movements are permitted. Therefore, such a random behavioral notion would probably remain present in the dancers indefinitely (a stable position, terminating

state, would never be reached). This could also be thought of as a mechanism for individual thought, rather than the herd mentality prescribed by the other behaviors.

4.3.2 Termination

Termination is proved using a monotonic state function. To prove program termination we define a ranking function as

$$f(S) = \sum_{1 \leq i \leq k} \left(\frac{1}{k^{D_i}} + C \right) \ ,$$

where k represents the number of dancers, C is the probabilistic annealing threshold count, and D_i is the minimum distance from dancing pair i to any of the other k-1 dancing partners. By the rules of the heuristic, the value D for some vertex i must be increased or the annealing threshold count decreased for each state transition. As D increases for an arbitrary vertex i or the annealing threshold count C decreases, the function $f(S)$ decreases. Therefore, $f(S)$ is monotonic and a terminating bound exists, proving termination for the heuristic.

4.4 Step Reduction

The step reduction heuristic addresses an artifact of modeling more than acting on some observed attribute of the problem. The discrete nature of the dancer's problem model means that dancers cannot always attain the greatest distance from their closest neighbors because this requires positions that are intermediate to the discrete placement they are allowed. This also affects their ability to move between

dancers on opposite sides of them. This heuristic is tightly coupled to the observation that a dancing couple may find themselves between multiple dancers with no position equidistant from their closest neighbors; but if they were able to move to a more equidistant position, this might open up movement for other dancers. For the discrete case of the heuristic, we double the graph size and move the selected vertices to correspond to positions indexed by double their previous indexed state. Consider a simple path of 7 vertices labelled 1 through 7 with 4 dancing couples. A terminating state for this path has dancing couples occupying vertices 1, 2, 4, and 7 respectively. If the graph is doubled these dancing couples are mapped to vertices 2, 4, 8, and 14 of the new doubled graph. After iterating through the dancer movements again, the terminating state of these dancing couples will be on vertices 1, 5, 9, and 14 respectively. If these positions were uniformly mapped back to the vertex label range of the original graph, the positions selected for the dancing couples would be on vertices, 1, 3, 5, and 7 respectively. Such a position is an improvement on the terminating state reached prior to doubling the graph size. If the vertex range of the doubled graphs are mapped back to the original graph, this act of doubling the graph can be seen as reducing the size of the dancers' steps.

Let G be an arbitrary graph with each vertex representing a position on the dance floor. Let $S \subseteq G$ be the set of selected vertices where the dancers are currently positioned. We randomly select a vertex from S representing a dancer who is evaluating his current position and movement. A dancer evaluates the position it would attain by taking one step in any direction (to any adjacent vertex) based on whether or not the minimum distance from its closest neighbor would increase. In the event

that the minimum distance does not increase but is equivalent, the distances of all vertices in S, excluding the selected vertex, are summed. If this summation is greater than where the selected dancer currently resides, this position is considered an improvement. If the dancer finds a better position and moves, then S is modified by removing the vertex representing the dancer's previous position and appending the dancer's new position. If multiple improved positions are found, then one is selected at random. Let this modified set be denoted by S'. If S' has been modified,then the same process of random selection from all the vertices of S' is carried out to find an improved position. If S is not modified for the first randomly selected vertex (dancer fails to find improved position), then that vertex is flagged as visited and another vertex is selected at random from the remaining unflagged set S. If any subsequent randomly selected vertex finds an adjacent vertex to improve its distance from other selected vertices, then all flags are removed, and the process starts all over again with S' as the dancer's initial position. Once all vertices in S have been flagged as visited for finding an improved position, the graph has reached an intermediate terminating state. Following this intermediate terminating state the graph is doubled in size as are the equivalent positions of the dancers. Once the graph is doubled, the modified dancer positions are used as the dancers' starting position. The heuristic follows the same steps from the beginning until, once again, all vertices in S have been flagged as visited for finding an improved position and the graph has reached another intermediate terminating state. Each time the graph reaches a terminating state, the graph is expanded and all of the movement steps are executed again. Ideally a terminating condition could be selected as the distance between the closest neighbors of all danc-

ing couples being equivalent, but as the positions required for such globally optimal solutions from a real number domain are not necessarily rational or achievable by the selected method of increasing the graph, the number of times the graph can be expanded is fixed to a constant positive integer C to prevent the heuristic from running indefinitely.

4.4.1 Behavioral Perspective

The behavior attributed to the dancers for this heuristic is the approach that dancers may need to reduce the size of their steps in order to find a position that maintains their current minimum distance or improves it while moving to a more open area. All of the general behavior constraints remain present, as all of the dancers adhere to the same standard of rules and their relative movement step size remains fixed, effectively decreased with each graph increase.

4.4.2 Termination

Termination is proved using a monotonic state function. To prove program termination we define a ranking function as

$$f(S) = \sum_{1 \leq i \leq k} \left(\frac{1}{k(\mathrm{D}_i + \mathrm{DSUM})} + C\right)$$

with

$$\mathrm{DSUM} = \frac{\left(\sum_{1 \leq i \leq k} \sum_{\substack{1 \leq j \leq k \\ j \neq i}} dist(i,j)\right)}{k^2 MAX\{Img(dist)\}} \ ,$$

where k represents the number of dancers; C is the number of step-reductions performed; D_i is the minimum distance from dancing pair i to any of the other k-1 dancing partners; and *dist* represents the graph's distance function. By the rules of the heuristic, the value D for some vertex i or DSUM must be increased for each state transition or the value of C is decremented. As D or DSUM increases for an arbitrary vertex i or the value of C decreases after a step reduction, the function $f(S)$ decreases. Therefore, $f(S)$ is monotonic and a terminating bound exists, proving termination for the heuristic.

4.5 Move To Open Area

The final heuristic considered is one that views vertex distribution from a different perspective. This heuristic was the result of a response to a special case graph referred to in this research as the "antenna graph"' (see Figure 4.1) that none of the previously mentioned heuristics handle very well. Consider the "antenna" graph with vertices u and v representing the nodes with dancers on them. The path below the two vertices could be an arbitrary length, but using the previous heuristics, the MMD parameter of the graph would always remain 2, even though the optimal solution would have a parameter equal to one less than the length of the path below. Even excluding consideration of this "antenna graph", the solutions obtained using this heuristic have some very nice properties. The observation that a non-dancer position should not have a greater distance to a dancers' position than the current MMD is incorporated into the heuristic, ensuring that non-dancer positions are less than or equal to the distance parameter MMD of a terminating state of this heuristic.

The proof is not considered, but the parameter MMD for a solution of this heuristic should be at least half of the MMD parameter of the optimal solution.

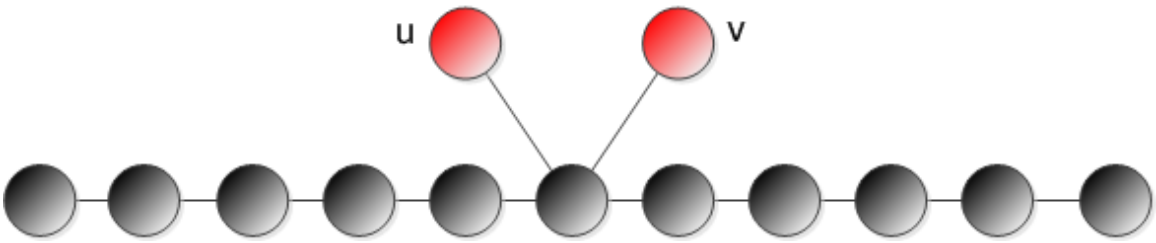

Figure 4.1: The "Antenna" Graph

Let G be an arbitrary graph with each vertex representing a position on the dance floor. Let S $\subseteq$ G be the set of selected vertices where the dancers are currently positioned. We randomly select a vertex from S representing a dancer who is evaluating his current position and movement. A dancer may take one step in any direction (to any adjacent vertex) where the new position strictly increases the minimum distance between itself and any other vertex in S. If the dancer finds a better position and moves, then S is modified by removing the vertex representing the dancer's previous position and appends the dancer's new position.If multiple improved positions are found, then one is selected at random. Let this modified set be denoted by S'. If S' has been modified, then the same process of random selection from all the vertices of S' is carried out to find an improved position. If S is not modified for the first randomly selected vertex (dancer fails to find improved position), then that vertex is flagged as visited; and another vertex is selected at random from the remaining unflagged set S. If any subsequent randomly selected vertex finds an adjacent vertex to improve its distance from other selected vertices, then all flags are removed, and the process

starts all over again with S' as the dancer's initial position. Once all vertices in S have been flagged as visited for finding an improved position, all non-dancer vertices distances from a dancer are evaluated. If there exist any non-dancer positions that are greater than the current minimum distance parameter, this position is marked for swapping with a dancer's position. Next in order to eliminate the dancer in the current worst condition, the dancer with the minimum distance from other dancers is selected for transfer to the previously selected non-dancer vertex. Following this position swap, the generic heuristic is used until once again all vertices in S have been flagged as visited for finding an improved position. The process continues until there are no non-dancer positions that have a distance greater than MMD from a dancer's vertex.

4.5.1 Behavioral Perspective

This heuristic approach could be likened to the behavior of a dancer who sees an open spot on the dance floor and immediately goes to the more open area regardless of any dancers in their path. Their local movement is constrained to positions of definite improvement. All of the general behavior constraints remain present, as all of the dancers adhere to the same standard of rules and their movement step sizes remain fixed.

4.5.2 Termination

Termination is proved using a monotonic state function. To prove program termination we define a ranking function as

$$f(S) = \sum_{1 \leq i \leq k} \frac{1}{k^{\mathrm{D}_i(MMD-MAX\{d_{non-dancer}\})}} \ ,$$

where k represents the number of dancers, D_i is the minimum distance from dancing pair i to any of the other k-1 dancing partners, and $\mathrm{MAX}\{d_{non-dancer}\}$ represents the maximum distance for a non-dancing pair position to a dancing pair position in the graph. By the rules of the heuristic, the value D for some vertex i must be increased for each state transition. As D increases for an arbitrary vertex i, the function $f(S)$ decreases. Therefore, $f(S)$ is monotonic and a terminating bound exists, proving termination for the heuristic.

4.6 Complexity

There is at least one exhaustive algorithm for an arbitrary graph, an application of the "clique problem"', finding a clique in a weighted graph that is capable of producing a globally optimal solution to the dancers' placement problem. Finding a clique in a weighted graph will serve as a starting ground for our comparisons of heuristic complexity because it produces a known globally optimal solution for an arbitrary graph but is not a viable solution as it is generally computationally intractable. The subject of complexity for the heuristics considered will be addressed collectively

as there are no closed forms in terms of the graph size and dancer parameters for any of the heuristics.

4.6.1 Clique Problem Application Approach

In the previous section, the claim was made that an application of the clique problem is capable of producing a globally optimal solution for our dancer's position. For this description, a grid graph is used identical to the type used for the descriptions of the dancer's problem. Given an m,n dimensioned graph with k selected vertices (or dancers), map all vertices to vertices in a complete graph with weighted edges representing the distance relationship between each vertex pair. Having constructed this distance graph it is easy to obtain the distance between two vertices of the original graph by looking at the edge weight between the two vertices. Start by finding the edge with the largest weight and eliminate all edges with lesser weights. For example, the maximum distance in a 5 by 5 grid graph is 9. Therefore, starting with 9, we eliminate all edges with weights less than 9. If a clique of size k representing the number of dancers can be found in this filtered distance graph, than the vertices comprising this clique would represent the placement locations of the dancers in the original graph which maintain a minimum distance of 9 from each other. Next, the edge filter is reduced from nine to eight and all of the edges with weight less than eight removed. If a clique of size k is found in this filtered graph then the graph has achieved our minimum distance parameter of eight. This process is repeated until a clique of size k is found in a graph with edges filtered by weight. For smaller graphs, the complexity of such an algorithm is acceptable, but as our problems deal with an

arbitrary number of positions or dancers, the algorithm quickly becomes intractable. Finding a clique of size $\geq$ J has already been shown to be np-complete.

4.6.2 Dancers' Placement Problem Complexity

The complexity of any heuristic is too important to ignore; however, the amount of variance already existing between models of different dimensions and dancer selections within the same graph model class leave the discovery of an actual pattern of iterations less important than the process to obtain it. Also consider that the subtle changes between different heuristics only looks to change the iteration count by some fixed multiple. The chief problem in deriving the complexity of any of the heuristics is determining the set of all terminating states. Within the same graph class, the dimensions and number of represented dancers cause the number of terminating states to vary in a manner which could only be called inconsistent.

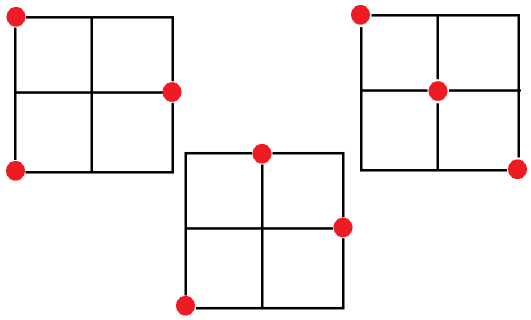

Figure 4.2: General Terminating State Forms using Strict Local Improvement Heuristic on 3x3 Grid with 3 Dancers

Consider a 3x3 grid graph with 3 dancers represented. Figure 4.2 illustrates the general form of the terminating states for the strict local improvement heuristic (rotating the graphs yields different dancer configurations of similar form) on this

graph. If an additional dancer was added to the graph, this would increase the number of terminating states. Adding more dancers might reduce the number of terminating states because the dancers' movements are more restricted. As a result, the lack of consistent terminating states related to these scaleable attributes of the system prevents obtaining a closed form for the complexity of any heuristic. To illustrate complexity of the Strict Local Improvement heuristic, an example is presented where the burden of analyzing terminating states is lessened.

Because the probabilities for dancer movement in the heuristics are only dependent on the previous state, the dancers' placement problem meets the criteria of a Markov Chain. Modeling the complexity of any of the graphs considered for our dancer configurations is very difficult, much less for an arbitrary graph model. There is no way to specify general forms for terminating states in order to establish general forms for probabilities. The first step in complexity analysis for the dancers' placement problem is to exhaustively determine all of the terminating states from the set of dancers' position states. The number of states for a particular problem case is given by the formula:

$$S = \binom{V}{D} \quad ,$$

where S is the number of dancer position states, V is the number of vertices in the graph, and D is the number of represented dancers. So for only a 3×3 grid graph with 3 dancers, there are a total of 84 potential dancer position states, for a 4×4 grid graph with 4 dancers there are 1820 potential dancer position states, and for a 5×5 grid graph with 5 dancers there are 53130 potential dancer position states. Finding

the terminating states of the heuristic amounts to iterating through all of these states and determining if any dancer position movement would be made based on the rules of the heuristic. For illustration purposes, a 2×2 grid graph with 2 dancers has been selected to demonstrate.

For the 2×2 grid graph with 2 dancers, there are a total of 6 dancer positions states:

$$\{\{(1,1),(1,2)\},\{(1,1),(2,1)\},\{(1,1),(2,2)\},\{(1,2),(2,1)\},\{(1,2),(2,2)\},\{(2,1),(2,2)\}\}$$

Following the strict improvement heuristic rules we find the terminating states as

$$\{\{(1,1),(2,2)\},\{(1,2),(2,1)\}\} \ .$$

The next step in the Markov Chain model of complexity analysis is to build a matrix with elements representing the probability of transition from one state to another. This matrix is in the form:

$$P_{m,n} = \begin{pmatrix} p_{1,1} & p_{1,2} & \cdots & p_{1,n} \\ p_{2,1} & p_{2,2} & \cdots & p_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ p_{m,1} & p_{m,2} & \cdots & p_{m,n} \end{pmatrix} \ ,$$

with $p_{i,j}$ representing the probability of transitioning from state i to state j during one dancers' position improvement cycle. Mapping an integer to each state, as in

$$f(n) = \begin{cases} \{(1,1),(1,2)\} & \text{if } n \text{ is } 1 \\ \{(1,1),(2,1)\} & \text{if } n \text{ is } 2 \\ \{(1,1),(2,2)\} & \text{if } n \text{ is } 3 \\ \{(1,2),(2,1)\} & \text{if } n \text{ is } 4 \\ \{(1,2),(2,2)\} & \text{if } n \text{ is } 5 \\ \{(2,1),(2,2)\} & \text{if } n \text{ is } 6 \end{cases},$$

a Markov Chain matrix representing the state's transition probabilities is established:

$$P_{6,6} = \begin{pmatrix} 0 & 0 & 0.5 & 0.5 & 0 & 0 \\ 0 & 0 & 0.5 & 0.5 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0.5 & 0.5 & 0 & 0 \\ 0 & 0 & 0.5 & 0.5 & 0 & 0 \end{pmatrix} .$$

Finally we define the general form of the Markov Chain with p_i representing the probability of being in state i at cycle n,

$$p(n) = \mathbf{p(0)} \begin{pmatrix} p_{1,1} & p_{1,2} & \cdots & p_{1,n} \\ p_{2,1} & p_{2,2} & \cdots & p_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ p_{m,1} & p_{m,2} & \cdots & p_{m,n} \end{pmatrix}^{\mathbf{n}} .$$

For the 2×2 grid graph with 2 dancers, after 1 pair of dancers moves we have

$$p(1) = p(0) \begin{pmatrix} 0 & 0 & 0.5 & 0.5 & 0 & 0 \\ 0 & 0 & 0.5 & 0.5 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0.5 & 0.5 & 0 & 0 \\ 0 & 0 & 0.5 & 0.5 & 0 & 0 \end{pmatrix} = [0, 0, 0.5, 0.5, 0, 0]$$

$$\text{with } \mathbf{p(0)} = [1/6,\ 1/6,\ 1/6,\ 1/6,\ 1/6,\ 1/6] \quad .$$

This assumes a uniformly random initial starting position for the dancers. So for a random starting state in the graph, there is a 50% probability that we will end in either terminating state. Now that we have established probabilities for states

at every cycle of heuristic iteration, we can introduce our indicator random variable (Cormen, Leiserson, Rivest, Stein [6]) as

$$I(j) = \begin{cases} 0 & \text{if state 3 or 4 has been reached at the beginning of cycle j} \\ 1 & \text{if state 3 or 4 has not been reached at the beginning of cycle j} \end{cases} .$$

For average case analysis the **expected value** of $\sum_{j=1}^{n} I(j)$ is used as the average complexity. One might expect that as terminating states or state transitions cannot be expressed in terms of the problems parameters (graph size, graph topology, and the number of dancers), the expected complexity cannot be put in a closed form either. For our purposes we will conclude with the note that the worst case for this particular instance is a mere one cycle, and because $\frac{1}{3}$ of the intial positions are equivalent to a terminating state, the average computation complexity is $\frac{2}{3}$ a cycle.

In conclusion, a general closed form for the computational complexity was not found in terms of the system parameters (graph class, graph dimensions, and number of elements) and analysis should be performed on a case-by-case basis.

CHAPTER 5

ANALYSIS

For analysis of the heuristics, Monte Carlo simulations are run 1000 times for each graph class and heuristic using uniformly random starting positions. The simulation results for each graph and dimension parameters are tabularized and the heuristics are compared. Several sizes of the proposed test bed cases are used to exercise the heuristics and provide analysis for. In addition, a non-ideal graph dimension of each graph class is used for comparison to the test bed cases of the different graph classes. Certain heuristics may perform better given a larger space for movement while the upper bound of the graph parameter MMD does not increase.

5.1 Graph Generation

The graphs' dimensions for the various classes considered in the research are not selected at random. Several forms with known globally optimal parameter solutions are selected as test bed cases as they provide a basis for the evaluating the heuristic results. The different types of graphs used for research did not have a general closed form which could be used to generate optimal parameter bounds. Instead, each graph type parameter bound is considered individually. The different closed

forms of globally optimal parameter bounds are defined in the immediate subsections which follow.

5.1.1 Path: Closed Form Bound

Starting with the path graph, the maximum MMD is proved for a specific path length and population of dancers. Let P_n be a path of n vertices and S be a subset of $V(P_n) \ni |S| = k$. Consider the variable $t = \frac{n-1}{k-1}$, selected because the total length of the path is given by $n-1$ and the number of non-overlapping segments given that a dancer is at each endpoint is $k-1$. Assume that $MAX\{MMD(P_n, k)\} > t$. Therefore $\forall x, y \in S \mid x \neq y \vdash dist(x, y) > t$. This implies that the

$$\sum_{k-1} \text{non-overlapping segments} > t(k-1) = n-1$$

a contradiction because the length of the entire path is only $n-1$. Therefore

$$MAX\{MMD(P_n, k)\} \leq \frac{n-1}{k-1} \ .$$

Equality can be achieved when $(k-1) \mid (n-1)$ if equivalent segment distances are used between the dancers' positions.

5.1.2 Cycle: Closed Form Bound

The closed form proof of the cycle is found in a similar way to the path. Let C_n be a cycle of n vertices and S be a subset of $V(C_n) \ni |S| = k$. Consider the variable $t = \frac{n}{k}$, we assume that $MAX\{MMD(C_n, k)\} > t$. Therefore $\forall x, y \in S \mid x \neq$

$y \vdash dist(x, y) > t$. There are k non-overlapping path segments to complete the cycle from the any vertex in S back to itself. This implies that

$$\sum_{k} \text{non-overlapping segments} > t(k) = n$$

a contradiction because the length of the entire cycle is only n. Therefore

$$MAX\{MMD(C_n, k)\} \leq \frac{n}{k} \ .$$

Equality can be achieved when $k \mid n$ if equivalent segment distances are used between the dancers' positions.

5.1.3 Torus: Closed Form Bound

A slightly more complex closed form is seen for the torus. Let G be a graph such that the vertices are drawn with Manhattan metric distances proportional to one unit from any adjacent vertices. Now consider closed balls of radius r in a Manhattan space. If k balls of radius r centered on vertices of a graph G are placed on graph G such that the closed balls are non-overlapping then we have achieved a minimum distance of $2r$ for k vertices acting as the center points of the closed balls. If k non-overlapping closed balls of radius r are selected such that r cannot be increased without causing the closed balls to overlap, then $2r$ is the maximum minimum distance between k vertices of the graph G. As a result, maximizing the total summed area of the non-overlapping balls (number of balls) contained on graph

G maximizes the number of vertices which can be contained at a minimum distance of $2r$.

Let $\mathrm{T}_{m\times n}$ be a torus of $m \times n$ vertices $\ni m, n \in \mathbb{N}$ and $2r \mid m, n$. What is the maximum area absorbable into such a torus?

While the Manhattan space is used for establishing the geometric form of the ball boundaries placed within our graphs, the Euclidean space will be used to determine the notion of area. Therefore, given that the length of any side of the square representing a ball in Manhattan space of radius r is $\sqrt{2}r$ in Euclidean space, the area of the ball is $(\sqrt{2}r)^2 = 2r^2$. The maximum area absorbable into a torus is equal to the area of the torus surface, $(m)(n)$ (See Figure 5.1). Therefore, if the

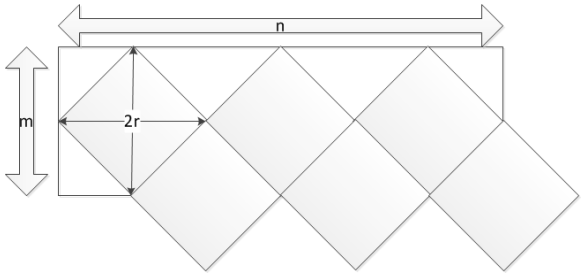

Figure 5.1: Maximum area contained by torus

radius parameter r can be selected such that the sum of k non-overlapping balls of radius r is equal to the area of the torus, $mn = k2r^2$, as no more surface area is available to increase size of radius r, this acts as an upper bound, and the parameter MMD of graph $\mathrm{T}_{m\times n}$ is $2r$. The equivalent mathematical model for this is

$$mn = k2r^2 \implies MAX\{MMD(T_{m\times n}, k)\} = 2r \ ,$$

with m,$n \ni m, n \in \mathbb{N}$ and $2r \mid m, n$ representing the dimensions of the graph and k representing the number of dancers.

5.1.4 Grid: Closed Form Bound

Similar to area of torus we consider k non-overlapping balls of a Manhattan metric that can be contained on a $(m+1) \times (n+1)$ grid graph G denoted $G_{(m+1)\times(n+1)}$ where $2r \mid m, n$. The number of non-overlapping balls contained by graph G is slightly more complex than the number of balls contained on a torus.

First we note that because the grid is not wrapped like the torus, the area of a ball of radius r can "extend" beyond the bounds of the outer edges of the graph, therefore the weaker bound of the number of balls contained on G can be determined by extending the length of G by r in every direction. The area contained inside this grid G' is $(m+2r)(n+2r)$. Therefore the maximum number of balls containable internal to the extended graph G' $= k \leq \frac{(m+2r)(n+2r)}{2r^2}$. As the balls of radius r in the original graph G cannot extend beyond the area internal to G', observe

$$MAX\{MMD((G,k)\} \leq \sqrt{\frac{2(m+2r)(n+2r)}{k}} \ .$$

Now we consider the maximum area of the number of balls radius r we can contain on the original grid G. The maximum containable area is equal to the coverage of the maximum internal area of the graph $(m)(n)$ and the maximum area external to the graph.

5.1.4.1 External Area

The area external to the graph can be found by summing the areas of the balls of radius r external to the graph formed by the intersection of the non-overlapping balls of radius r with the external boundaries of the graph (See Figure 5.2). Let the

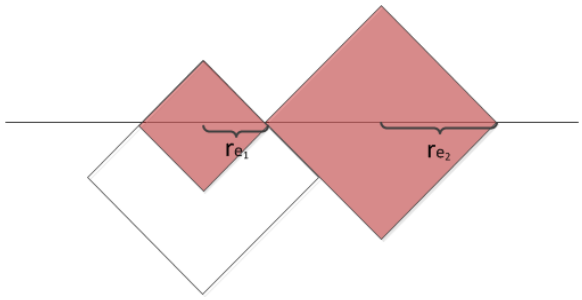

Figure 5.2: Balls formed by using boundary segment as diameter

areas of the newly formed balls external to the graph formed by the intersection of the graph external boundaries with non-overlapping balls of radius r be denoted by e_i. Since half the area of e_i is external to the graph, the sum of the area external to the graph is

$$\sum \frac{2r_{e_i}^2}{2} = \sum r_{e_i}^2 \ .$$

Now consider two arbitrary boundary balls e_j, e_k such that $0 < r_{e_j} \leq r_{e_k} \leq r$. Can their total area be increased by increasing r_{e_j} by Δ and decreasing r_{e_k} by Δ? Based on the equation

$$[r_{e_j} + \Delta]^2 + [r_{e_k} - \Delta]^2 - [r_{e_j}^2 + r_{e_k}^2] > 0 \equiv 2\Delta^2 + 2\Delta(r_{e_j} - r_{e_k}) > 0 \ ,$$

the area is greatest when $r_{e_j} = r_{e_k}$. Therefore the maximum area external to the graph occurs when r_{e_j} is r.

Consider a boundary of length l. The maximum area external to a boundary is kr^2 where k is the number of balls external to the boundary and $k2r \leq l+2r$ because the length of any ball cannot extend beyond the boundaries by r in either direction. It is easy to see that r is maximum when $k2r = l + 2r$ and the only configuration that allows for this is when there are balls centered on the corner of the grid and l is divisible by $2r$.

Then, the maximum area external to grid is the sum of the maximal areas external to each side of the grid minus the overlap caused by the corners, which is equivalent to a quarter of the area of a ball of radius r (See Figure 5.3). Therefore

$$\text{Maximum external area} = r(m + n + 2r) \ .$$

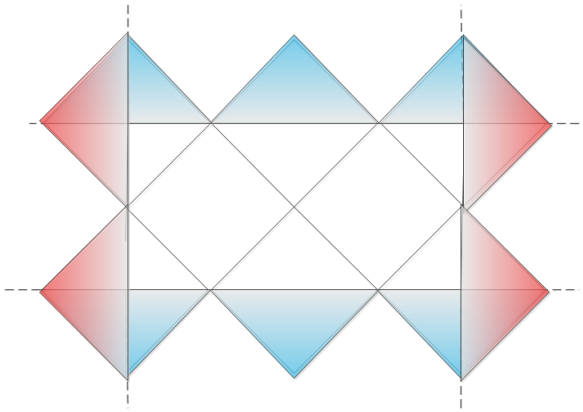

Figure 5.3: Color areas external to graph

5.1.4.2 Total Area

This yields the maximum area of the grid as the sum of the internal and external areas, $mn + (m + n + 2r)r = kr^2$, leading to the implication

$$mn + (m + n + 2r)r = kr^2 \implies MAX\{MMD(G_{(m+1)\times(n+1)}, k)\} = 2r \ ,$$

with m,$n \ni m, n \in \mathbb{N}$ and $2r \mid m, n$ representing the dimensions of the graph and k representing the number of dancers.

5.2 Test Bed Cases

Based on the boundary cases discussed in the previous section, several test bed cases are introduced and illustrated. Using the formulas from the preceding section, the test bed cases are selected for graphs with a known globally optimal MMD (maximal minimum distance between any pair of dancers) parameter bound. The attributes of these graphs are determined by first selecting the desired MMD parameter bound and the number of dancers. Then, the dimensions of the graph are chosen to solve the formulas for equality with the selected MMD bound. The illustrations meeting the boundary conditions for each graph class are smaller dimensions then those used for simulation results, but the patterns persist to larger scales and asymmetric graph dimensions. The graphs are provided as a visual aid to compare to the heuristic generated graphs in a subsequent section.

5.2.1 Path

Using the formula

$$MAX\{MMD(P_n, k)\} = \frac{n-1}{k-1} \ ,$$

a path of 21 (n=21) illustrates the globally optimal configuration for a path with 5 dancers (k=5) on it (see Figure 5.4). For analysis, the parameter MMD was selected as 4, 6, and 8 with a fixed number k=10 dancers which translates to graphs P_{37}, P_{55}, and P_{73}. In addition, an intermediate graph, P_{60}, to the P_{55} and P_{73} cases is considered to compare the resulting frequency distributions with the P_{55} frequency distribution results to observe the improvement due to added dancer positions while remaining limited to the optimal P_{55} MMD bound.

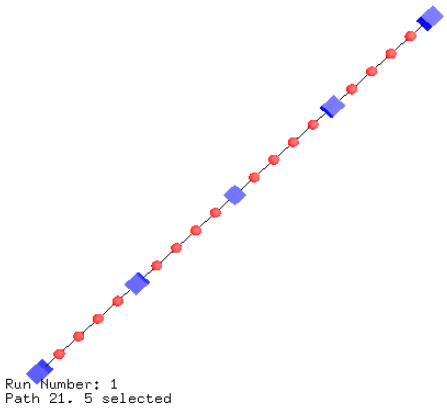

Figure 5.4: Test Bed Case: Path Length: 21, Dancers: 5, MMD = 4

5.2.2 Cycle

Using the formula

$$MAX\{MMD(C_n, k)\} = \frac{n}{k} \ ,$$

a cycle of 25 (n=25) illustrates the globally optimal configuration for a cycle with 5 dancers (k=5) on it (see Figure 5.5). For analysis, the parameter MMD was selected as 4, 6, and 8 with a fixed number k=10 dancers which translates to graphs C_{40}, C_{60}, and C_{80}. In addition, an intermediate graph, C_{65}, to the C_{60} and C_{80} cases is considered to compare the resulting frequency distributions with the C_{60} frequency distribution results to observe the improvement due to added dancer positions while remaining limited to the optimal C_{60} MMD bound.

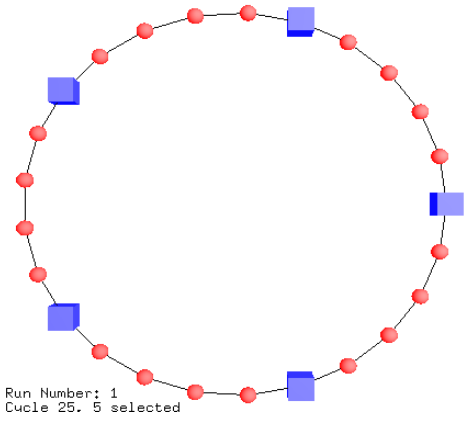

Figure 5.5: Test Bed Case: Cycle Length: 25, Dancers: 5, MMD = 5

5.2.3 Grid

Using the equation

$$mn + (m + n + 2r)r = kr^2 \text{ with } 2r|m, n$$

implies

$$MAX\{MMD(G_{(m+1)\times(n+1)}, k)\} = 2r \quad .$$

A 9x9 grid (m,n = 9,9) illustrates the globally optimal configuration for a grid with 13 dancers (k=13) on it (see Figure 5.6). For analysis the parameter MMD will be selected as 4, 6, and 10 with a fixed number k=13 dancers which translates to graphs $G_{9\times9}$, $G_{13\times13}$, and $G_{21\times21}$. In addition, an intermediate graph, $G_{14\times14}$, to the $G_{13\times13}$ and $G_{21\times21}$ cases is considered to compare the resulting frequency distributions with the $G_{13\times13}$ frequency distribution results to observe the improvement due to added dancer positions while remaining limited to the optimal $G_{13\times13}$ MMD bound.

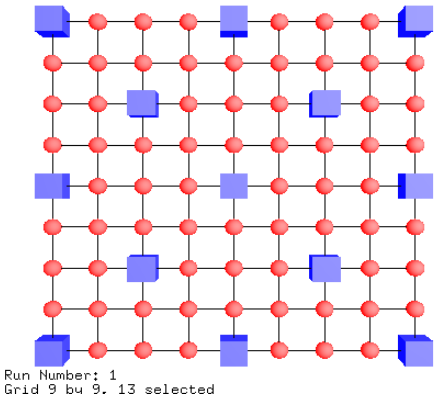

Figure 5.6: Test Bed Case: 9x9 Grid, Dancers: 13, MMD = 4

5.2.4 Torus

Using the equation $mn = k2r^2$ with $2r|m, n$ implies

$$MAX\{MMD(T_{m\times n}, k)\} = 2r \ .$$

For an 8x8 torus (m,n = 8,8), a configuration is in the form of Figure 5.7, but this figure is difficult to view projected onto a 2-dimensional image so the object is flattened into the grid form of Figure 5.8. For this flattened torus, consider the top and bottom boundaries as adjacent and the left and right boundaries as adjacent. For analysis, the parameter MMD will be selected as 4, 6, and 8 with a fixed number k=12 dancers which translates to graphs $T_{8\times12}$, $T_{12\times18}$, and $T_{16\times24}$. In addition, an intermediate graph, $T_{13\times19}$, to the $T_{12\times18}$ and $T_{16\times24}$ cases is considered to compare the resulting frequency distributions with the $T_{12\times18}$ frequency distribution results to observe the improvement due to added dancer positions while remaining limited to the optimal $T_{12\times18}$ MMD bound.

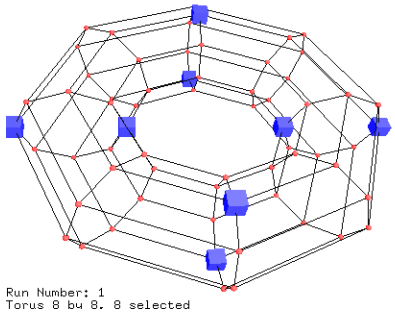

Figure 5.7: Test Bed Case: 8x8 Torus, Dancers: 8, MMD = 4

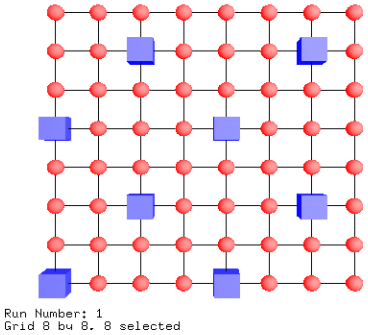

Figure 5.8: Test Bed Case: 8x8 Torus Unwrapped, Dancers: 8, MMD = 6

5.3 Path Analysis

The first Monte Carlo run is for P_{37} with 10 dancer vertices. Because the path runs in a straight line, it is difficult to provide illustrations capturing the length of the path and the position of the dancers. For this reason, worst case path configurations will not be given for every heuristic. Instead a poor dancer placement example is given for a shorter path (Figure 5.9) from a previous simulation. It is easy to see that the cause of such positioning is the result of incrementing or decrementing segment lengths between consecutive dancer vertices in the graph.

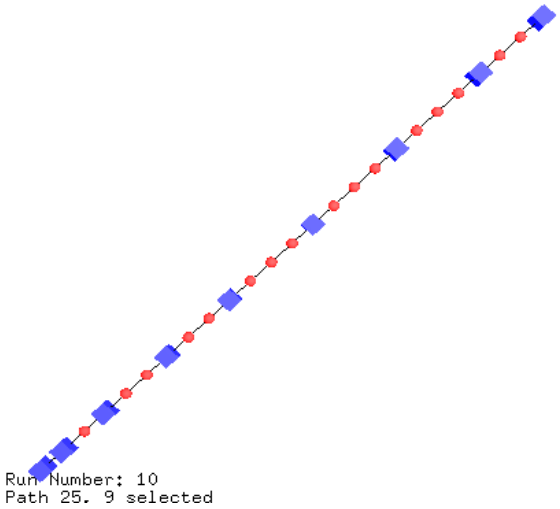

Figure 5.9: P_{25}, Strict Local Improvement: Poor Dancer Positions

5.3.1 P_{37}

Table 5.1 shows the Monte Carlo results for P_{37} with 10 dancer vertices. The distribution of the P_{37} has similar results for all of the heuristics. The distribution between heuristics is not radically different, but the Reduced Step heuristic manages

to avoid having any terminating states with a MMD parameter of 2, and this was after doubling the size of the graph only once. Continued iteration may yield even better results. Figure 5.10 offers a side by side comparison of the heuristic results.

Table 5.1: P_{37} with 10 Dancers, Monte Carlo Results

Graph P_{37}	Min. Distance			
	1	2	3	4
Local Improvement	34	415	550	1
Weighted Equivalent	0	266	606	128
Simulated Annealing	27	328	645	0
Reduced Step	0	0	903	97
Move To Open Area	0	252	640	108

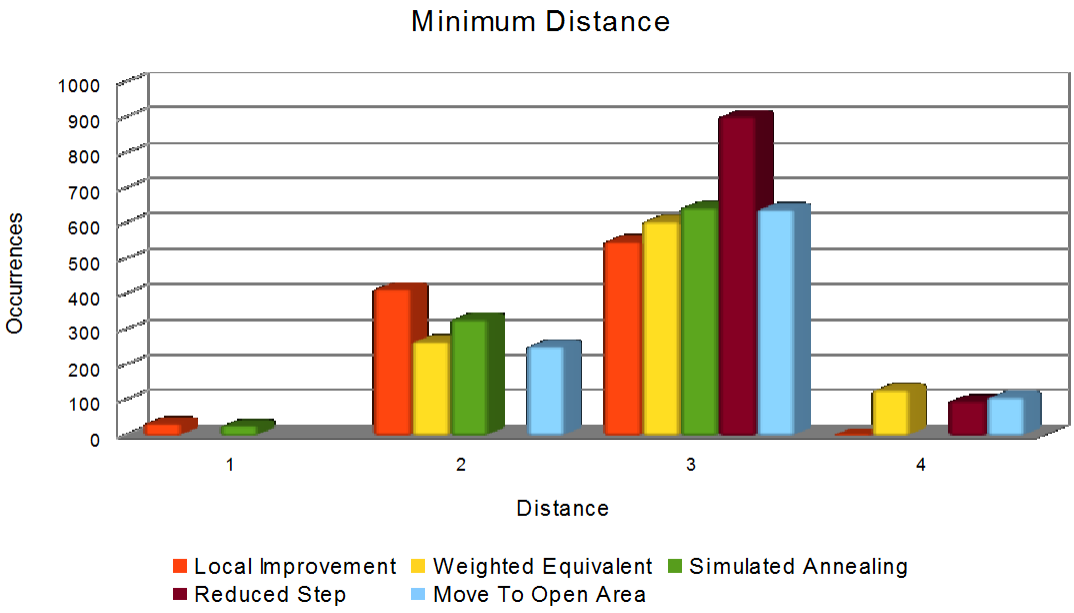

Figure 5.10: P_{37} Frequency Distribution Graph

5.3.2 P_{55}

Table 5.2 shows the Monte Carlo results for P_{55} with 10 dancer vertices. Increasing the radius of the perfect packing causes the MMD distribution to be more spread out. Again the Reduced Step heuristic maintains a consistent near optimal distance. Figure 5.11 illustrates these heuristic values side by side. Notice that while the Weighted Equivalent and Move to Open Area heuristics never yield a "worst case" distance configuration, the statistical distributions for all of the heuristics excluding Reduced Step are very similar.

Table 5.2: P_{55} with 10 Dancers, Monte Carlo Results

Graph	Min. Distance				
P_{55}	2	3	4	5	6
Local Improvement	29	99	510	360	2
Weighted Equivalent	0	0	602	315	83
Simulated Annealing	17	94	545	342	2
Reduced Step	0	0	0	915	85
Move To Open Area	0	0	621	298	81

5.3.3 P_{60}

Table 5.3 shows the Monte Carlo results for P_{60} with 10 dancer vertices. The addition of movement space with the optimal MMD bound fixed allowed for all of the heuristics to show improvement and to grow closer to the previously dominant Reduced Step heuristic's distribution. It was not expected that the Strict Local Improvement and Simulated Annealing would have a higher frequency of MMD parameters of 5 than the Weighted Equivalent and Move To Open Area heuristic. If

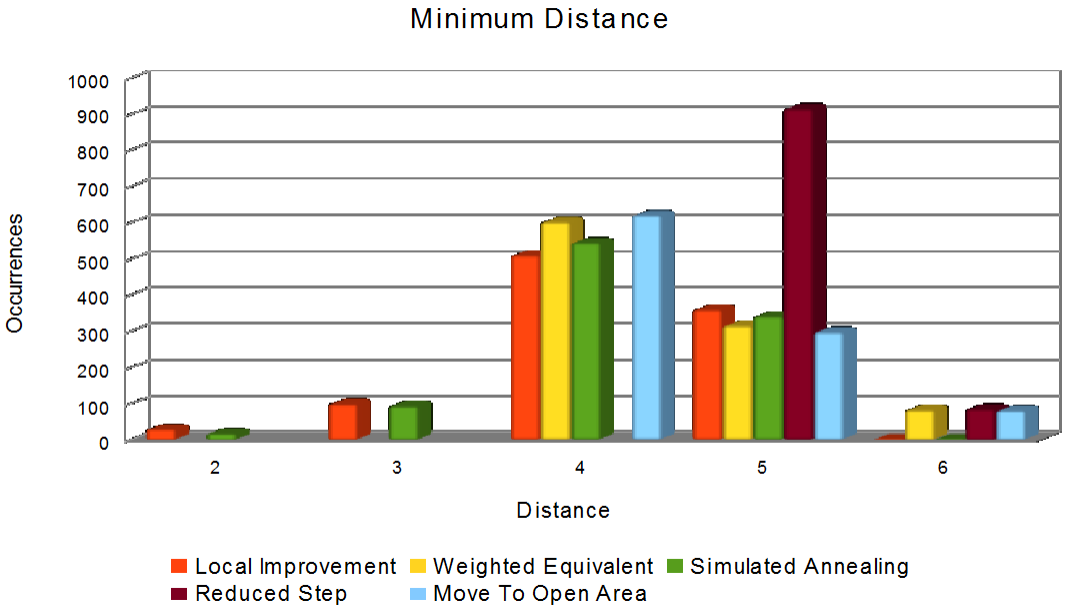

Figure 5.11: P_{55} Frequency Distribution Graph

one observes this P_{60} distribution and the previous P_{55} distribution, it is seen that the majority of the difference between these two pairs' distributions is often found in the extremes of the MMD frequencies; the remaining differences are attributable to random initial starting positions and decision-making. Figure 5.12 illustrates these heuristic values side-by-side.

Table 5.3: P_{60} with 10 Dancers, Monte Carlo Results

Graph	Min. Distance			
P_{60}	3	4	5	6
Local Improvement	86	216	644	54
Weighted Equivalent	0	413	418	169
Simulated Annealing	72	248	632	48
Reduced Step	0	0	805	195
Move To Open Area	0	431	416	153

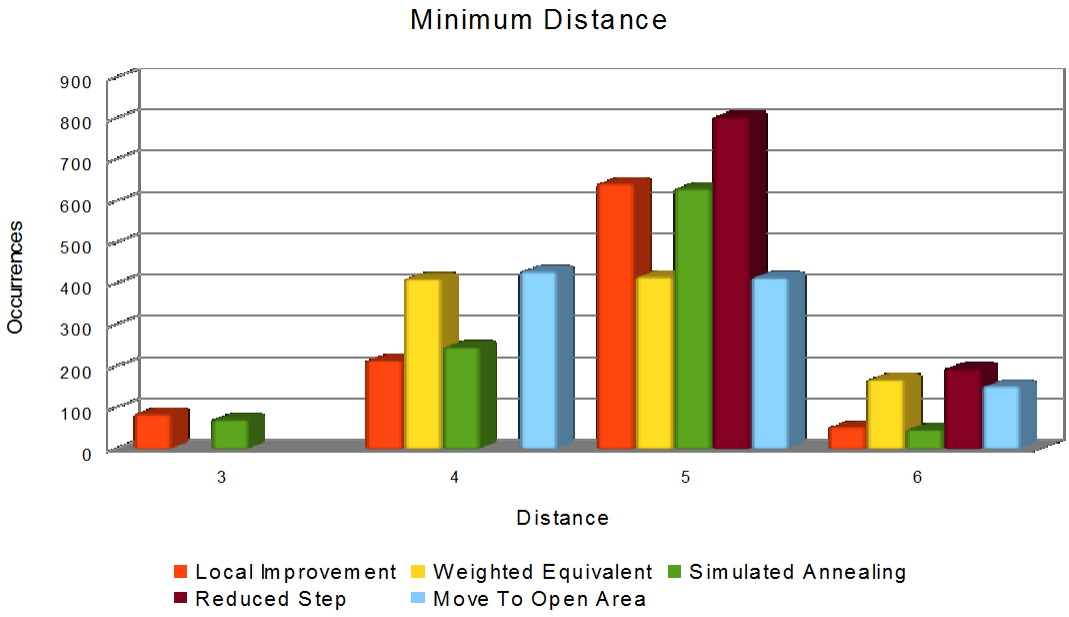

Figure 5.12: P_{60} Frequency Distribution Graph

5.3.4 P_{73}

Table 5.4 shows the Monte Carlo results for P_{73} with 10 dancer vertices. While the Reduced Step heuristic continues to dominate the optimal solution distribution, the Weighted Equivalent and Move to Open Area heuristic begin to separate themselves from the Strict Local Improvement and Simulated Annealing heuristics in their optimal solution distributions. Figure 5.13 illustrates these heuristic values side by side.

5.4 Cycle Analysis

Next, Monte Carlo simulations are performed for cycles beginning with a C_{40} with 10 dancer vertices. As cycles continue the same trends as paths for poor con-

Table 5.4: P_{73} with 10 Dancers, Monte Carlo Results

Graph P_{73}	Min. Distance 4	5	6	7	8
Local Improvement	53	184	535	228	0
Weighted Equivalent	0	0	579	349	72
Simulated Annealing	77	189	499	235	0
Reduced Step	0	0	0	918	82
Move To Open Area	0	0	572	364	64

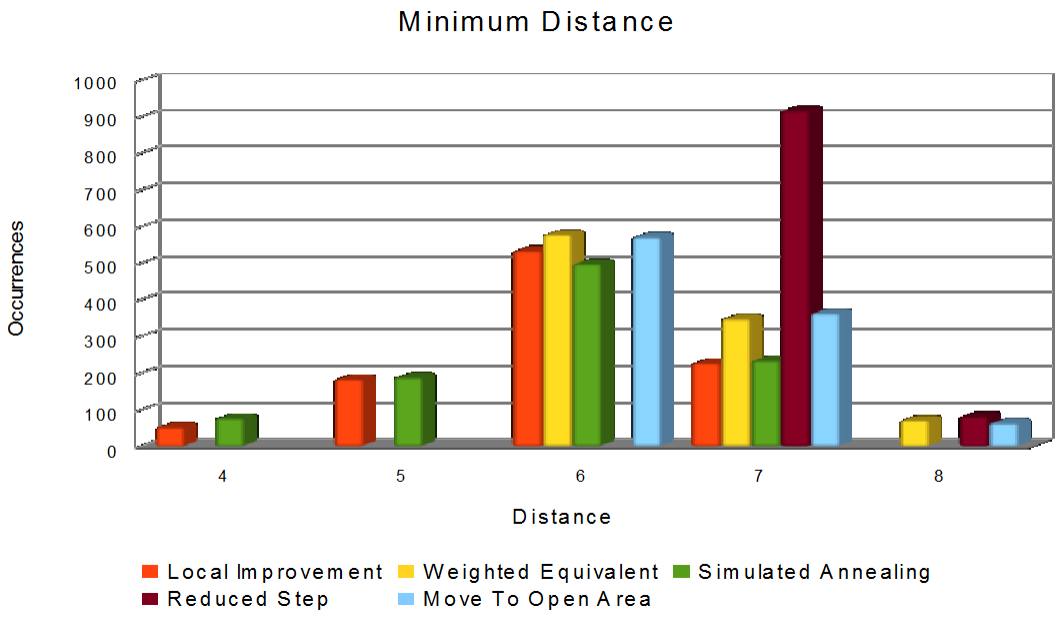

Figure 5.13: P_{73} Frequency Distribution Graph

figurations, a worst case cycle configuration is given to illustrate the incrementing and decrementing segment lengths between consecutive dancer vertices in the graph. Figure 5.14 presents a cycle with a poor cycle configuration.

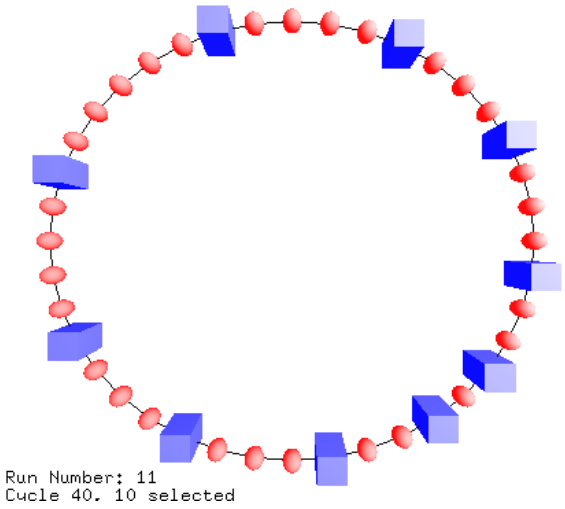

Figure 5.14: C_{40}, Strict Local Improvement: Poor Dancer Positions

5.4.1 C_{40}

Table 5.5 shows the Monte Carlo results for C_{40} with 10 dancer vertices. Here there is a clear separation between the Strict Local Improvement and Simulated Annealing heuristics as compared to the others. While the worst possible distance parameter achievable by a graph (a distance of 1) has been eliminated, the trends of the Strict Local Improvement and Simulated Annealing heuristics continue to be sub-optimal, while the Weighted Equivalent, Reduced Step, and Move to Open Area heuristics produce optimal solutions a clear majority of the time. Once again, the Reduced Step heuristic is marginally better than its competitors, but more significantly, it produced an optimal solution for every run of the simulation. Figure 5.15 offers a side by side comparison of the heuristic results.

Table 5.5: C_{40} with 10 Dancers, Monte Carlo Results

Graph C_{40}	Min. Distance 2	3	4
Local Improvement	351	648	1
Weighted Equivalent	0	63	937
Simulated Annealing	321	679	0
Reduced Step	0	0	1000
Move To Open Area	0	55	945

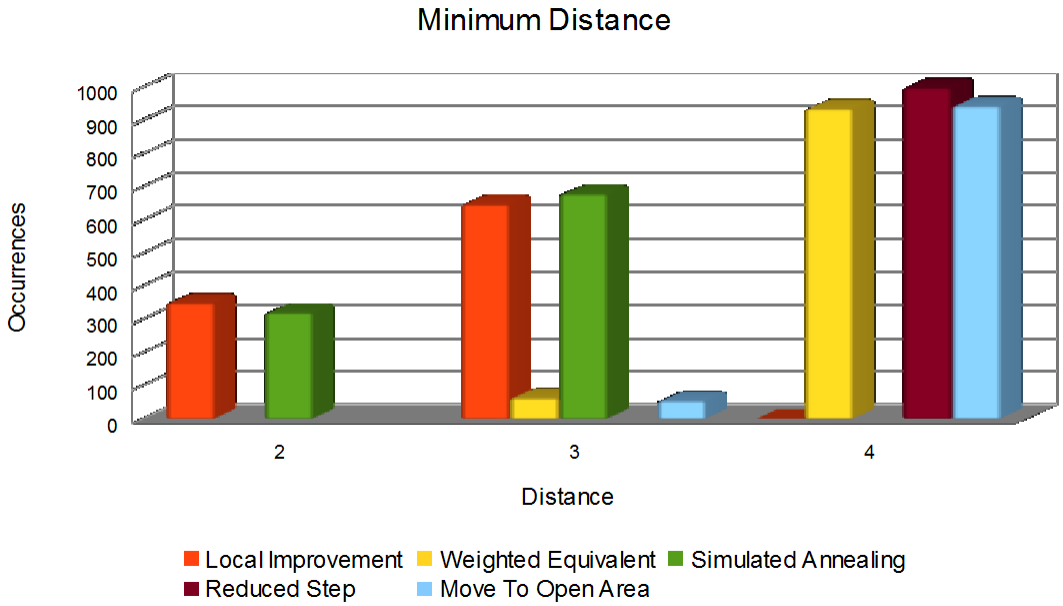

Figure 5.15: C_{40} Frequency Distribution Graph

5.4.2 C_{60}

Table 5.6 shows the Monte Carlo results for C_{60} with 10 dancer vertices. One trend to notice is the limit to the distance between the MMD for the Strict Local Improvement and Simulated Annealing heuristics and the optimal MMD. Figure 5.16 offers a side-by-side comparison of the heuristic results.

Table 5.6: C_{60} with 10 Dancers, Monte Carlo Results

Graph C_{60}	Min. Distance 4	5	6
Local Improvement	587	413	0
Weighted Equivalent	0	40	960
Simulated Annealing	599	400	1
Reduced Step	0	0	1000
Move To Open Area	0	47	953

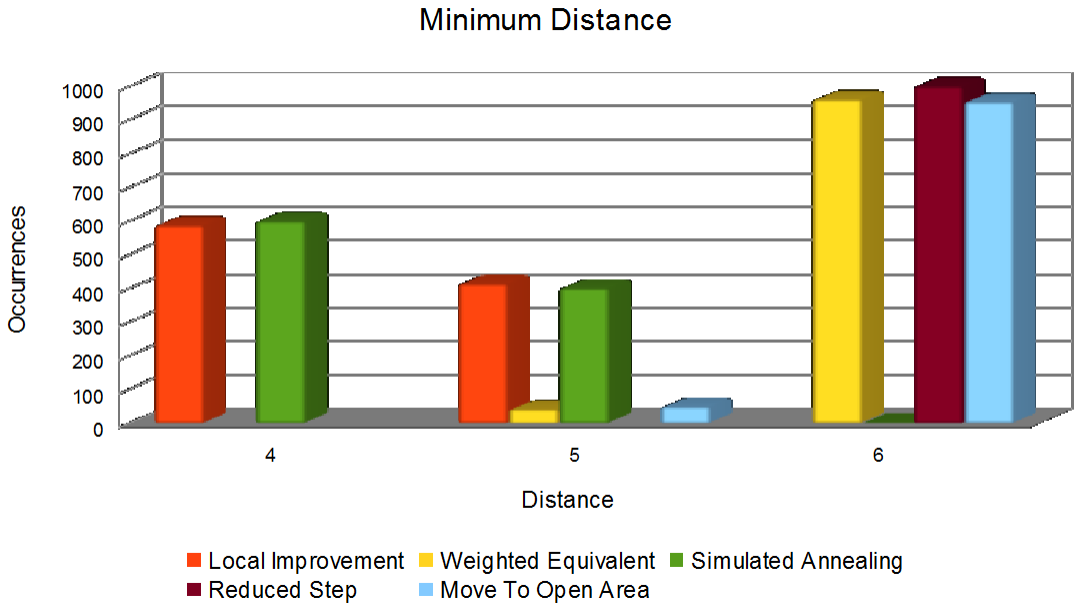

Figure 5.16: C_{60} Frequency Distribution Graph

5.4.3 C_{65}

Table 5.7 shows the Monte Carlo results for C_{65} with 10 dancer vertices. The addition of movement space with the optimal MMD bound fixed allows for the improvement of the Strict Local Improvement and Simulated Annealing heuristic and pushes all the MMD distributions of the remaining heuristics to that of the optimal MMD bound. Figure 5.17 offers a side-by-side comparison of the heuristic results.

Table 5.7: C_{65} with 10 Dancers, Monte Carlo Results

Graph C_{65}	Min. Distance 4	5	6
Local Improvement	122	804	74
Weighted Equivalent	0	0	1000
Simulated Annealing	110	797	93
Reduced Step	0	0	1000
Move To Open Area	0	0	1000

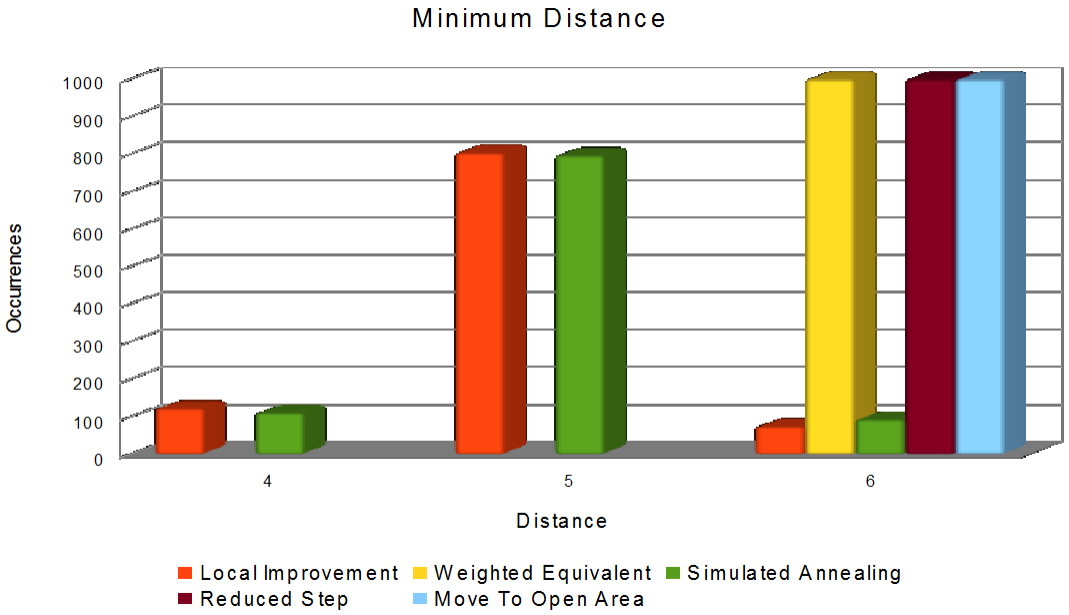

Figure 5.17: C_{65} Frequency Distribution Graph

5.4.4 C_{80}

Table 5.8 shows the Monte Carlo results for C_{80} with 10 dancer vertices. The Strict Local Improvement and Simulated Annealing heuristics continue to follow the trends of the preceding cycle lengths being limited to MMD parameters within 2 of the optimal MMD bound, and the progression of increasing the distribution of poor configurations for these heuristics continues (see the ratio of worst to best occurrences

for these heuristics from C_{40} to C_{80}). The Reduced Step heuristic again produces the best results of analyzed heuristics. Figure 5.18 offers a side-by-side comparison of the heuristic results.

Table 5.8: C_{80} with 10 Dancers, Monte Carlo Results

Graph C_{80}	Min. Distance 6	7	8
Local Improvement	726	274	0
Weighted Equivalent	0	35	965
Simulated Annealing	733	267	0
Reduced Step	0	0	1000
Move To Open Area	0	50	950

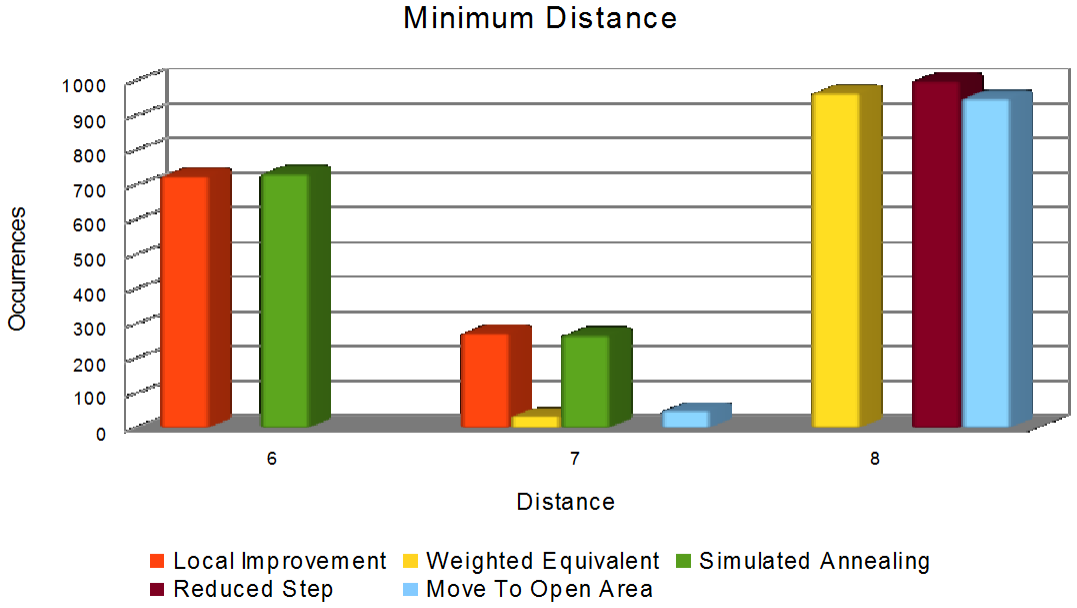

Figure 5.18: C_{80} Frequency Distribution Graph

5.5 Grid Analysis

The grid was the first graph considered for evaluating dancer positions; it offers the first glimpse of increasing graph complexity and presumably provides much better insight into the effectiveness of the heuristics as it is trivial to select an optimal dancer placement for both the path and cycle graphs. In addition, the graph provides the first look at a 2-dimensional space which could represent a surface and abstractly very closely resembles the sphere packing problem.

5.5.1 $G_{9\times9}$

The first grid graph analyzed is a $G_{9\times9}$ grid with 13 dancer vertices. For this grid almost all of the solutions were non-optimal. Grids are susceptible to clustering in the corners and aligning themselves along the diagonal. Consider Figure 5.19 and the Strict Local Improvement heuristic. This scenario was undoubtedly created by a poor initial position for the dancers. It is easily seen that it is the restriction of the dancers to only consider a definite increase in minimum distance as an improvement that contributes to this poor solution. This is not limited to the Strict Local Improvement heuristic however. Figure 5.20 illustrates a situation that the Weighted Equivalent heuristic cannot improve.

Table 5.9 shows the distributions of the MMD parameter for the different heuristics. The heuristics for the $G_{9\times9}$ approximately form 3 different distributions. The poorly performing heuristics are the Strict Local Improvement and Simulated Annealing heuristics failing to produce solutions which are greater than half the opti-

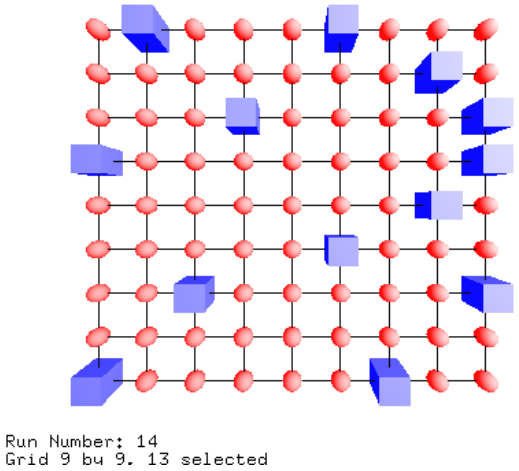

Figure 5.19: $G_{9\times9}$, Strict Local Improvement: Poor Dancer Positions

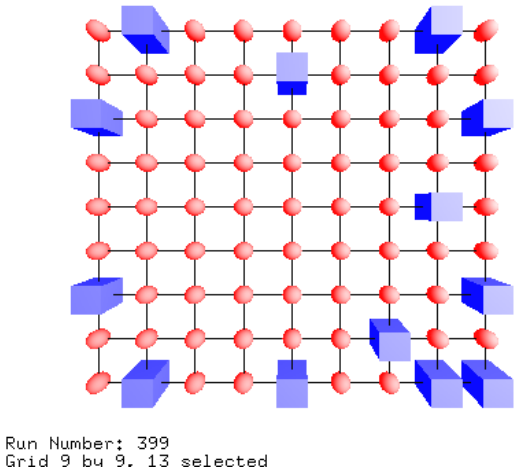

Figure 5.20: $G_{9\times9}$, Weighted Equivalent: Poor Dancer Positions

mal minimum distance the majority of the time. The Weighted Equivalent heuristic is very nearly evenly divided on its distribution of minimum distances. The Reduced Step and Move to Open Area heuristics form the final distribution class, producing a near optimal solution the majority of the time. There is a very low number of optimal solutions produced by any of the heuristics. Figure 5.21 offers a side-by-side comparison of the heuristic results.

Table 5.9: $G_{9\times9}$ with 13 Dancers, Monte Carlo Results

Graph	Min. Distance			
$G_{9\times9}$	1	2	3	4
Local Improvement	6	845	149	0
Weighted Equivalent	1	492	507	0
Simulated Annealing	1	755	244	0
Reduced Step	0	28	972	0
Move To Open Area	0	16	983	1

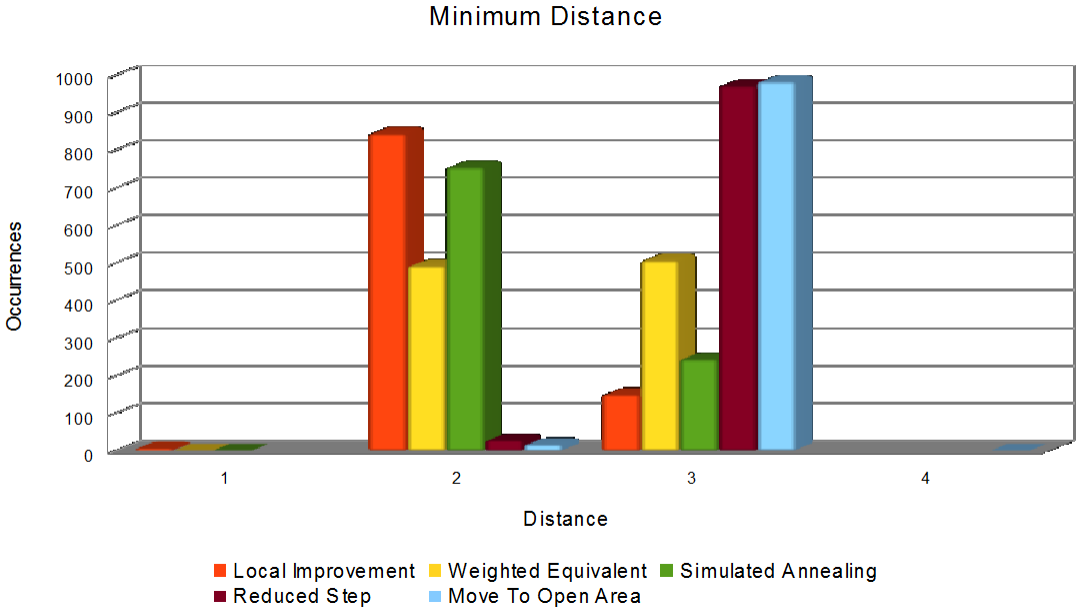

Figure 5.21: P_9xP_9 Frequency Distribution Graph

5.5.2 $G_{13\times13}$

The next grid graph analyzed is a $G_{13\times13}$ grid with 13 dancer vertices. The distribution for this graph marks a clear change in the distribution of the solutions for different heuristics. The absence of the worst case distance (minimum distance of 1) for the Strict Local Improvement heuristic run on a larger graph is likely because of the greater initial position selection space and not because such a case cannot occur. Nonetheless, poor configurations still exist in the form of Figure 5.22. The dancers have loosely aligned themselves on several diagonals present in the graph.

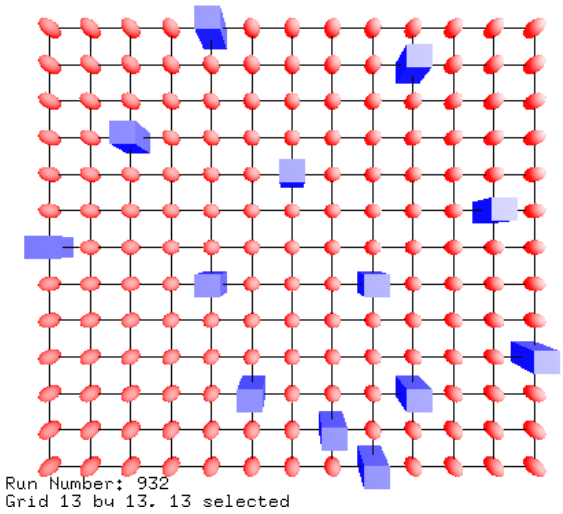

Figure 5.22: $G_{13\times13}$, Strict Local Improvement: Poor Dancer Positions

Table 5.10 shows the distributions of the minimum distance for the different heuristics. The heuristic distributions for the $G_{13\times13}$ grid breaks most of the patterns which were seen in all previous distributions. The Reduced Step heuristic continues to provide the most reliable dancer configurations of the heuristics. Figure 5.23 offers a side-by-side comparison of the heuristic results.

Table 5.10: $G_{13\times13}$ with 13 Dancers, Monte Carlo Results

Graph $G_{13\times13}$	Min. Distance 2	3	4	5	6
Local Improvement	14	371	609	6	0
Weighted Equivalent	1	67	837	95	0
Simulated Annealing	5	271	710	14	0
Reduced Step	0	6	352	636	6
Move To Open Area	0	0	674	326	0

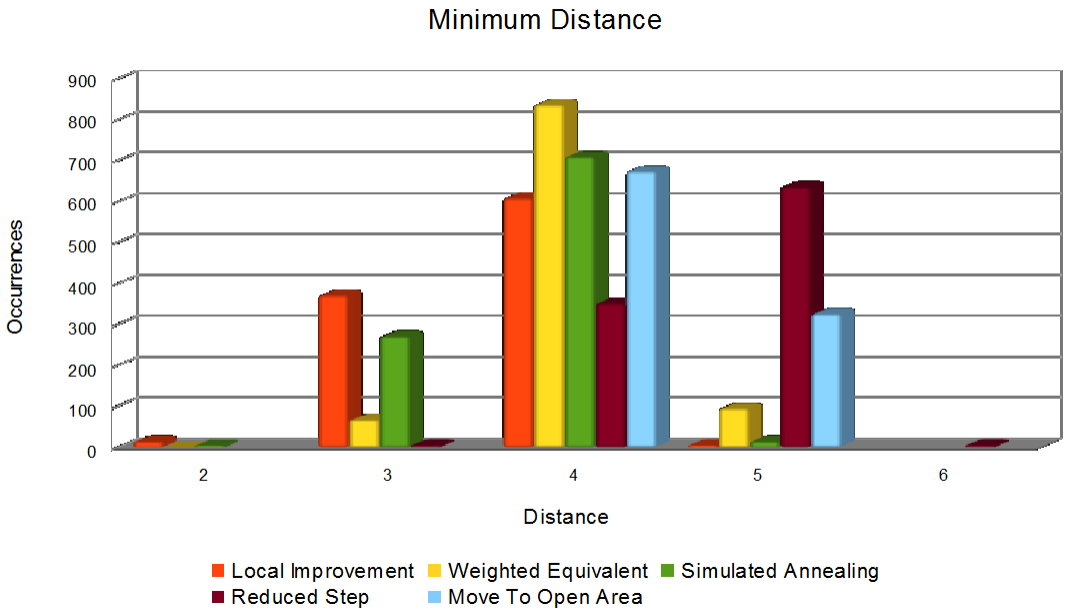

Figure 5.23: $G_{13\times13}$ Frequency Distribution Graph

5.5.3 $G_{14\times14}$

Analysis of the $G_{14\times14}$ grid with 13 dancer vertices provides more space for the dancers to configure themselves to meet the $G_{13\times13}$ MMD bound. Table 5.11 shows the distributions of the minimum distance for the different heuristics. This amount of additional space did very little to improve the distributions of the low achievement heuristics (Strict Local Improvement and Weighted Equivalent) but significantly im-

proved the distributions of the other heuristics. Figure 5.24 offers a side-by-side comparison of the heuristic results.

Table 5.11: $G_{14\times14}$ with 13 Dancers, Monte Carlo Results

Graph $G_{14\times14}$	Min. Distance				
	2	3	4	5	6
Local Improvement	3	164	748	85	0
Weighted Equivalent	1	21	451	525	2
Simulated Annealing	0	76	768	156	0
Reduced Step	0	4	58	900	38
Move To Open Area	0	0	59	934	7

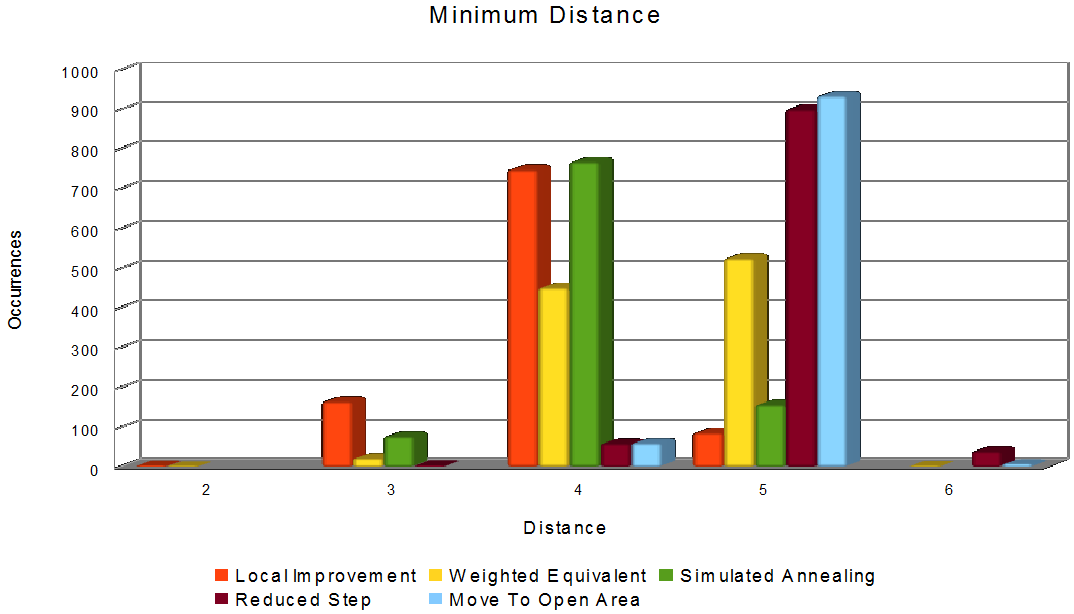

Figure 5.24: $G_{14\times14}$ Frequency Distribution Graph

5.5.4 $G_{21\times21}$

Finally, the grid graphs are concluded with $G_{21\times21}$ with 13 dancer vertices. The worst case MMD parameter for any of the heuristics continued to climb with the

graph size. A case of the worst MMD found during the Monte Carlo runs is shown in Figure 5.25. Here one can see that there is very little clustering found in the corners, but the alignment of the dancers on a diagonal restricts movement to positions above.

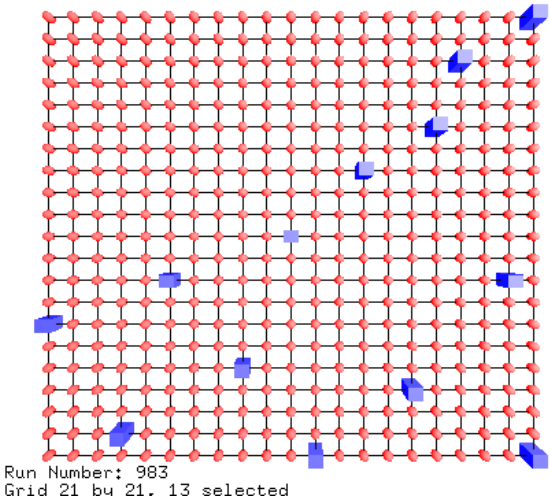

Figure 5.25: $G_{21\times21}$, Strict Local Improvement: Poor Dancer Positions

Table 5.12 shows the distributions of the minimum distance for the different heuristics. The heuristic distributions for the $G_{21\times21}$ tends to realign with the results for the $G_{9\times9}$ Monte Carlo runs. Figure 5.26 offers a side-by-side comparison of the heuristic results.

Table 5.12: $G_{21\times21}$ with 13 Dancers, Monte Carlo Results

Graph	Min. Distance					
$G_{21\times21}$	4	5	6	7	8	9
Local Improvement	6	76	419	422	77	0
Weighted Equivalent	1	5	65	503	413	13
Simulated Annealing	2	36	349	536	77	0
Reduced Step	0	0	19	167	760	54
Move To Open Area	0	0	0	213	748	39

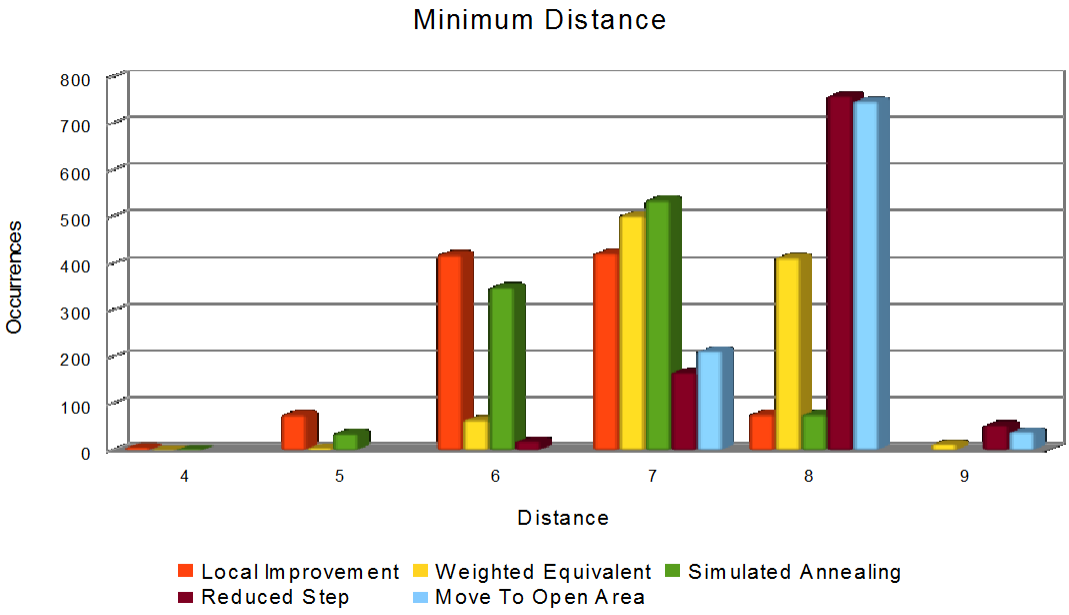

Figure 5.26: $G_{21\times21}$ Frequency Distribution Graph

5.6 Torus Analysis

The final graphs considered for analysis are the torus graphs. Torus graphs are similar to cycle graphs because they do not have fixed boundaries as the path and grid graphs.

5.6.1 $T_{8\times12}$

The first of the torus graphs considered is a $T_{8\times12}$ with 12 dancer vertices. Just as seen with the cycle, a configuration having a minimum distance of 1 was not observed with the torus. Both the Strict Local Improvement (Simulated Annealing is very similar) and Weighted Equivalent heuristics are still capable of producing poor dancer placement solutions. Figure 5.27 illustrates a poor distribution created by

the Strict Local Improvement heuristic and Figure 5.28 illustrates a poor distribution created by the Weighted Equivalent heuristic.

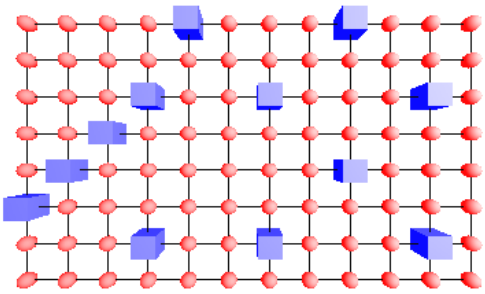

Figure 5.27: $T_{8\times12}$, Strict Local Improvement: Poor Dancer Positions

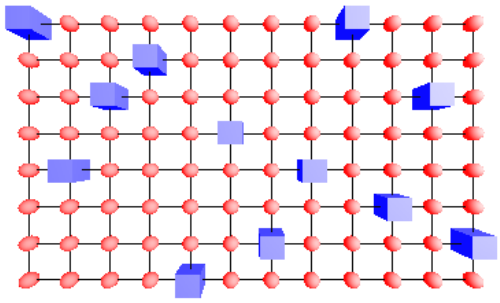

Figure 5.28: $T_{8\times12}$, Weighted Equivalent: Poor Dancer Positions

Table 5.13 shows the distributions of the MMD parameter for the different heuristics. The low optimal MMD bound of this test case limits the distributions only 2 different MMD parameter values. The distributions follows the same heuristics trends as for all the other graphs already considered. Figure 5.29 offers a side by side comparison of the heuristic results.

Table 5.13: $T_{8\times12}$ with 12 Dancers, Monte Carlo Results

Graph	Min. Distance		
$T_{8\times12}$	2	3	4
Local Improvement	693	307	0
Weighted Equivalent	110	890	0
Simulated Annealing	432	568	0
Reduced Step	43	957	0
Move To Open Area	4	994	2

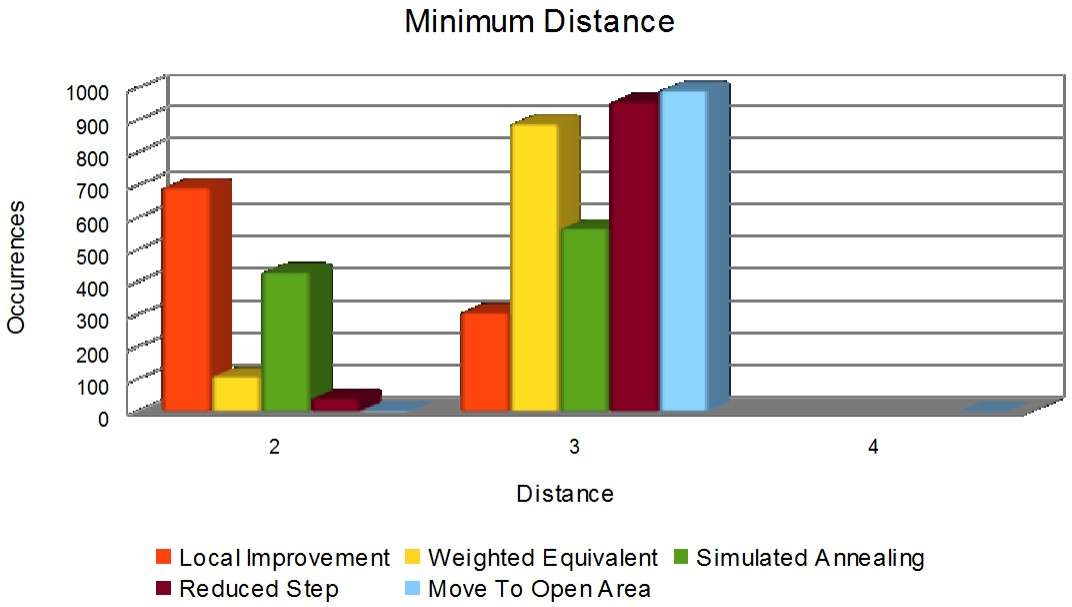

Figure 5.29: $T_{8\times12}$ Frequency Distribution Graph

5.6.2 $T_{12\times18}$

Next, a $T_{12\times18}$ torus with 12 dancer vertices is tested. Table 5.14 shows the distributions of the minimum distance for the different heuristics. The heuristic distributions for these torus dimensions are much more similar than the previous torus test bed case. The Reduced Step and Move to Open Area heuristcs continue to prove the mose reliable for near optimal solutions. Figure 5.30 offers a side-by-side comparison of the heuristic results.

5.6.3 $T_{13\times19}$

A $T_{13\times19}$ torus with 12 dancer vertices is selected as the intermediate torus configuration for comparison. Table 5.14 shows the distributions of the minimum distance for the different heuristics. This torus increases each dimension by one,

Table 5.14: $T_{12\times18}$ with 12 Dancers, Monte Carlo Results

Graph $T_{12\times18}$	Min. Distance 2	3	4	5
Local Improvement	1	148	831	20
Weighted Equivalent	0	14	749	237
Simulated Annealing	0	50	869	81
Reduced Step	0	0	537	463
Move To Open Area	0	0	586	414

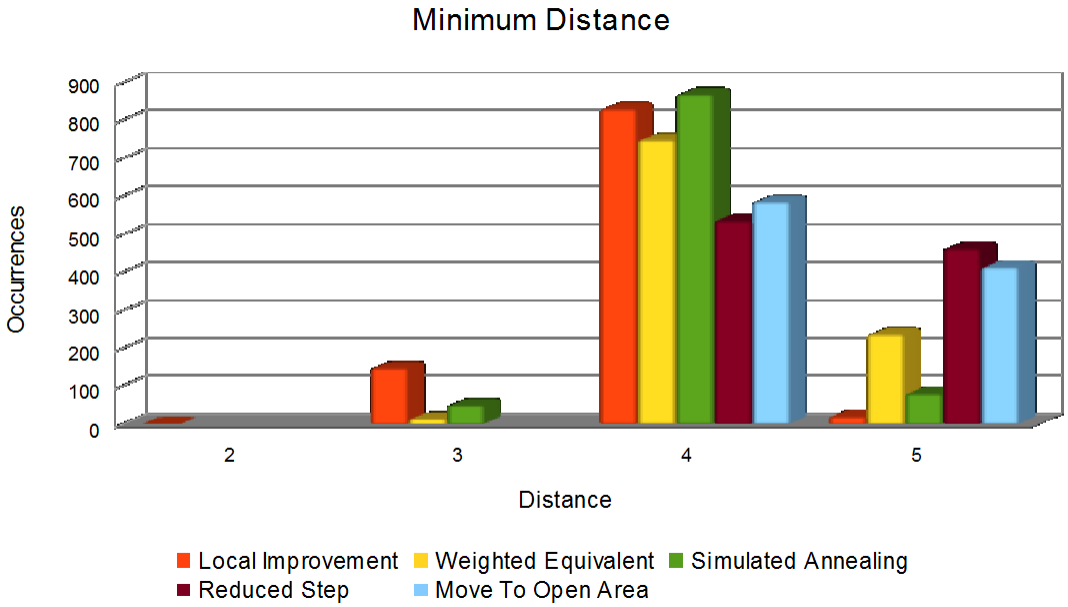

Figure 5.30: $T_{12\times18}$ Frequency Distribution Graph

allowing additional space for the movement of dancers throughout the torus. The distribution trends are very similar to the non-test bed analysis performed for the grid. Certain heuristics are able to improve their MMD distributions significantly while the previously poor performing heuristics mark little improvement. Figure 5.31 offers a side-by-side comparison of the heuristic results.

Table 5.15: $T_{13\times19}$ with 12 Dancers, Monte Carlo Results

Graph $T_{13\times19}$	Min. Distance 2	3	4	5	6
Local Improvement	1	65	720	214	0
Weighted Equivalent	0	8	484	508	0
Simulated Annealing	0	16	634	349	1
Reduced Step	0	0	72	836	92
Move To Open Area	0	0	93	894	13

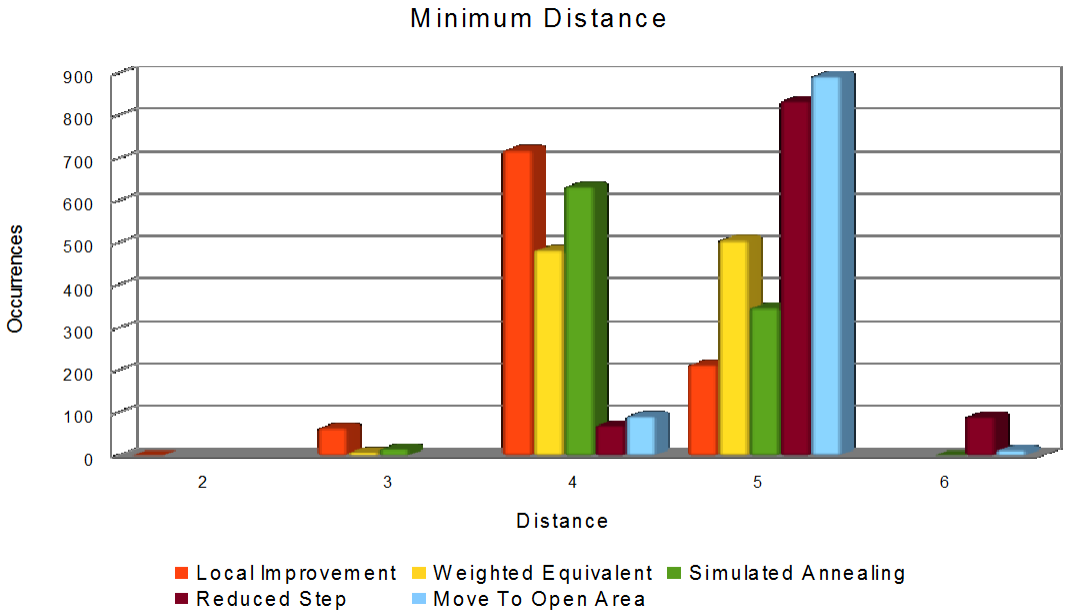

Figure 5.31: $T_{13\times19}$ Frequency Distribution Graph

5.6.4 $T_{16\times24}$

A $T_{16\times24}$ torus with 12 dancer vertices is the final class of graphs considered for analysis. Table 5.16 shows the distributions of the minimum distance for the different heuristics. Figure 5.32 offers a side by side comparison of the heuristic results.

Table 5.16: $T_{16\times24}$ with 12 Dancers, Monte Carlo Results

Graph $T_{16\times24}$	Min. Distance 3	4	5	6	7
Local Improvement	1	45	497	452	5
Weighted Equivalent	0	5	199	719	77
Simulated Annealing	0	16	370	604	10
Reduced Step	0	0	17	806	177
Move To Open Area	0	0	9	897	94

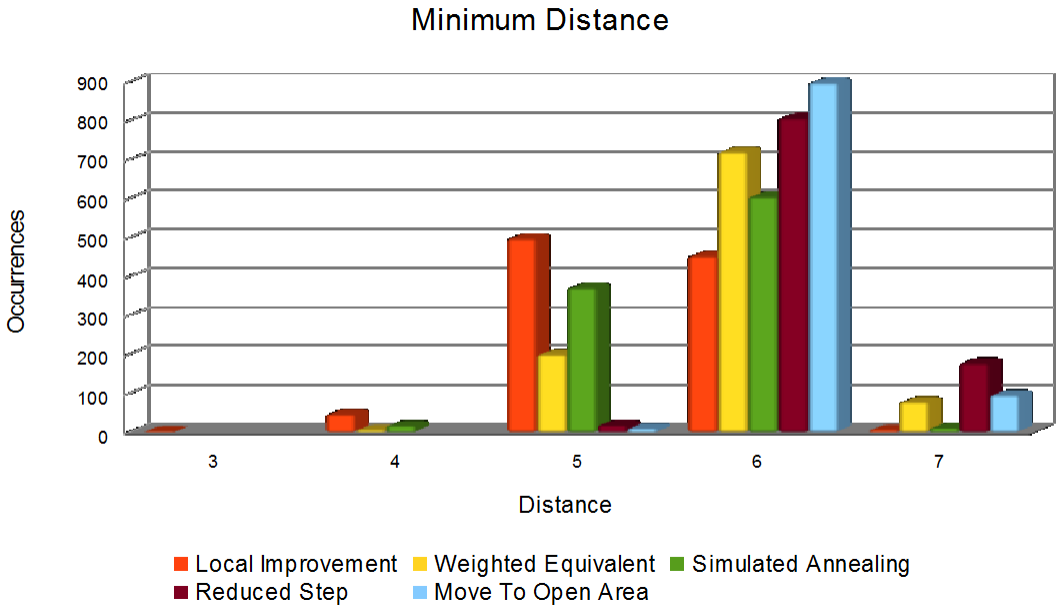

Figure 5.32: $T_{16\times24}$ Frequency Distribution Graph

5.7 Summary

Based on the results of all Monte Carlo simulations, it is apparent that the Strict Local Improvement heuristic produces rather poor solutions the majority of the time. Similarly, the method of implementing randomness within Simulated Annealing does very little to distinguish it from the results obtained through the Strict Local Improvement heuristic. The Weighted Equivalent heuristic produces markedly better

results with very little added complexity to the Strict Local Improvement heuristic, and as the Reduced Step heuristic uses the same technique throughout its iterations, this may be partially why this heuristic produced the most reliable and optimal solutions. Though not necessariliy supported by the simulation results of any of the graphs of focus in this research, for an arbitrary graph, the Move to Open Area heuristic seems to guarantee a reasonable upper bound.

While complexity was touched on in an earlier section, the analysis did not include any quantification of the runtime of each heuristic. Further complexity analysis will be mentioned as future work. However, it should be noted that while the Reduced Step heuristic often had the most favorable MMD frequency distribution, a qualitative description comparing its execution time to the other heuristics would be that it took a greater factor of time to execute the Monte Carlo simulations.

CHAPTER 6

CONCLUSION

It is what it is...

—Charles Groves

The heuristic research is concluded by summarizing the successful outcome of the two goals of focus in this research and presenting ideas for future research. There is a great deal of research for future consideration.

6.1 Summary of Results

Several of the simulated heuristics demonstrate consistently high optimized parameter values for an arbitrary initial state. Those that do not may still produce acceptable solutions for a subset of the possible initial states. The number of graph classes analyzed was too limited to validate success for an arbitrary graph. Based on the observations of the heuristics executed on the path, grid, and torus graphs, the greatest obstacle to equal distribution of elements in a graph was the density of the graph; it is the straight line structure of the path graph and the lack of diagonal movement on the grid and torus graphs that frequently limit solutions to premature terminating states. It is expected that the heuristics would be very successful for

graphs with a greater average vertex degree than the path, torus, and grid graphs already observed.

The assumptions made for the heuristics modeling behavior appear sound, but in the absence of recorded data, the behavior is not related to any known models. The successful abstraction of the heuristics to modeled behaviors indicates the possible presence of such optimizing behaviors in nature.

6.2 Future Research

The dancers' placement problem carries a great deal of significance as it is applicable to so many different fields of science and its parallel, the sphere packing problem, remains a largely unsolved problem. As such, there is a great deal of research which has still not been done.

There are many models of this problem that have already been actively researched. As with any study there lies a great deal of extension that could be done with the already existing methods. Below, several techniques are mentioned briefly, which have already been explored. As a late discovery in this research process, no comparisons between these "heuristics" and this papers' heuristics were made (Bowick, Cecka, Giomi, Middleton, Zielnicki [3]).

Relaxation Force vectors are calculated for every vertex pair and normalized against the heuristic unit step, vertices are moved in accordance with the combined forces acting on them

Local Relaxation This method follows the prevoius relaxation heuristic but only considers the force from local vertex neighbors

Local Monte Carlo Uses a localized Gaussian probability distribution to determine movement and if movement lowers energy potential accepts position with a Boltzmann probability distribution

Thermal Relaxation Hybrid approach combining local relaxation and local Monte Carlo methods

Monte Carlo Anneal A modified version of Krauth's annealing algorithm for arranging circles on a sphere

Lattice Energy Treats the edges as springs and conserves the Delaunay trigulation of a point set

Genetic Algorithm General Genetic Algorithm

There was also no comparison made to continuous distribution techniques. The close relationship that the dancers' placement problem has with sphere packing demands comparison to see in what ways the two domains overlap or complement each other. Can continuous techniques be used effectively to find discrete solutions (Chen, Huy, Huang, Li, Xu [4])?

Additional research should expand the number of analyzed graph classes. While the dancers' placement problem models the distribution of elements on a surface rather well, when the problem is compared to coding theory, analysis of graph classes such as hypercubes could provide a means to evalulate whether the analyzed

heuristics are capable of selecting codewords with reasonable success. For the graph classes already considered, analysis needs to be performed for all heuristics on the same set of inital starting positions in addition to random starting positions. If the number of starting configurations is sufficiently small, they should be exhaustively simulated for each heuristic.

As it pertains to the work directly attributed to this paper there is certainly enough room for future research. First as it pertains to the heuristic, there are many parameters which could be altered. Our original heuristic limited step movement to one step. This prevents some configurations such as the grid diagonal from ever achieving an optimal solution if they are already locked into their initial position because it is already locally optimal. The issue becomes more complex as we consider just how many steps should be allowed to be taken. Depending on the scale of the allowed steps to the size diameter of the graph, we may end up with the same heuristic as stated as the last considered; i.e., once single step local optimal configuration has been determined, we look for an open vertex and move our vertex with the closest neighbors to the open position. Perhaps movement decisions should always be made to the most open space. This is at least one other heuristic change which was not considered. It is expected that combining the Reduced Step heuristic with the Move to Open Area heuristic would produce even better results than either one of them by themselves.

Complexity analysis of the heuristics is a large enough topic to which research can be devoted. The illustration provided in our research was introductory. There are likely techniques which could be used to take advantage of many of the symmetric

states inherent to the graphs and might also be a means of improving the heuristics once such properties were examined. Finally, as insufficient examples of the heuristics were analyzed for complexity, there was no numerical analysis performed for the results of the average and worst case complexities. It was already stated that as number of terminating states varies in an inconsistent manner; therefore, expressing complexity in a closed form in terms of the input parameters (graph size and number of dancers) is not possible. However, this does not rule out a close curve-fitting relationship between some parameters of the system.

The other class of topics to be explored was how well our stated problem maps to real applications. From the beginning, it was known that the distribution of electrons on the surface of a sphere was a potential application for the dancing problem, as this was in many ways similar to the origin of the model for Thomson's problem. There are still models being developed along these lines, as in the case of multi-electron bubbles. Many of the applications which we mentioned in our introduction were not quantified with any data and appear to have little research dedicated to them. Consider the animal population problem. It was presented as a potential application without mentioning any real comparisons between obtained animal population distributions and any configurations generated by our heuristics. There are certain additional distinctions which would need to considered when comparing animal population distributions, the first being different notions of distance. Distance is usually spoken of in meters or feet; when dealing with animals, the distance function in determining repellent behavior in animals may be based on food or water sources, and other terrain considerations.

APPENDIX

SOURCE CODE

```
#!/usr/bin/python
# mainwindow.py
# Author: David Barber

import sys
import math
import cmath
import ThesisSim
import string

from PIL import Image

from OpenGL.GL import *
from OpenGL.GLU import *
from OpenGL.GLUT import *
from PyQt4 import QtGui, QtCore
from PyQt4.QtOpenGL import *

### Template code generated by Qt Designer suite

try:
    _fromUtf8 = QtCore.QString.fromUtf8
except AttributeError:
    _fromUtf8 = lambda s: s


class MainWindow(QtGui.QMainWindow):
    def __init__(self):
        QtGui.QMainWindow.__init__(self)
        self.ui = Ui_MainWindow()
        self.ui.setupUi(self)
```

```
        self.setWindowTitle('mainwindow')

        exit = QtGui.QAction(QtGui.QIcon('icons/exit.png'), '
            Exit', self)
        exit.setShortcut('Ctrl+Q')
        exit.setStatusTip('Exit application')
        self.connect(exit, QtCore.SIGNAL('triggered()'),
            QtCore.SLOT('close()'))

        self.statusBar()

        menubar = self.menuBar()
        file = menubar.addMenu('&File')
        file.addAction(exit)

        toolbar = self.addToolBar('Exit')
        toolbar.addAction(exit)



class Ui_MainWindow(object):
    def setupUi(self, MainWindow):
        MainWindow.setObjectName(_fromUtf8("MainWindow"))
        MainWindow.resize(440, 750)
        self.centralwidget = QtGui.QWidget(MainWindow)
        self.centralwidget.setObjectName(_fromUtf8("
            centralwidget"))
        self.comboBox = QtGui.QComboBox(self.centralwidget)
        self.comboBox.setGeometry(QtCore.QRect(130, 600, 251,
             22))
        font = QtGui.QFont()
        font.setPointSize(10)
        self.comboBox.setFont(font)
        self.comboBox.setObjectName(_fromUtf8("comboBox"))
        self.comboBox.addItem(_fromUtf8(""))
        self.comboBox.addItem(_fromUtf8(""))
        self.comboBox.addItem(_fromUtf8(""))
        self.comboBox.addItem(_fromUtf8(""))
        self.comboBox.addItem(_fromUtf8(""))
        self.label = QtGui.QLabel(self.centralwidget)
        self.label.setGeometry(QtCore.QRect(40, 600, 71, 21))
        font = QtGui.QFont()
        font.setPointSize(10)
        self.label.setFont(font)
```

```
self.label.setObjectName(_fromUtf8("label"))
self.pushButton = QtGui.QPushButton(self.
    centralwidget)
self.pushButton.setGeometry(QtCore.QRect(180, 400,
    91, 31))
self.pushButton.setObjectName(_fromUtf8("pushButton")
    )
self.comboBox_2 = QtGui.QComboBox(self.centralwidget)
self.comboBox_2.setGeometry(QtCore.QRect(130, 510,
    251, 22))
font = QtGui.QFont()
font.setPointSize(10)
self.comboBox_2.setFont(font)
self.comboBox_2.setObjectName(_fromUtf8("comboBox_2")
    )
self.comboBox_2.addItem(_fromUtf8(""))
self.comboBox_2.addItem(_fromUtf8(""))
self.comboBox_2.addItem(_fromUtf8(""))
self.comboBox_2.addItem(_fromUtf8(""))
self.label_2 = QtGui.QLabel(self.centralwidget)
self.label_2.setGeometry(QtCore.QRect(40, 510, 71,
    21))
font = QtGui.QFont()
font.setPointSize(10)
self.label_2.setFont(font)
self.label_2.setObjectName(_fromUtf8("label_2"))
self.widget = SpiralWidget(self.centralwidget)
self.widget.setGeometry(QtCore.QRect(10, 10, 421,
    381))
self.widget.setObjectName(_fromUtf8("widget"))
self.horizontalSlider = QtGui.QSlider(self.
    centralwidget)
self.horizontalSlider.setGeometry(QtCore.QRect(130,
    460, 251, 20))
self.horizontalSlider.setMinimum(2)
self.horizontalSlider.setMaximum(25)
self.horizontalSlider.setOrientation(QtCore.Qt.
    Horizontal)
self.horizontalSlider.setObjectName(_fromUtf8("
    horizontalSlider"))
self.label_3 = QtGui.QLabel(self.centralwidget)
self.label_3.setGeometry(QtCore.QRect(40, 460, 51,
    21))
font = QtGui.QFont()
font.setPointSize(10)
```

```
self.label_3.setFont(font)
self.label_3.setObjectName(_fromUtf8("label_3"))
self.label_4 = QtGui.QLabel(self.centralwidget)
self.label_4.setGeometry(QtCore.QRect(100, 460, 31,
    21))
font = QtGui.QFont()
font.setPointSize(10)
self.label_4.setFont(font)
self.label_4.setObjectName(_fromUtf8("label_4"))
self.textEdit = QtGui.QTextEdit(self.centralwidget)
self.textEdit.setGeometry(QtCore.QRect(130, 550, 61,
    31))
self.textEdit.setObjectName(_fromUtf8("textEdit"))
self.textEdit_2 = QtGui.QTextEdit(self.centralwidget)
self.textEdit_2.setGeometry(QtCore.QRect(290, 550,
    61, 31))
self.textEdit_2.setObjectName(_fromUtf8("textEdit_2")
    )
self.label_5 = QtGui.QLabel(self.centralwidget)
self.label_5.setGeometry(QtCore.QRect(200, 560, 16,
    21))
font = QtGui.QFont()
font.setPointSize(10)
self.label_5.setFont(font)
self.label_5.setObjectName(_fromUtf8("label_5"))
self.label_6 = QtGui.QLabel(self.centralwidget)
self.label_6.setGeometry(QtCore.QRect(360, 560, 16,
    21))
font = QtGui.QFont()
font.setPointSize(10)
self.label_6.setFont(font)
self.label_6.setObjectName(_fromUtf8("label_6"))
self.checkBox = QtGui.QCheckBox(self.centralwidget)
self.checkBox.setGeometry(QtCore.QRect(130, 640, 70,
    21))
self.checkBox.setObjectName(_fromUtf8("checkBox"))
self.textEdit_3 = QtGui.QTextEdit(self.centralwidget)
self.textEdit_3.setGeometry(QtCore.QRect(180, 630,
    61, 31))
self.textEdit_3.setObjectName(_fromUtf8("textEdit_3")
    )
self.textEdit_3.setEnabled(False)
MainWindow.setCentralWidget(self.centralwidget)
self.menubar = QtGui.QMenuBar(MainWindow)
self.menubar.setGeometry(QtCore.QRect(0, 0, 439, 21))
```

```
        self.menubar.setObjectName(_fromUtf8("menubar"))
        MainWindow.setMenuBar(self.menubar)
        self.statusbar = QtGui.QStatusBar(MainWindow)
        self.statusbar.setObjectName(_fromUtf8("statusbar"))
        MainWindow.setStatusBar(self.statusbar)

        self.retranslateUi(MainWindow)
        QtCore.QMetaObject.connectSlotsByName(MainWindow)

    def retranslateUi(self, MainWindow):

        MainWindow.setWindowTitle(QtGui.QApplication.
            translate("MainWindow", "MainWindow", None, QtGui.
            QApplication.UnicodeUTF8))
        self.comboBox.setItemText(0, QtGui.QApplication.
            translate("MainWindow", "Local Strict Improvement
            ", None, QtGui.QApplication.UnicodeUTF8))
        self.comboBox.setItemText(1, QtGui.QApplication.
            translate("MainWindow", "Remote Weight", None,
            QtGui.QApplication.UnicodeUTF8))
        self.comboBox.setItemText(2, QtGui.QApplication.
            translate("MainWindow", "Simulated Anneal", None,
            QtGui.QApplication.UnicodeUTF8))
        self.comboBox.setItemText(3, QtGui.QApplication.
            translate("MainWindow", "Reduced Step Size", None,
             QtGui.QApplication.UnicodeUTF8))
        self.comboBox.setItemText(4, QtGui.QApplication.
            translate("MainWindow", "Move To Open Area", None,
             QtGui.QApplication.UnicodeUTF8))
        self.label.setText(QtGui.QApplication.translate("
            MainWindow", "Heuristic", None, QtGui.QApplication
            .UnicodeUTF8))

        self.pushButton.setText(QtGui.QApplication.translate
            ("MainWindow", "Run", None, QtGui.QApplication.
            UnicodeUTF8))
        MainWindow.connect(self.pushButton, QtCore.SIGNAL('
            clicked()'), self.startButtonPressed)

        self.textEdit.setText("5")
        self.textEdit.setEnabled(False)
        self.textEdit_2.setText("5")
        self.textEdit_3.setEnabled(False)
        self.textEdit_3.setText("1")
```

```
        MainWindow.connect(self.horizontalSlider, QtCore.
            SIGNAL('valueChanged(int)'), self.
            sliderValueChanged)

        MainWindow.connect(self.checkBox, QtCore.SIGNAL('
            stateChanged(int)'), self.checkBoxRunsChanged)

        self.comboBox_2.setItemText(0, QtGui.QApplication.
            translate("MainWindow", "Path", None, QtGui.
            QApplication.UnicodeUTF8))
        self.comboBox_2.setItemText(1, QtGui.QApplication.
            translate("MainWindow", "Cycle", None, QtGui.
            QApplication.UnicodeUTF8))
        self.comboBox_2.setItemText(2, QtGui.QApplication.
            translate("MainWindow", "Grid", None, QtGui.
            QApplication.UnicodeUTF8))
        self.comboBox_2.setItemText(3, QtGui.QApplication.
            translate("MainWindow", "Torus", None, QtGui.
            QApplication.UnicodeUTF8))
        MainWindow.connect(self.comboBox_2, QtCore.SIGNAL('
            currentIndexChanged(int)'), self.
            comboGraphTypeChanged)

        self.label_2.setText(QtGui.QApplication.translate("
            MainWindow", "Graph", None, QtGui.QApplication.
            UnicodeUTF8))
        self.label_3.setText(QtGui.QApplication.translate("
            MainWindow", "Selected", None, QtGui.QApplication.
            UnicodeUTF8))
        self.label_4.setText(QtGui.QApplication.translate("
            MainWindow", "2", None, QtGui.QApplication.
            UnicodeUTF8))
        self.label_5.setText(QtGui.QApplication.translate("
            MainWindow", "m", None, QtGui.QApplication.
            UnicodeUTF8))
        self.label_6.setText(QtGui.QApplication.translate("
            MainWindow", "n", None, QtGui.QApplication.
            UnicodeUTF8))
        self.checkBox.setText(QtGui.QApplication.translate("
            MainWindow", "Runs", None, QtGui.QApplication.
            UnicodeUTF8))

    def startButtonPressed(self):
        if self.pushButton.text() == "Run":
```

```
            self.pushButton.setText(QtGui.QApplication.
                translate("MainWindow", "Reset", None, QtGui.
                QApplication.UnicodeUTF8))

            self.horizontalSlider.setEnabled(False)
            self.textEdit.setEnabled(False)
            self.textEdit_2.setEnabled(False)
            self.textEdit_3.setEnabled(False)
            self.comboBox.setEnabled(False)
            self.comboBox_2.setEnabled(False)
            self.checkBox.setEnabled(False)

            self.issueStart()
        else:
            self.pushButton.setText(QtGui.QApplication.
                translate("MainWindow", "Run", None, QtGui.
                QApplication.UnicodeUTF8))

            self.horizontalSlider.setEnabled(True)
            self.comboGraphTypeChanged(self.comboBox.
                currentIndex())
            self.textEdit_2.setEnabled(True)
            self.comboBox.setEnabled(True)
            self.comboBox_2.setEnabled(True)
            self.checkBox.setEnabled(True)
            self.checkBoxRunsChanged(self.checkBox.checkState
                ())

            self.issueStop()

    def resetButtonPressed(self):
        self.issueReset()

    def sliderValueChanged(self, newValue):
        self.label_4.setText(QtGui.QApplication.translate("
            MainWindow", str(newValue), None, QtGui.
            QApplication.UnicodeUTF8))

    def comboGraphTypeChanged(self, newIndex):
        if self.comboBox_2.currentText().__str__() == 'Path'
            or \
              self.comboBox_2.currentText().__str__() == 'Cycle
                  ':
              self.textEdit.setEnabled(False)
        else:
```

```
            self.textEdit.setEnabled(True)


    def checkBoxRunsChanged(self, newValue):
        if newValue == QtCore.Qt.Unchecked:
            self.textEdit_3.setEnabled(False)
        else:
            self.textEdit_3.setEnabled(True)

    def issueStart(self):
        sim = ThesisSim.dancer_sim(self.widget, self.
            comboBox_2.currentText().__str__(), self.comboBox.
            currentText().__str__(), int(self.textEdit_2.
            toPlainText()), int(self.textEdit.toPlainText()))
        sim.run(self.horizontalSlider.value(), int(self.
            textEdit_3.toPlainText()))

    def issueStop(self):
        pass

    def issueReset(self):
        pass

class SpiralWidget(QGLWidget):
    '''
    Widget for drawing two spirals.
    '''

    def __init__(self, parent):
        QGLWidget.__init__(self, parent)
        self.zoom = 10
        self.rotx = 0
        self.roty = 0
        self.rotz = 0
        self.begin_x = 0
        self.begin_y = 0
        self.rotate_xy = False
        self.rendermethod = self.paintSpiral
        self.graph = {}
        self.selected = set()

        self.run_no = 1
        self.graph_type = ""
        self.heuristic_type = ""
        self.no_selected = 0
```

```
        self.dim1 = 0
        self.dim2 = 0

# Spiral code adopted from public domain code and used as
   template
# for other painting operations
'''
Created on Jul 7, 2009

author: Stou Sandalski (stou@icapsid.net)
license:  Public Domain
'''
    def paintSpiral(self):
        # Draw the spiral in 'immediate mode'
        # WARNING: You should not be doing the spiral
           calculation inside the loop
        # even if you are using glBegin/glEnd, sin/cos are
           fairly expensive functions
        # I've left it here as is to make the code simpler.
        radius = 1.0
        x = radius*math.sin(0)
        y = radius*math.cos(0)
        glColor(0.0, 128.0, 0.0)
        glBegin(GL_LINE_STRIP)
        for deg in xrange(1000):
            glVertex(x, y, 0.0)
            rad = math.radians(deg)
            radius -= 0.001
            x = radius*math.sin(rad)
            y = radius*math.cos(rad)
        glEnd()

        glEnableClientState(GL_VERTEX_ARRAY)

        spiral_array = []

        # Second Spiral using "array immediate mode" (i.e.
           Vertex Arrays)
        radius = 0.8
        x = radius*math.sin(0)
        y = radius*math.cos(0)
        glColor(128.0, 0.0, 0.0)
        for deg in xrange(820):
            spiral_array.append([x, y])
            rad = math.radians(deg)
```

```
            radius -= 0.001
            x = radius*math.sin(rad)
            y = radius*math.cos(rad)

        glVertexPointerf(spiral_array)
        glDrawArrays(GL_LINE_STRIP, 0, len(spiral_array))

        glFlush()

    # method for drawing path graph
    def paintPath(self):
        self.renderRunNo()

        glRotatef(45., 0., 0., 1.)
        x = max(el[0] for el in self.graph.keys())
        y = max(el[1] for el in self.graph.keys())

        # center natural grid points
        glTranslatef(-(x+1)/2.0, -(y+1)/2.0, 0)

        str_graph = 'Path_' + str(x) \
                    + ',_' + str(len(self.selected)) + '
                        _selected'
        glWindowPos2f(1.,1.)
        for c in str_graph:
            glutBitmapCharacter(GLUT_BITMAP_8_BY_13, ord(c))

        for node in set(self.graph.keys()) - self.selected:
            glColor(1.0, 0.0, 0.0)
            glPushMatrix()
            glTranslatef(node[0],node[1],0.) #move to where
               we want to put object
            glutSolidSphere(.20,20,20) # make radius 1 sphere
                of res 10x10
            glPopMatrix()


        for sel_node in self.selected:
            glColor(0.0, 0.0, 1.0)
            glPushMatrix()
            glTranslatef(sel_node[0],sel_node[1],0.) #move to
                where we want to put object
            glutSolidCube(.60) # make radius 1 sphere of res
               10x10
            glPopMatrix()
```

```
        # lazy impl
        glBegin(GL_LINES)
        glColor(0.0, 0.0, 0.0)
        for node in self.graph.items():
            for adj_node in node[1].keys():
                glVertex(node[0][0], node[0][1], 0.0)
                glVertex(adj_node[0], adj_node[1], 0.0)
        glEnd()

    # method for drawing cycle graph
    def paintCycle(self):
        self.renderRunNo()

        x = max(el[0] for el in self.graph.keys())
        y = max(el[1] for el in self.graph.keys())

        str_graph = 'Cycle ' + str(x) \
                    + ', ' + str(len(self.selected)) + '
                       selected'
        glWindowPos2f(1.,1.)
        for c in str_graph:
            glutBitmapCharacter(GLUT_BITMAP_8_BY_13, ord(c))

        plotPoints = set(self.graph.keys()) - self.selected
        for node in plotPoints:
            glColor(1.0, 0.0, 0.0)
            glPushMatrix()
            pos = cmath.rect(5, (node[0]-1)*2*math.pi/x)
            glTranslatef(pos.real, pos.imag,0.) #move to
               where we want to put object
            glutSolidSphere(.20,20,20) # make radius 1 sphere
                of res 10x10
            glPopMatrix()

        for node in self.selected:
            glColor(0.0, 0.0, 1.0)
            glPushMatrix()
            pos = cmath.rect(5, (node[0]-1)*2*math.pi/x)
            glTranslatef(pos.real, pos.imag,0.) #move to
               where we want to put object
            glutSolidCube(.60) # make radius 1 sphere of res
               10x10
            glPopMatrix()
```

```
        # lazy impl
        glBegin(GL_LINES)
        glColor(0.0, 0.0, 0.0)
        for node in self.graph.items():
            for adj_node in node[1].keys():
                pos1 = cmath.rect(5, (node[0][0]-1)*2*math.pi
                   /x)
                pos2 = cmath.rect(5, (adj_node[0] -1)*2*math.
                   pi/x)
                glVertex(pos1.real, pos1.imag, 0.0)
                glVertex(pos2.real, pos2.imag, 0.0)
        glEnd()

    # method for drawing grid graph
    def paintGrid(self):
        self.renderRunNo()

        x = max(el[0] for el in self.graph.keys())
        y = max(el[1] for el in self.graph.keys())


        # center natural grid points
        glTranslatef(-(x+1)/2.0, -(y+1)/2.0, 0)

        str_graph = 'Grid ' + str(x) + ' by ' + str(y) \
                    + ', ' + str(len(self.selected)) + '
                       selected'
        glWindowPos2f(1.,1.)
        for c in str_graph:
            glutBitmapCharacter(GLUT_BITMAP_8_BY_13, ord(c))

        for node in set(self.graph.keys()) - self.selected:
            glColor(1.0, 0.0, 0.0)
            glPushMatrix()
            glTranslatef(node[0],node[1],0.) #move to where
               we want to put object
            glutSolidSphere(.20,20,20) # make radius 1 sphere
                of res 10x10
            glPopMatrix()


        for sel_node in self.selected:
            glColor(0.0, 0.0, 1.0)
            glPushMatrix()
```

```
            glTranslatef(sel_node[0],sel_node[1],0.) #move to
                where we want to put object
            glutSolidCube(.60) # make radius 1 sphere of res
               10x10
            glPopMatrix()


        # lazy impl
        glBegin(GL_LINES)
        glColor(0.0, 0.0, 0.0)
        for node in self.graph.items():
            for adj_node in node[1].keys():
                glVertex(node[0][0], node[0][1], 0.0)
                glVertex(adj_node[0], adj_node[1], 0.0)
        glEnd()

    # method for drawing torus graph
    def paintTorus(self):
        self.renderRunNo()

        x = max(el[0] for el in self.graph.keys())
        y = max(el[1] for el in self.graph.keys())

        str_graph = 'Torus ' + str(x) + ' by ' + str(y) \
                    + ', ' + str(len(self.selected)) + '
                       selected'
        glWindowPos2f(1.,1.)
        for ch in str_graph:
            glutBitmapCharacter(GLUT_BITMAP_8_BY_13, ord(ch))


        # angle step sizes
        d_u = 2*math.pi/x
        d_v = 2*math.pi/y

        # radius from center of hole to center of radius
        c = 5
        # radius of tube
        a = 2

        plotPoints = set(self.graph.keys()) - self.selected

        for node in plotPoints:
            glColor(1.0, 0.0, 0.0)
            glPushMatrix()
```

```
        x_pt = (c + a*math.cos((node[1] - 1)*d_v))*math.
           cos((node[0] - 1)*d_u)
        y_pt = (c + a*math.cos((node[1] - 1)*d_v))*math.
           sin((node[0] - 1)*d_u)
        z_pt = a*math.sin((node[1] - 1)*d_v)
        glTranslatef(x_pt, y_pt, z_pt) #move to where we
           want to put object
        glutSolidSphere(.10,20,20) # make radius 1 sphere
            of res 10x10
        glPopMatrix()

    for node in self.selected:
        glColor(0.0, 0.0, 1.0)
        glPushMatrix()
        x_pt = (c + a*math.cos((node[1] - 1)*d_v))*math.
           cos((node[0] - 1)*d_u)
        y_pt = (c + a*math.cos((node[1] - 1)*d_v))*math.
           sin((node[0] - 1)*d_u)
        z_pt = a*math.sin((node[1] - 1)*d_v)
        glTranslatef(x_pt, y_pt, z_pt) #move to where we
           want to put object
        glutSolidCube(.60) # make radius 1 sphere of res
           10x10
        glPopMatrix()


    # lazy impl
    glBegin(GL_LINES)
    glColor(0.0, 0.0, 0.0)
    for node in self.graph.items():
        for adj_node in node[1].keys():
            x_pt1 = (c + a*math.cos((node[0][1] - 1)*d_v)
               )*math.cos((node[0][0] - 1)*d_u)
            y_pt1 = (c + a*math.cos((node[0][1] - 1)*d_v)
               )*math.sin((node[0][0] - 1)*d_u)
            z_pt1 = a*math.sin((node[0][1] - 1)*d_v)

            x_pt2 = (c + a*math.cos((adj_node[1] - 1)*d_v
               ))*math.cos((adj_node[0] - 1)*d_u)
            y_pt2 = (c + a*math.cos((adj_node[1] - 1)*d_v
               ))*math.sin((adj_node[0] - 1)*d_u)
            z_pt2 = a*math.sin((adj_node[1] - 1)*d_v)

            glVertex(x_pt1, y_pt1, z_pt1)
            glVertex(x_pt2, y_pt2, z_pt2)
```

```
        glEnd()

    # sets up base paint operation
    def paintGL(self):
        '''
        Drawing routine
        '''

        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
        glClearColor(1.0, 1.0, 1.0, 0.);
        glLoadIdentity()

        glMatrixMode(GL_PROJECTION)
        glLoadIdentity()
        gluPerspective(100.0, 1.0, 1.0, 50.0)

        glMatrixMode(GL_MODELVIEW)
        glLoadIdentity()
        gluLookAt(0,0,self.zoom,0,0,0,0,1,0)

        glRotatef(self.roty,0,1,0)
        glRotatef(self.rotx,1,0,0)

        self.rendermethod()

    # renders graph information
    def renderRunNo(self):
        glColor(0., 0., 0.)
        str_graph = 'Run Number: ' + str(self.run_no)
        glWindowPos2f(1.,14.)
        for c in str_graph:
            glutBitmapCharacter(GLUT_BITMAP_8_BY_13, ord(c))

    # saves graph image
    def saveImage(self, graph_type, heuristic, selected, dim1
        , dim2, run_no, min_dist):
        pixelBuffer = glReadPixels(1, 1, 421, 381, GL_RGB,
            GL_UNSIGNED_BYTE)
        image = Image.fromstring( "RGB", (421, 381),
            pixelBuffer )
        image = image.transpose( Image.FLIP_TOP_BOTTOM)
        filepath = self.graph_type + '_' + self.
            heuristic_type + '_select' + str(self.no_selected)
             + '_' + str(self.dim1) + 'by' + str(self.dim2) +
```

```
        '_run_' + str(self.run_no) + '_min_dist' + str(
        min_dist) + '.png'
    image.save( "./img3/" + filepath, "PNG" )

# resize render window
def resizeGL(self, w, h):
    '''
    Resize the GL window
    '''

    glViewport(0, 0, w, h)
    glMatrixMode(GL_PROJECTION)
    glLoadIdentity()
    gluPerspective(40.0, 1.0, 1.0, 30.0)

# intialize render engine
def initializeGL(self):
    '''
    Initialize GL
    '''

    # set viewing projection
    glClearColor(0.0, 0.0, 0.0, 1.0)
    glClearDepth(1.0)


    #setup lighting
    glEnable(GL_CULL_FACE)
    glEnable(GL_DEPTH_TEST)
    glEnable(GL_LIGHTING)
    lightZeroPosition = [10.,4.,10.,1.]
    lightZeroColor = [1.0,1.0,1.0,1.0] # greenish
    glLightfv(GL_LIGHT0, GL_POSITION, lightZeroPosition)
    glLightfv(GL_LIGHT0, GL_DIFFUSE, lightZeroColor)
    glLightf(GL_LIGHT0, GL_CONSTANT_ATTENUATION, 0.1)
    glLightf(GL_LIGHT0, GL_LINEAR_ATTENUATION, 0.05)
    glEnable(GL_LIGHT0)

    glColorMaterial ( GL_FRONT_AND_BACK, GL_EMISSION )
    glEnable(GL_COLOR_MATERIAL)

# moves view point based on scroll wheel
def wheelEvent(self, ev):
     numDegrees = ev.delta() / 8
     numSteps = numDegrees / 15
```

```
            self.zoom -= numSteps
            self.updateGL()
            ev.accept()

        # supporting function for axis rotations
        def mousePressEvent(self, ev):
            if ev.buttons() == QtCore.Qt.LeftButton:
                self.x_begin = ev.x()
                self.y_begin = ev.y()
                self.rotate_xy = True
            else:
                self.rotate_xy = False

        # performs axis rotations
        def mouseMoveEvent (self, ev):
            if self.rotate_xy:
                self.rotx = self.rotx + (ev.y() - self.y_begin)
                self.roty = self.roty + (ev.x() - self.x_begin)
                self.x_begin = ev.x()
                self.y_begin = ev.y()

            self.updateGL()
            ev.accept()

    # Beginning of Script
    app = QtGui.QApplication(sys.argv)
    main = MainWindow()
    main.show()
    sys.exit(app.exec_())

    #!usr/bin/env python
    # Thesis Simulation python module
    # Author: David Barber

    import sys, os
    import math
    import random
    import pickle
    import copy
    import time

    master_distance = {}

    # Class used to pass a mutable primitive
```

```
class ref_wrapper:
    def __init__(self, ref):
        self.obj = ref

### Graph utilities

# generate random sublist
def uniq_rand_obj_sublist(collection, size):
    random.seed()
    clone = copy.copy(collection)
    for i in range(0, size):
        yield clone.pop(random.randint(0, len(clone) - 1))

# generate random copy of vertex list
def rand_copy_list(orig_list):
    obj_list = list(orig_list)
    random.seed()
    while len(obj_list) > 0:
        yield obj_list.pop(random.randint(0, len(obj_list) -
           1))

# expand path graph
def expand_path(g, sel):
    temp = set()

    g = gen_path(len(g)*2)
    for el in sel:
        temp.add((el[0]*2, el[1]))

    sel = temp
    return g, sel

# expand cycle graph
def expand_cycle(g, sel):
    temp = set()

    g = gen_cycle(len(g)*2)
    for el in sel:
        temp.add((el[0]*2, el[1]))

    sel = temp
    return g, sel

# expand grid graph
def expand_grid(g, sel):
```

```
    h = max([i[0] for i in g.keys()])
    w = max([i[1] for i in g.keys()])
    temp = set()

    g = gen_grid(h*2, w*2)
    for el in sel:
        temp.add((el[0]*2, el[1]*2))

    sel = temp
    return g, sel

# expand torus graph
def expand_torus(g, sel):
    h = max([i[0] for i in g.keys()])
    w = max([i[1] for i in g.keys()])
    temp = set()

    g = gen_torus(h*2, w*2)
    for el in sel:
        temp.add((el[0]*2, el[1]*2))

    sel = temp
    return g, sel

# calculate arbitrary graph vertex distances
def arb_graph_distances(x, graph):
    visited = set([x])
    distance = {x : 0}
    for neighbor in graph[x].keys():
        distance[neighbor] = graph[x][neighbor]
    selectable = set(distance.keys()) - visited
    while len(selectable) > 0:
        v_min = min(selectable, key = lambda x: distance.get(
           x) )
        visited.add(v_min)
        selectable.remove(v_min)
        for v_min_neigh in graph[v_min].keys():
            segment = distance[v_min] + graph[v_min][
               v_min_neigh]
            if v_min_neigh not in distance.keys() or distance
               [v_min_neigh] > segment:
                distance[v_min_neigh] = segment
        selectable = set(distance.keys()) - visited

    del distance[x]
```

```
    return distance

# Much faster method of calculating distance between path
   vertices
def path_equigraph_dist(x, graph):
    distance = {}
    for v in graph:
        distance[v] = abs(x[0] - v[0])

    del distance[x]
    return distance

# Much faster method of calculating distance between cycle
   vertices
def cycle_equigraph_dist(x, graph):
    max_row = max(x[0] for x in graph.keys())
    max_col = max(x[1] for x in graph.keys())

    # internal fnc def handles graph boundary conditions
    def cycle_line_dist(x,y, _max):
        if abs(x - y) > _max / 2:
            return _max - abs(x - y)
        else:
            return abs(x - y)

    distance = {}
    for v in graph:
        distance[v] = cycle_line_dist(x[0],v[0], max_row)

    del distance[x]
    return distance

# Much faster method of calculating distance between grid
   vertices
def grid_equigraph_dist(x, graph):
    distance = {}
    for v in graph:
        distance[v] = abs(x[0] - v[0]) + abs(x[1] - v[1])

    del distance[x]
    return distance

# Much faster method of calculating distance between torus
   points
def torus_equigraph_dist(x, graph):
```

```
    max_row = max(x[0] for x in graph.keys())
    max_col = max(x[1] for x in graph.keys())

    # internal fnc def handles graph boundary conditions
    def cycle_line_dist(x,y, _max):
        if abs(x - y) > _max / 2:
            return _max - abs(x - y)
        else:
            return abs(x - y)

    distance = {}
    for v in graph:
        distance[v] = cycle_line_dist(x[0],v[0], max_row) +
           cycle_line_dist(x[1],v[1], max_col)

    del distance[x]
    return distance


# generates h by w grid
def gen_grid(h,w):
    #initialize graph map
    t = {}
    # construct all neighbors for (1,1) through (k, k)
    for i in range(1,h+1):
        for j in range(1,w+1):
            t[(i,j)] = {}
            if 0 < i - 1 <= h and 0 < j <= w:
                t[(i,j)][(i-1, j)] = 1
            if 0 < i + 1 <= h and 0 < j <= w:
                t[(i,j)][(i+1, j)] = 1
            if 0 < i <= h and 0 < j - 1 <= w:
                t[(i,j)][(i, j-1)] = 1
            if 0 < i <= h and 0 < j + 1 <= w:
                t[(i,j)][(i, j+1)] = 1
    return t

# generates h by w torus
def gen_torus(h, w):
    #initialize graph map
    t = {}
    # construct all neighbors for (1,1) through (k, k)
    for i in range(1,h+1):
        for j in range(1,w+1):
            t[(i,j)] = {}
```

```
            if 0 < i - 1 <= h and 0 < j <= w:
                t[(i,j)][(i-1, j)] = 1
            if 0 < i + 1 <= h and 0 < j <= w:
                t[(i,j)][(i+1, j)] = 1
            if 0 < i <= h and 0 < j - 1 <= w:
                t[(i,j)][(i, j-1)] = 1
            if 0 < i <= h and 0 < j + 1 <= w:
                t[(i,j)][(i, j+1)] = 1
            if 0 == i - 1 and 0 < j <= w:
                t[(i,j)][(h, j)] = 1
            if h < i + 1 and 0 < j <= w:
                t[(i,j)][(1, j)] = 1
            if 0 < i <= h and 0 == j - 1:
                t[(i,j)][(i, w)] = 1
            if 0 < i <= h and w < j + 1:
                t[(i,j)][(i, 1)] = 1
    return t

# generate n length path
def gen_path(n):
    #initialize graph map
    t = {}
    # construct all neighbors for (1,1) through (n, 1)
    for i in range(1, n + 1):
            t[(i,1)] = {}
            if i < n:
                t[(i,1)][(i+1,1)] = 1
            if i > 1:
                t[(i,1)][(i-1,1)] = 1

    return t

def gen_cycle(n):
    #initialize graph map
    t = {}
    # construct all neighbors for (1,1) through (n, 1)
    for i in range(1, n + 1):
            t[(i,1)] = {}
            if i < n:
                t[(i,1)][(i+1,1)] = 1
            if i > 1:
                t[(i,1)][(i-1,1)] = 1
            if i == 1:
                t[(1,1)][(n, 1)] = 1
            if i == n:
```

```
                t[(n, 1)][(1,1)] = 1

    return t

# generate potential grid positions
def new_pos_grid(v, g):
    # Calculate possible new position
    a = (v[0] + 1, v[1])
    if a in g.keys():
        yield a
    a = (v[0], v[1] + 1)
    if a in g.keys():
        yield a
    a = (v[0] - 1, v[1])
    if a in g.keys():
        yield a
    a = (v[0], v[1] - 1)
    if a in g.keys():
        yield a

# generate potential torus positions
def new_pos_torus(v, g):
    h = max(x[0] for x in g)
    w = max(x[1] for x in g)

    # Calculate possible new position
    a = (v[0] + 1, v[1])
    if a in g.keys():
        yield a
    else:
        yield (1, v[1])
    a = (v[0], v[1] + 1)
    if a in g.keys():
        yield a
    else:
        yield (v[0], 1)
    a = (v[0] - 1, v[1])
    if a in g.keys():
        yield a
    else:
        yield (h, v[1])
    a = (v[0], v[1] - 1)
    if a in g.keys():
        yield a
    else:
```

```
            yield (v[0], w)

    # generate potential path positions
    def new_pos_path(v, g):
        # Calculate possible new position
        a = (v[0] - 1, 1)
        if a in g.keys():
            yield a
        a = (v[0] + 1, 1)
        if a in g.keys():
            yield a

    # generate potential cycle positions
    def new_pos_cycle(v, g):
        # Calculate possible new position
        a = (v[0] - 1, 1)
        if (v[0] - 1, 1) in g.keys():
            yield (v[0] - 1, 1)
        else:
            yield (len(g), 1)

        if (v[0] + 1, 1) in g.keys():
            yield (v[0] + 1, 1)
        else:
            yield (1, 1)

    # global threshold for moving on equivalent positions
    g_threshold = 75

    ### Heuristics

    # Strict improvement
    def strict_improv_impl(orig_v, orig_g, orig_sel, adj_func,
       counter, dist_func):
        sig_selected = set(orig_sel)
        sig_selected.remove(orig_v)

        global master_distance

        # Obtain distances from vertex in question
        base_dist = master_distance[orig_v]

        # determine the minimum distance between this vertex and
           the others selected
```

```
        min_dist = min(base_dist[x] for x in sig_selected - set([
            orig_v]))

        # Container for distances of potential new points
        adj_dist = {}
        for adj_v in adj_func(orig_v, orig_g):
            adj_dist[adj_v] = master_distance[adj_v]

        # assume to begin that current point in question is best
        # if potential points has greater minimum distance from
            others move there
        best_v = orig_v
        for adj_v in rand_copy_list(set(adj_func(orig_v, orig_g))
            - sig_selected):
            lower_b = min(adj_dist[adj_v][x] for x in
                sig_selected)
            if lower_b > min_dist:
                best_v = adj_v
                min_dist = lower_b

        return best_v

    # Equivalent sum improvement
    def equi_sum_improv_impl(orig_v, orig_g, orig_sel, adj_func,
        counter, dist_func):
        sig_selected = set(orig_sel)
        sig_selected.remove(orig_v)

        global master_distance

        # Obtain distances from vertex in question
        base_dist = master_distance[orig_v]

        # determine the minimum distance between this vertex and
            the others selected
        min_dist = min(base_dist[x] for x in sig_selected - set([
            orig_v]))#base_dist.values())
        sum_dist = sum(base_dist[x] for x in sig_selected - set([
            orig_v]))

        # Container for distances of potential new points
        adj_dist = {}
        for adj_v in adj_func(orig_v, orig_g):
            adj_dist[adj_v] = master_distance[adj_v]
```

```
    # assume to begin that current point in question is best
    # if potential points has greater minimum distance from
       others move there
    best_v = orig_v
    for adj_v in rand_copy_list(set(adj_func(orig_v, orig_g))
        - sig_selected):
        lower_b = min(adj_dist[adj_v][x] for x in
           sig_selected)
        bound_adj_sum = sum(adj_dist[adj_v][x] for x in
           sig_selected)
        if lower_b > min_dist:
            best_v = adj_v
            min_dist = lower_b
        elif lower_b == min_dist \
             and bound_adj_sum > sum_dist:
            sum_dist = bound_adj_sum
            best_v = adj_v

    return best_v

# Simulated Annealing strict local improvement (Uniform
   random sampling)
def strict_improv_rand_anneal_impl(orig_v, orig_g, orig_sel,
   adj_func, counter, dist_func):
    sig_selected = set(orig_sel)
    sig_selected.remove(orig_v)

    global master_distance

    # Obtain distances from vertex in question
    base_dist = master_distance[orig_v]

    # determine the minimum distance between this vertex and
       the others selected
    min_dist = min(base_dist[x] for x in sig_selected - set([
       orig_v]))#base_dist.values())

    # Container for distances of potential new points
    adj_dist = {}
    for adj_v in adj_func(orig_v, orig_g):
        adj_dist[adj_v] = master_distance[adj_v]

    # assume to begin that current point in question is best
    # if potential points has greater minimum distance from
       others move there
```

```
    best_v = orig_v
    for adj_v in rand_copy_list(set(adj_func(orig_v, orig_g))
        - sig_selected):
        lower_b = min(adj_dist[adj_v][x] for x in
            sig_selected)
        if lower_b > min_dist:
            best_v = adj_v
            min_dist = lower_b
        elif best_v == orig_v:
            counter.obj -= 1
            random.seed()
            temp = random.randint(1, 100)
            if(temp < counter.obj):
                best_v = adj_v

    return best_v

# Simulated Annealing equivalent sum improvement(Uniform
    random sampling)
def equi_sum_improv_anneal_impl(orig_v, orig_g, orig_sel,
    adj_func, counter, dist_func):
    sig_selected = set(orig_sel)
    sig_selected.remove(orig_v)

    global master_distance

    # Obtain distances from vertex in question
    base_dist = master_distance[orig_v]

    # determine the minimum distance between this vertex and
        the others selected
    min_dist = min(base_dist[x] for x in sig_selected)#
        base_dist.values())
    sum_dist = sum(base_dist[x] for x in sig_selected)

    # Container for distances of potential new points
    adj_dist = {}
    for adj_v in adj_func(orig_v, orig_g):
        adj_dist[adj_v] = master_distance[adj_v]

    # assume to begin that current point in question is best
    # if potential points has greater minimum distance from
        others move there
    best_v = orig_v
```

```
    for adj_v in rand_copy_list(set(adj_func(orig_v, orig_g))
        - sig_selected):
        lower_b = min(adj_dist[adj_v][x] for x in
            sig_selected)
        bound_adj_sum = sum(adj_dist[adj_v][x] for x in
            sig_selected)
        if lower_b > min_dist:
            best_v = adj_v
            min_dist = lower_b
        elif pt_min == min_dist \
            and bound_adj_sum > sum_dist:
            counter -= 1
            if(random.randint(1, 100) < counter):
                sum_dist = bound_adj_sum
                best_v = adj_v

    return best_v

###

# Runs through complete cycle of position improvement until
    no more improved positions are found for
# the given heuristic
def run_cycle(GLWidget, graph, selected, improve_heuristic,
    adj_func, counter, dist_func):
        unvisited = set(rand_copy_list(selected))

        # improvement cycle
        while len(unvisited) > 0:
            v_iter = unvisited.pop()
            v_improve = improve_heuristic(v_iter, graph,
                selected, adj_func, counter, dist_func)

            if v_improve != v_iter and v_improve not in
                selected:
                # modify selected
                selected.remove(v_iter)
                selected.add(v_improve)
                unvisited = set(rand_copy_list(selected))

            #GLWidget.updateGL()



# original heuristic
```

```
def strict_local(GLWidget, graph, selected, adj_func,
    expansion, dist_func):
     counter = 0
     mod_selected = set(selected)
     GLWidget.graph = graph
     GLWidget.selected = mod_selected
     run_cycle(GLWidget, graph, mod_selected,
        strict_improv_impl, adj_func, counter, dist_func)

     return mod_selected

# remote weight heuristic
def remote_weight(GLWidget, graph, selected, adj_func,
    expansion, dist_func):
     counter = 0
     mod_selected = set(selected)
     GLWidget.graph = graph
     GLWidget.selected = mod_selected
     run_cycle(GLWidget, graph, mod_selected,
        equi_sum_improv_impl, adj_func, counter, dist_func)

# simulated annealing heuristic
def simulated_anneal(GLWidget, graph, selected, adj_func,
    expansion, dist_func):
     counter = ref_wrapper(75)
     mod_selected = set(selected)
     GLWidget.graph = graph
     GLWidget.selected = mod_selected
     run_cycle(GLWidget, graph, mod_selected,
        strict_improv_rand_anneal_impl, adj_func, counter,
        dist_func)

# reduced step size heuristic
def red_step_size(GLWidget, graph, selected, adj_func,
    expansion, dist_func):
     counter = ref_wrapper(1)
     mod_selected = set(selected)
     GLWidget.graph = graph
     GLWidget.selected = mod_selected
     run_cycle(GLWidget, graph, mod_selected,
        equi_sum_improv_impl, adj_func, counter, dist_func)
     while counter.obj != 0:
         graph, mod_selected = expansion(graph, mod_selected)

         global master_distance
```

```
        master_distance.clear()
        for v in graph:
            master_distance[v] = dist_func(v, graph)

        GLWidget.graph = graph
        GLWidget.selected = mod_selected
        run_cycle(GLWidget, graph, mod_selected,
           equi_sum_improv_impl, adj_func, counter, dist_func
           )
        counter.obj -= 1


# move to open area heuristic
def move_open_area(GLWidget, graph, selected, adj_func,
   expansion, dist_func):
    counter = ref_wrapper(2)
    mod_selected = set(selected)
    GLWidget.graph = graph
    GLWidget.selected = mod_selected
    run_cycle(GLWidget, graph, mod_selected,
       equi_sum_improv_impl, adj_func, counter, dist_func)
    while True:
        # determine minimum distance
        min_dist = len(graph)
        for node in mod_selected:
            dist_dict = dist_func(node, graph)
            tempset = mod_selected - set([node])
            local_min = min(dist_dict[x] for x in tempset)
            if(local_min < min_dist):
                min_dist = local_min

        # obtain vertex distance for graph
        non_selected = set(graph.keys()) - mod_selected

        min_non_sel_dist = {}
        for vert in non_selected:
            dist_dict = dist_func(vert, graph)
            min_sel_vert = min(mod_selected, key = lambda x:
               dist_dict.get(x))
            min_non_sel_dist[vert] = (min_sel_vert, dist_dict
               .get(min_sel_vert))

        # find vertex with maximum distance from selected
           node
```

```
            farthest_vert = max(min_non_sel_dist.keys(), key =
                lambda x: min_non_sel_dist.get(x)[1])

            # Only operate if swap is strict improvement
            max_dist_non_sel = min_non_sel_dist[farthest_vert][1]

            if max_dist_non_sel > min_dist:
                min_dist = len(graph)
                # determine worst vertex
                for v in mod_selected:
                    vert_dist = dist_func(v, graph)
                    temp_vert = min(mod_selected - set([v]), key
                        = lambda x : vert_dist.get(x))

                    if min_dist > vert_dist[temp_vert]:
                        min_dist = vert_dist[temp_vert]
                        worst_vert = temp_vert


                # Swap vertices from farthest to the worst
                    selected vertex
                mod_selected.remove(worst_vert)
                mod_selected.add(farthest_vert)
            else:
                break

            run_cycle(GLWidget, graph, mod_selected,
                equi_sum_improv_impl, adj_func, counter, dist_func
                )

class dancer_sim(object):
    def __init__(self, GLWidget, graph_type, heuristic_type,
        n, m):
        self.glref = GLWidget
        self.adj_func, self.graph, self.glref.rendermethod,
            self.expansion, self.dist_func = { \
            'Path' : (new_pos_path, gen_path(n), self.glref.
                paintPath, expand_path, path_equigraph_dist),
                \
            'Cycle' : (new_pos_cycle, gen_cycle(n), self.
                glref.paintCycle, expand_cycle,
                cycle_equigraph_dist), \
            'Grid' : (new_pos_grid, gen_grid(n,m), self.glref
                .paintGrid, expand_grid, grid_equigraph_dist),
                 \
```

```
            'Torus' : (new_pos_torus, gen_torus(n,m), self.
               glref.paintGrid, expand_torus,
               torus_equigraph_dist) }[graph_type]

        self.heuristic = { \
            'Local Strict Improvement' : strict_local, \
            'Remote Weight' : remote_weight, \
            'Simulated Anneal' : simulated_anneal, \
            'Reduced Step Size' : red_step_size, \
            'Move To Open Area' : move_open_area }[
               heuristic_type]

        self.glref.graph_type = graph_type
        self.glref.heuristic_type = heuristic_type
        self.glref.dim1 = n
        self.glref.dim2 = m

    def run(self, k, runs):
        self.glref.no_selected = k

        abs_min = len(self.graph.keys())

        f = open(self.glref.graph_type + '_' + self.glref.
           heuristic_type + '_select' + str(self.glref.
           no_selected) + '_' + str(self.glref.dim1) + 'by' +
            str(self.glref.dim2) + '.csv', 'a')

        global master_distance
        for v in self.graph:
            master_distance[v] = self.dist_func(v, self.graph
               )

        for i in range(1, runs + 1):
            selected = set(uniq_rand_obj_sublist(self.graph.
               keys(), k))
            self.glref.run_no = i

            # Run heuristic on the graph with the randomly
               selected vertices
            self.heuristic(self.glref, self.graph, selected,
               self.adj_func, self.expansion, self.dist_func)

            # determine minimum distance
            min_dist = len(self.graph)
            for node in self.glref.selected:
```

```
            dist_dict = master_distance[node]
            tempset = self.glref.selected - set([node])
            local_min = min(dist_dict[x] for x in tempset
                )
            if(local_min < min_dist):
                min_dist = local_min

        if(abs_min >= min_dist):
            abs_min = min_dist

            # I don't know what's going on with updateGL
               but it seems like maybe the first call
            # sets the double buffer because grabbing
               pixels with save after updateGL does not
            # yield expectation
            self.glref.updateGL()
            self.glref.updateGL()
            self.glref.saveImage("Grid", "Local", 5, 5,
               5, 1, min_dist)

        f.write(str(min_dist) + ',')

    # Close file
    f.close()
    # Clean distance map
    master_distance.clear()
```

REFERENCES

[1] Amakobe, G., Grabczewski, K., and Patel, T. Swarm behavior. Tech. rep., University of Delaware, Mathematics, 2006.

[2] Berger, B., and Shor, P. W. On the mathematics of virus shell assembly. Tech. rep., MIT, July 1994.

[3] Bowick, M., Cecka, C., Giomi, L., Middleton, A., and Zielnicki., K. Thompson problem. Website, August 2011. http://thomson.phy.syr.edu.

[4] Chen, D. Z., Hu, X., Huang, Y., Li, Y., and Xu, J. Algorithms for congruent sphere packing and applications. In *SCG '01 Proceedings of the Seventeenth Annual Symposium on Computational Geometry* (2001), ACM Press.

[5] Cohn, H., and Kumar, A. Universally optimal distribution of points on spheres. *Journal of the American Mathematical Society 20* (2007), 99–148.

[6] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction to Algorithms*. The MIT Press, 2001.

[7] Goldberg, M. Stability configurations of electrons on a sphere. *Mathematics of Computation 23* (1969), 785–786.

[8] Leon-Garcia, A. *Probability and Random Processes for Electrical Engineering*. Addison-Wesley, 1994.

[9] Nelson, D. R. Crystallography talk, 2007. http://cnls.lanl.gov/External/.

[10] Roth, R. *Introduction to Coding Theory*. Cambridge University Press, Cambridge, UK, 2006.

[11] Silvera, I. F. Multi-electron bubbles. Website, August 2011. http://silvera.physics.harvard.edu/bubbles.htm.

[12] West, D. B. *Introduction to Graph Theory*. Prentice Hall, 2001.