# CACHING IN ITERATIVE HILL CLIMBING

THESIS

Presented to the Graduate Council of
Texas State University-San Marcos
in Partial Fulfillment
of the Requirements

for the Degree

Master of SCIENCE

by

David Karhi, B.S.

San Marcos, Texas
December 2008

CACHING IN ITERATIVE HILL CLIMBING

Committee Members Approved:

_____

Dan Tamir, Chair

_____

Wuxu Peng

_____

Wilhelmus Geerts

Approved:

_____

J. Michael Willoughby

Dean of the Graduate College

## ACKNOWLEDGEMENTS

I would like to thank Dr. Dan Tamir for all of his invaluable help. I would also like to thank Dr. Wuxu Peng and Dr. Wilhelmus Geerts for taking time out of their busy schedules to provide me with feedback. Finally, I would like to thank the Department of Computer Science at Texas State University-San Marcos for allowing me to use their HPC.

This manuscript was submitted on October 28, 2008.

## TABLE OF CONTENTS

**Page**

## LIST OF TABLES

**Table** **Page**

## LIST OF FIGURES

# Chapter 1

# INTRODUCTION

Efficient utilization of resources is a problem that exists in all facets of life, including computer science. Because resources that are available but aren't utilized are wasted, it is important to examine all available resources and their potential for use when efficiency is a concern. Within the domain of computer science, there are two main resources available: computing speed, and storage space. Surprisingly, though computing speed is normally utilized very efficiently, storage space is sometimes utilized very poorly. This is especially true of algorithms that can benefit from the utilization of storage space, yet leave the resource completely unused.

Anytime algorithms are algorithms that can produce better results if they are given additional computation time [3, 5]. Similarly, anyspace algorithms are algorithms that can produce better results if they are given addition storage space [6]. One anytime algorithm that is widely used is the Iterative Hill Climbing (IHC) algorithm. Like all anytime algorithms, IHC produces better results if it is given additional computation time. However, IHC is plagued by redundancy for reasons that will be explained later in this thesis. Because it is subject to redundant computations, IHC can benefit from the use of storage space in order to maintain a list of computations that have already been

performed. This list, which can be implemented as a cache, can be used to prevent computations from being performed redundantly. By adding the ability to store data, IHC becomes an anyspace algorithm in addition to being an anytime algorithm, and a trade-off is created between adding more space to a problem and adding more time to a problem in order to produce the best solution while being as efficient with resources as possible.

Several researchers have studied time/space trade-offs in the past. One such example is Hertel and Pitassi, who studied time/space trade-offs within the context of satisfiability algorithms [7]. They found that the long-held assumption that there is a nearly linear trade-off between time and space is not always true within the domain of satisfiability algorithms, and at times the trade-off could become exponential. However, their findings do not translate into the domain of IHC. Additionally, Allen and Darwiche studied time/space trade-offs within the domain of probabilistic inference [4]. They found that their memory requirements could be significantly reduced with only a small increase in time. This minor optimization made exact inference possible in situations that were previously impractical. Unfortunately, their findings do not pertain to IHC. In fact, to the best of our knowledge, nobody has studied time/space trade-offs within the context of IHC.

The main motivation behind choosing to study and optimize IHC is that IHC is currently used in various fields of research [1, 8]. One such field is phylogenetics, which is the biological study of evolutionary relatedness among different groups of organisms. The Maximum Parsimony Problem (MPP), also called the Hamming Distance Steiner Tree Problem, is an NP-hard problem that is commonly used by biologists to estimate

phylogenies [8]. An NP-hard problem is a problem that is at least as hard as the hardest problems in NP, where NP is the set of decision problems that can be solved in polynomial time using a non-deterministic Turing machine [9]. MPP is typically solved with IHC, and there are cases where researchers were forced to wait for years for IHC to return an acceptable solution that may or may not be optimal. Ganapathy, et al. improved this situation by improving search techniques used with IHC in MPP, thus reducing the computation time needed. However, they did not consider adding a cache to further reduce the amount of computation time needed.

In summary, IHC and time/space trade-offs are current areas of study. However, time/space trade-offs have not been studied specifically in the context of IHC. Therefore, this thesis studies the time/space trade-offs that result from adding a cache to IHC. The Traveling Salesman Problem (TSP), which is explained in detail later, is used as a basis for this research.

## Chapter 2

## THE TRAVELING SALESMAN PROBLEM

TSP is an NP-hard problem that has been thoroughly researched [11, 12]. Given a set of cities and the distances between each of them, TSP seeks the shortest route that visits each city exactly once and returns to the starting point. A connection between two cities is called an edge, and a solution to TSP is a collection of edges that visit each city exactly once before returning to the starting point. Thus, each instance of a problem is a fully-connected graph that is weighted and undirected. In addition, each TSP solution, which is often called a tour, is a minimum spanning cycle.

TSP was chosen as a platform to study time/space trade-offs in IHC for several reasons. First, the problem is NP-hard, so it provides ample complexity with which to gage our results. Second, though TSP is incredibly complex, TSP is easy to analyze and understand. Third, TSP is a problem that is used in many different fields of research, and it is a problem that is known by most members of the scientific community. Finally, TSP translates well to MPP while being more widely understood than MPP, which means that people who may have no understanding of MPP can potentially help improve it by studying TSP.

Additionally, these experiments do not limit the TSP to planar graphs, nor do they

assume that the triangle inequality holds. The triangle inequality states that the direct path between two cities must be the shortest route. However, in the real world this is not always the case. For example, traffic congestion may result in a longer travel distance taking less time than a direct path. Thus, in this thesis, the most general TSP is used.

# Chapter 3

# HEURISTIC SEARCH

A heuristic is a method by which various alternatives can be ranked according to how desirable they are [2, 18]. A heuristic search is a search method that uses a search algorithm in conjunction with a heuristic in order to limit the size of the global search space that must be examined. Heuristic searches are often used when the global search space is so large that examining each element in a reasonable amount of time is not possible. Thus, the size of the search space is reduced by using a heuristic in order to determine which areas of the search space are worth examining and which are not.

The level of optimality of the solution returned by a heuristic search depends on the search algorithm used, the heuristic used, and the specific problem used. Search algorithms are general and can be used to solve a wide variety of problems. On the other hand, heuristics are specific to a given problem. In the following sections, several search algorithms are examined, as well as several heuristics that are specific to TSP.

## 3.1 Search Algorithms

There are several search algorithms that are commonly used in heuristic search. Some of the most popular search algorithms used are: A*, Hill Climbing (HC), IHC, and

Simulated Annealing (SA) [2, 13, 18]. These algorithms are discussed in the following sections.

### 3.1.1 A*

A* is a heuristic search algorithm that is guaranteed to find the shortest path from a given solution to an optimal solution as long as an admissible heuristic is used [2]. An admissible heuristic is a heuristic that is guaranteed to never overestimate the cost of reaching the optimal solution from the current solution. A* finds the shortest path from a given solution to an optimal solution by keeping a record of the shortest route from the initial solution to the current solution and using an admissible heuristic to determine which node will provide the shortest path from the current solution to the optimal solution.

Despite the fact that A* is guaranteed to find the shortest path from a given solution to an optimal solution, it is not often used because the amount of memory A* requires grows very quickly [2]. A* must maintain a record of the solutions that it has already examined as well as the solutions that it still needs to examine, which often grows exponentially. Therefore, A* is not feasible with large problems.

### 3.1.2 Hill Climbing

HC is a greedy search algorithm that is used mainly in the domain of optimization [1]. A greedy algorithm is any algorithm that takes the best immediate solution at any given step, and this is exactly what HC does [2]. There are two parts to the HC algorithm. First, a valid solution, called an initial configuration, is generated from the global search

space. Next, HC attempts to improve the solution by making some changes to the solution and taking the best new solution that it can find in the local search space. Each time an improvement is made, it is called a step. This improvement process continues until HC can no longer find a better solution, at which point the algorithm returns the best solution found.

Taking the best immediate solution at any given step may result in the best final solution. However, in practice it has been shown that greedy algorithms like HC often find a local minimum instead of finding the global minimum [2]. The reason for this is that HC only takes steps that improve the current solution, and can't take steps that are worse than the current solution. This results in HC getting stuck at a peak. A minimum is always found with HC, but there is no way to verify that the minimum discovered is a global minimum and not a local minimum without knowing what the global minimum is. Regardless, HC is widely used be cause it is easy to implement, it requires no memory, and it returns relatively good solutions.

### 3.1.3 Iterative Hill Climbing

IHC is a variation of HC that was developed in order to address the problem of HC getting stuck at a local minimum. In IHC, once a minimum is found, another initial configuration is generated from the global search space and the climbing process repeats. This process of generating initial configurations and improving them continues iteratively either until the global search space is exhausted or a desired number of iterations are completed. In this way, though IHC can still get stuck at a local minimum, the potential

to find numerous locally minimum solutions exists. Ideally one of those minimums is the global minimum.

Like HC, IHC is easy to implement, requires no memory, and returns relatively good solutions. However, many redundant solutions are examined in IHC. This is due to the greedy nature of the IHC algorithm. From any given intermediate solution, the next step taken will always be the same. Therefore, if a configuration is reached that had been found during a previous climb, every step from the current configuration up until the local optimum, where the climber restarts, will be redundant. Figure 1 illustrates this fact. In Figure 1, the red, green, and blue circles denote steps taken by three different climbers.

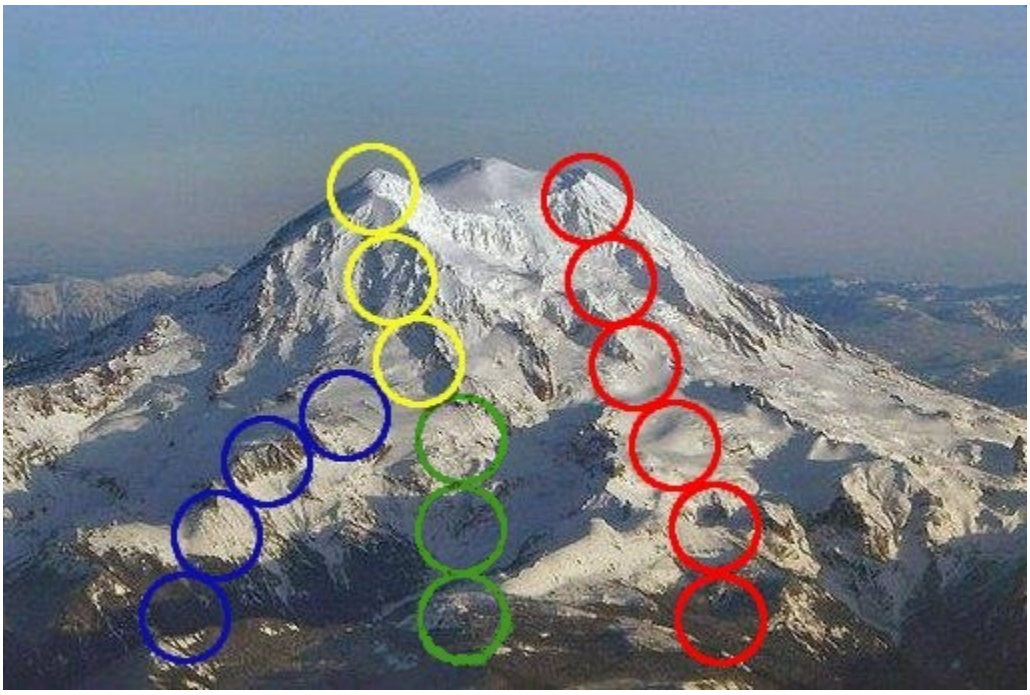

**Figure 1.** Iterative Hill Climbing

The blue and green climbers collide at a specific point and continue along the same path to the same local minimum. The redundant work performed is denoted by the yellow circles. Redundant work adds a lot of unnecessary computation time to a problem that is already extremely computationally complex. This is the situation that this thesis hopes to improve by adding a cache to IHC.

### 3.1.4 Simulated Annealing

SA is an algorithm whose name and inspiration come from the metallurgical process of annealing [13]. Like IHC, SA was developed in order to address the problem of HC getting stuck at a local minimum. SA is very similar to HC, except that SA allows the improvement process to replace a solution with a less-optimal solution under certain circumstances. These circumstances are determined by an acceptance probability function and a global time-varying parameter called temperature. The acceptance probability function determines the probability that any solution will be replaced with a different solution by taking into account their relative optimality as well as the current value of the temperature variable. The value of the temperature variable, which varies according to how much of the desired search space has already been examined, determines how greedy the next move will be. Later moves are relatively less greedy than early moves in order to prevent getting stuck at a local minimum.

The biggest problem with SA is that it is incredibly complex to implement. The success of SA depends heavily on the probability function that is used, and the probability function must be tailored especially for the specific problem that is being

examined. In contrast, IHC is very easy to implement and requires no detailed knowledge of the specific problem that it is analyzing. Thus, though SA can be much more effective than IHC at finding a global minimum, SA is not widely used because it is incredibly complex to implement effectively.

### 3.2 TSP-Specific Heuristics

Several heuristics have been studied in the context of TSP. Among them are the 2-Opt algorithm, the 3-Opt algorithm, the N-Opt algorithm, and the Lin-Kernighan (LK) algorithm [12].

#### 3.2.1 2-Opt

The 2-Opt algorithm removes 2 edges from a tour and replaces them with 2 edges whose sum is less than the sum of the edges that were removed [12]. This replacement is only performed if the new edges result in a valid tour being maintained. In this thesis, each of these exchanges is called a butterfly. Figure 2 shows an example of a butterfly. In this figure, solid lines denote edges and dotted lines denote any number of edges that is greater than 0. Assuming that $\omega_1+\omega_2 > \omega_3+ \omega_4$, the 2-Opt algorithm would remove $\omega_1$ and $\omega_2$ and replace them with $\omega_3$ and $\omega_4$.

#### 3.2.2 3-Opt

The 3-Opt algorithm is very similar to the 2-Opt algorithm, except that 3 edges are removed from the tour and replaced with 3 edges whose sum is less than the sum of the edges that were removed. Studies have shown that the 3-Opt algorithm produces

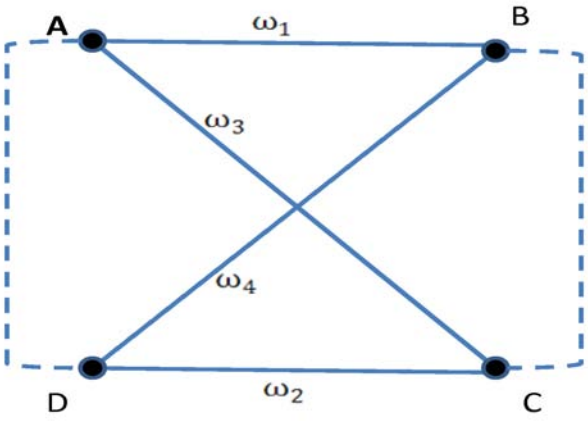


**Figure 2.** An Example Of A Butterfly

slightly better results than the 2-Opt algorithm [12]. However, the 3-Opt algorithm requires more computation time than the 2-Opt algorithm.

### 3.2.3 N-Opt

The 2-Opt algorithm and the 3-Opt algorithm are specific cases of what can generally be called the N-Opt algorithm, where N is the number of edges to swap at each step. Studies have shown that the quality of the results returned by N-Opt increase as N increases [12]. However, as N increases the computation time required increases as well. It is generally accepted that the increase in computational complexity between 3-Opt and 4-Opt is too great to justify the slight increase in the quality of the results that are returned. Therefore, in practice, N is rarely set higher than 3. In these experiments, 2-Opt is used for reasons that are explained later.

### 3.2.4 Lin-Kernighan

LK is very similar to N-Opt, except that LK determines which value of N to use at each step. Thus, some steps may swap 2 edges while other steps swap 4 or more edges. During each step, LK slowly increases the value of N until a point is reached where increasing N does not result in a significantly more optimal solution. At this point, LK swaps the edges.

Though LK has been shown to return better results than any of the other heuristics mentioned, it should be noted that LK is an extremely difficult algorithm to implement. For this reason, LK is rarely used [12].

# Chapter 4

# ITERATIVE HILL CLIMBING IN TSP

IHC in TSP can be divided into two parts: tour construction and tour improvement. Tour construction is the process of generating initial configurations while tour improvement is the process of improving the current configuration. In this chapter, the specific algorithms that are used for each part of IHC in order to perform experiments in this thesis are discussed.

## 4.1 Tour Construction

Several tour construction algorithms for the TSP have been thoroughly examined in the context of HC [12]. However, very few tour construction algorithms have been examined in the context of IHC. Most tour construction algorithms that are used in HC are not suitable for IHC because, in order for an algorithm to be suitable for IHC, the algorithm should be able to generate all possible valid initial configurations that exist in the global search space. If an algorithm cannot generate all possible initial configurations, then there is a risk that many good solutions, including the optimal solution, may not be able to be reached. Also, if an algorithm cannot generate all possible initial configurations, then there is the possibility that the algorithm will quickly run out of

unique initial configurations that it can generate and a lot of redundancy will result. Most of the algorithms that are used for tour construction in HC cannot generate all possible initial configurations without some modification, so they are not ideal for IHC. In fact, the only tour construction algorithm that has been used with IHC in TSP is the Random Restart (RR) algorithm, which randomly generates initial configurations by selecting a random order for a given set of cities during the tour construction phase. However, for the purpose of these experiments, an algorithm called Greedy Enumeration (GE) was developed. For the experiments conducted in this thesis, both RR and GE are used to generate initial configurations. In the following sections, both of these algorithms are discussed.

### 4.1.1 Random Restart

As stated above, RR generates initial configurations by randomly selecting an order for a given set of cities. One property of RR is that the initial configurations are widely dispersed throughout the domain. Due to the nature of IHC, initial configurations that are very similar may lead to the same minimum. Using a set of initial configurations that are dissimilar may avoid finding the same minimum and provide the opportunity to discover new minimums in a small number of iterations. On the other hand, there may be areas of a given problem that are known to contain poor solutions. RR doesn't provide the ability to control which areas of a problem will be explored. Also, with RR there is a possibility of generating redundant initial configurations. The probability of generating a redundant initial configuration with RR grows as more and more initial configurations are generated. As previously discussed, redundant initial configurations result in redundant

computations with no possibility of finding any solution that hasn't already been discovered.

### 4.1.2 Greedy Enumeration

GE is an algorithm that generates initial configurations for IHC in TSP. The algorithm takes, as input, a list of edges that are sorted in ascending order by weight. This list of edges denotes a specific instance of TSP. The first time the algorithm is run, it creates an initial configuration by assembling edges from the list of edges beginning with the shortest edge on the list. If adding a particular edge to the initial configuration would not result in the creation of a valid tour, then the edge is skipped and the next edge on the list is examined. There are two criteria that determine whether adding an edge to the initial configuration would result in the creation of a valid tour. First, if adding an edge to the tour results in a vertex that has more than two connections, then the edge cannot be taken. Second, if adding an edge to the tour results in a cycle being created that does not connect every vertex, then the edge cannot be taken. In this way, edges are assigned to the initial configuration until a valid tour is created.

Once the first initial configuration has been generated, subsequent initial configurations are generated by altering the previous initial configuration that was generated. Specifically, the second-largest edge in the previous initial configuration is removed and replaced with the next-largest edge on the edges list. Also, the largest edge in the previous initial configuration is removed. Then, starting with the smallest edge in this new initial configuration, the algorithm checks if each edge maintains a valid tour. If each edge maintains a valid tour, then the final edge that completes the tour is added and

the tour that results is the new current initial configuration. However, if there is an edge that violates the criteria for a valid tour, then the edge that violates the criteria for a valid tour is removed and replaced with the next-largest edge on the edges list. If the edge that violates the criteria for a valid tour is the largest edge on the edges list, then the next-smallest edge on the edges list is removed and replaced with the next-largest edge on the edges list instead. Once that is done, each edge in the new initial configuration that is larger than the edge that was just added is removed. Each edge that is removed in this way is replaced with the edge that is one step larger than the largest edge that is currently assigned to the new initial configuration. Next, the algorithm checks the edges in this new initial configuration to verify that they create a valid tour. If they do not, then the process repeats until a valid tour is created.

For example, consider a problem with 4 cities. There are 6 edges in a problem with 4 cities, and in this case they are numbered from 1 to 6 where 1 is the shortest edge on the list and 6 is the longest. Each time GE is run, it returns a set of edges. Imagine that it returns the set {1,2,4,5} the first time it is run. The first set of edges generated by GE always creates a valid tour, so the edges don't need to be checked. In this case, the second time it runs it returns the set {1,2,5,6}. Imagine that this second set of edges creates a valid tour, then the third set of edges generated will be {1,3,4,5}. Now, consider that including edge 4 in this set of edges results in an invalid tour. If this is the case, then the next set of edges generated will be {1,3,5,6}. If including edge 5 in this set of edges results in an invalid tour, then the next set of edges generated will be {1,4,5,6}. This process continues until each possible initial configuration is generated, or until a desired

number of iterations is reached.

Figure 3 illustrates the process of generating initial configurations with GE. Each horizontal bar in Figure 3 represents an edge on the sorted edges list. Red bars denotes edges that are assigned to the current initial configuration while black bars are unassigned. The figure progresses from left to right on the top row, then from left to right on the bottom row. Notice that the shortest edges on the list, the edges at the top of the list, are kept as long as possible while the largest edges are replaced.

**Figure 3.** Greedy Enumeration

Like RR, GE can always produce all possible initial configurations. However, unlike RR, GE produces no redundant initial configurations. This reduces the amount of

redundant computations, especially as the number of initial configurations generated grows larger in relation to the size of the problem domain.

### 4.2 Tour Improvement

As previously stated, several heuristics have been studied in the context of TSP. While all of the heuristics that have been studied can be used for the purpose of tour improvement, the 2-Opt algorithm is used for the purpose of tour improvement in the experiments that are performed in this thesis. The reasons for this choice are explained in the following section.

#### 4.2.1 2-Opt

The 2-Opt algorithm is used for the purpose of tour improvement in the experiments that are performed in this thesis for several reasons. First, the algorithm is relatively fast. In fact, the worst-case complexity for a single step in 2-Opt is $O(N^2)$ [12]. Second, the algorithm is very simple to implement. Finally, 2-Opt produces very good results, especially considering the fact that it uses a fairly small amount of computation time.

# Chapter 5

# CACHING

A cache is temporary storage that is used in computer science to decrease the amount of time needed to access data [10]. This decrease in access time can be due to one of two reasons: either the original data are located on a storage medium that takes longer to access than the cache, or the original data must be recalculated before being used while accessing the cache is quicker than recalculating the data. In our case, the latter is true. By storing data that has been computed, the need to compute the data again is either eliminated completely or the number of times that the data must be computed is reduced. It may not be possible to completely eliminate the need to compute data redundantly, since every cache is finite and storing every piece of information that is desirable to store is, in most cases, not feasible. Additionally, because cache capacity is limited, choices must be made that determine which data are stored and which data are not stored. These choices are determined by a cache replacement policy. A cache replacement policy is an algorithm that determines which data remain in the cache once the cache is full and which data are removed in order to place new items in the cache.

A cache is simply a collection of locations where data can be stored. However, there are 3 different ways of addressing those locations: direct mapping, set associative

mapping, and fully associative mapping [10]. Direct mapping means that each address points to only one location. If the location is occupied, then whatever is in that location is removed and the new data are stored in its place. Set associative mapping means that a single address can point to several locations. This is typically also referred to as N-way mapping, where N is the number of locations that a single address can point to. With set associative mapping, when all of the locations that an address can point to are occupied, a cache replacement policy must be used to determine which data the new data will overwrite. Finally, fully associative mapping means that every address can access every location in the cache. Again, once all locations are full, a cache replacement policy must be used to determine which data will be overwritten.

For the experiments performed in this thesis, set associative mapping is used. Direct mapping is too simple and doesn't allow for any cache replacement policies to be studied, and fully associative mapping is too complex to implement in many situations. Set associative mapping allows various cache replacement policies to be studied while still being relatively easy to implement.

An important concept that is used to study the potential effectiveness of implementing a cache in conjunction with a particular algorithm is locality of reference [15, 16]. Locality of reference is a statistical property of a given data stream that serves as a measure of redundancy. In other words, locality of reference measures the probability of seeing a particular piece of data in the near future that had been used in the recent past. Thus, strong locality of reference is typically equated with lower miss rates in the cache since the data that we are about to encounter should already be stored in the cache if it

was seen recently. Because it is such an invaluable source of data, locality of reference is studied in the context of IHC later in this thesis.

# Chapter 6

# EXPERIMENTS

In this chapter, the experiments that are performed in order to study time/space trade-offs are described and the data gathered from the experiments are presented. Each problem used in these experiments is generated by assigning random real-number weights between 0 and 1 to each edge of a fully-connected graph. The experiments are performed using an implementation that is described in this thesis, as well as a cache that is simulated in software. Four types of experiment are performed in this thesis. Initially, experiments are performed that study the temporal and spatial locality of a non-cached version of IHC. Next, experiments are performed that study miss ratios for various cache sizes using various cache replacement policies in conjunction with a cached version of IHC. After that, experiments are performed that study the difference in the number of instructions needed to run cached and non-cached versions of IHC. Finally, experiments are performed that compare the optimality of the solutions returned by cached and non-cached versions of IHC.

## 6.1 Iterative Hill Climbing Without a Cache

Before a cache can be implemented with IHC, several factors must be studied in

order to determine the best course of action. The most important of these factors is locality of reference [15, 16]. Two types of locality are studied in this thesis: spatial locality and temporal locality. Spatial locality measures the distribution of data items in reference to their location in the problem domain. Temporal locality measures the distribution of data items in reference to time. In other words, spatial locality measures the number of times a particular piece of data is encountered while temporal locality measures the amount of time elapsed between encountering the same data item.

Studying locality answers many questions pertaining to how the cache should be implemented, including which replacement policies to use and what the size of the cache should be. Without knowing the level of locality, implementing an efficient cache is much more difficult. In these experiments, good locality is considered to be the case where 80% of the data points fall within 20% of the domain.

### 6.1.1 Spatial Locality

In order to gather data about spatial locality, IHC is run without a cache on problems of different sizes using both RR and GE to generate initial configurations. Figure 4 shows the spatial locality for a problem with 8 cities using RR. In this case, the search was done exhaustively, meaning each tour in the problem domain is used as an initial configuration. It is easy to see that the best tour is encountered more often than any other tour in the domain. In fact, the best tour is encountered more than two times as often as the second-best tour and many times more often than most of the other tours in the problem domain. The explanation for this lies in the fact that IHC is a greedy algorithm. At every given step, the algorithm is attempting to move towards the optimal

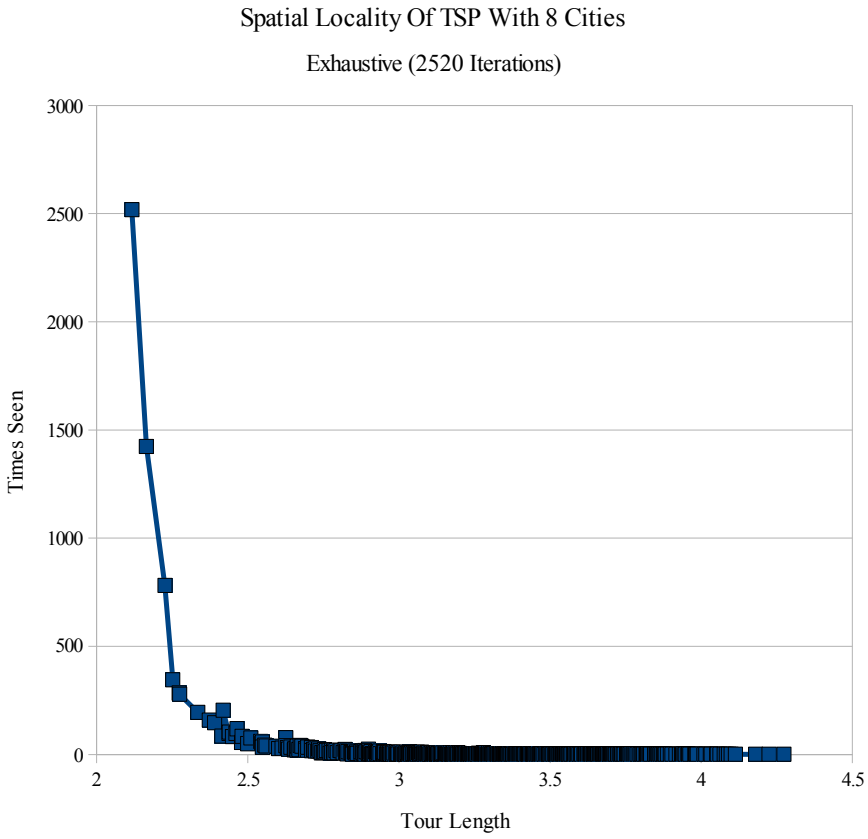


**Figure 4.** Spatial Locality Of TSP With 8 Cities Using RR

solution, so it is not surprising that the optimal solution is seen so often. It should be noted that this figure is considered to show good spatial locality.

Figure 5 and Figure 6 show the spatial locality of TSP with 10 cities and 12 cities, respectively. In these figures, RR is used to generate initial configurations. Also, note that both figures use the same number of iterations even though they use problems that have a different number of cities. In both cases, the most optimal tour is again seen much more often than any other tour in the domain. However, while Figure 5 displays good spatial locality, Figure 6 does not. With RR, as the problem domain grows and the number of

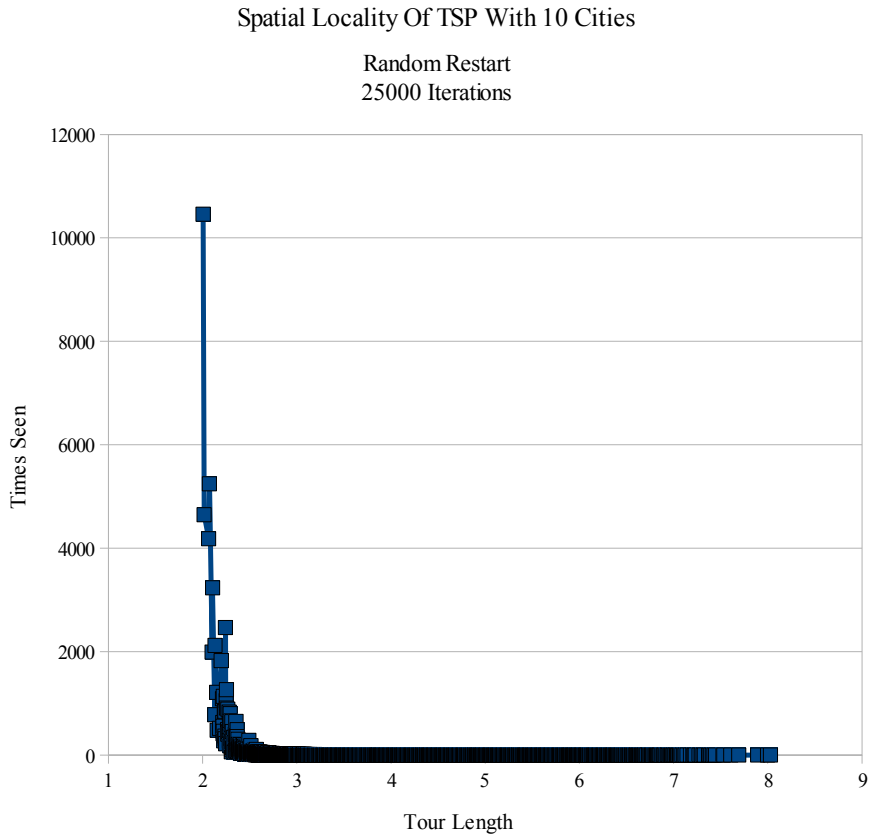


**Figure 5.** Spatial Locality Of TSP With 10 Cities Using RR

iterations remain the same, good spatial locality is quickly lost. This is because the initial configurations generated with RR are widely distributed so there are few collisions as the problem domain grows and the search space remains the same size. However, the fact that a very few number of tours are seen many times more often than any of the other tours in the domain when the level of locality is good suggests that the Least Frequently Used (LFU) cache replacement policy is a good cache replacement policy to examine once a cache is added to IHC. Once the cache is full, LFU removes the data item in the cache that has been encountered the least number of times and replaces it with the next

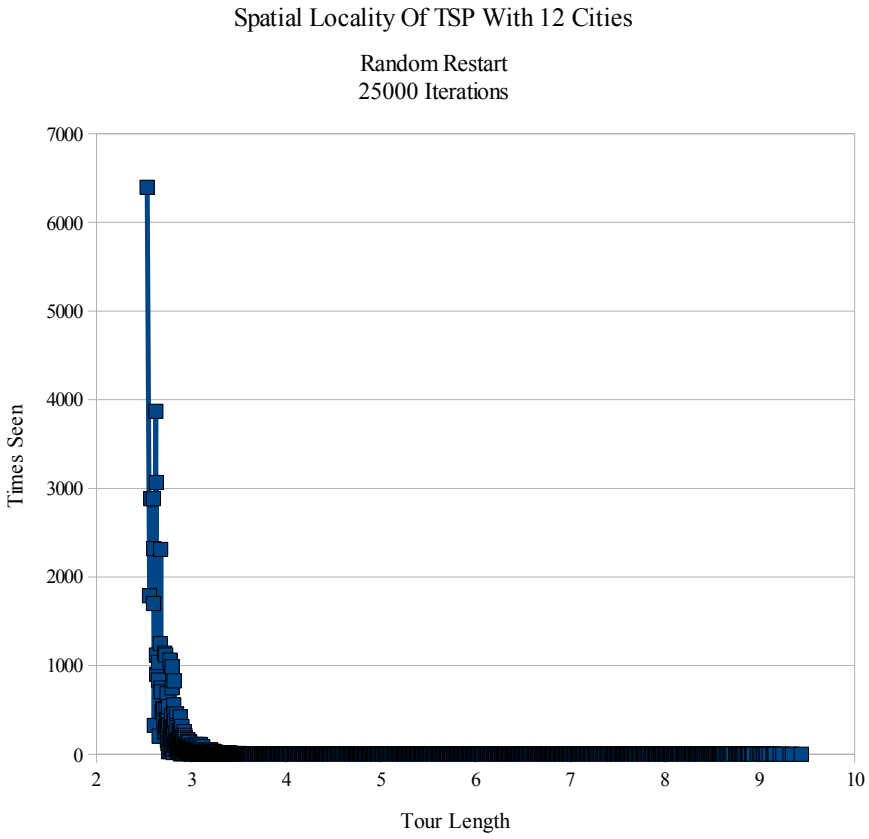


**Figure 6.** Spatial Locality Of TSP With 12 Cities Using RR

data item that is encountered. The idea is that data items which are encountered most often in the past will be encountered most often in the future.

Figure 7 shows the spatial locality for a problem with 12 cities using GE. Like RR, GE finds the same solutions many times, so LFU would be a good replacement policy to test with it as well. Notice that while RR did not show good spatial locality with 12 cities and 25,000 iterations, GE does. In fact, Figure 8 shows that GE maintains good spatial locality using 25,000 iterations even up to 18 cities. The fact that GE maintains good spatial locality with a wider range of problems than RR means that GE is more

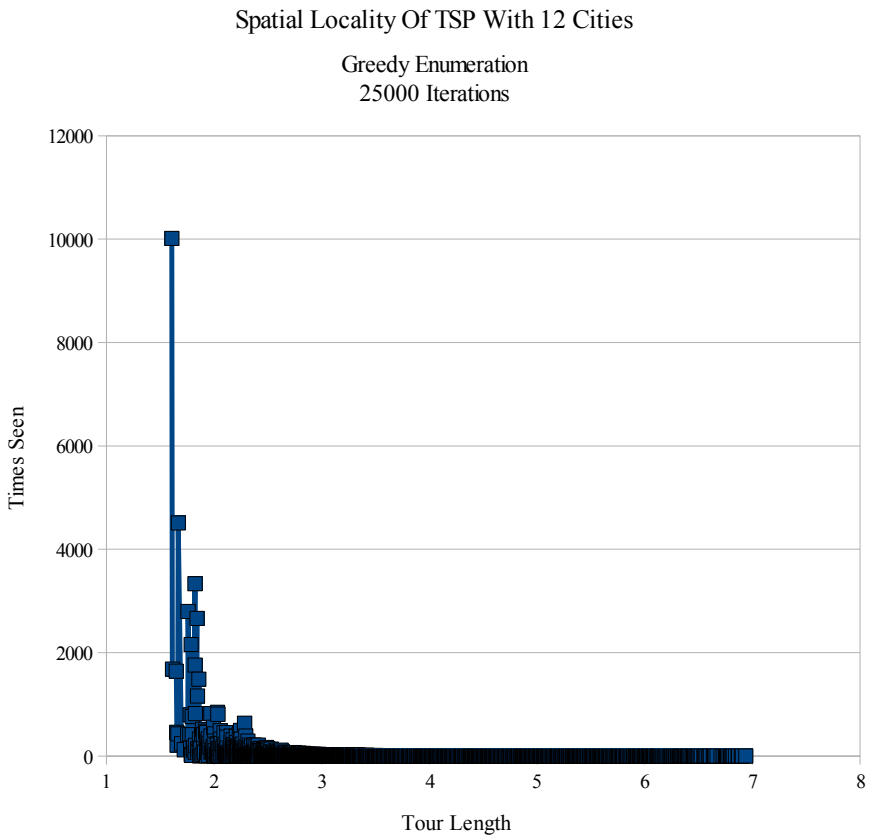


**Figure 7.** Spatial Locality Of TSP With 12 Cities Using GE

suited to the addition of a cache than RR. In any case, Figure 9 shows that GE loses good spatial locality around 20 cities using 25,000 iterations.

Thus, there are times when both RR and GE exhibit good spatial locality. This is especially true when the percentage of the problem domain that is examined is large compared to the size of the problem domain. However, as the percentage of the problem domain that is examined decreases, the level of spatial locality decreases as well with both RR and GE. GE maintains good spatial locality longer than RR, but both of them eventually lose good spatial locality.

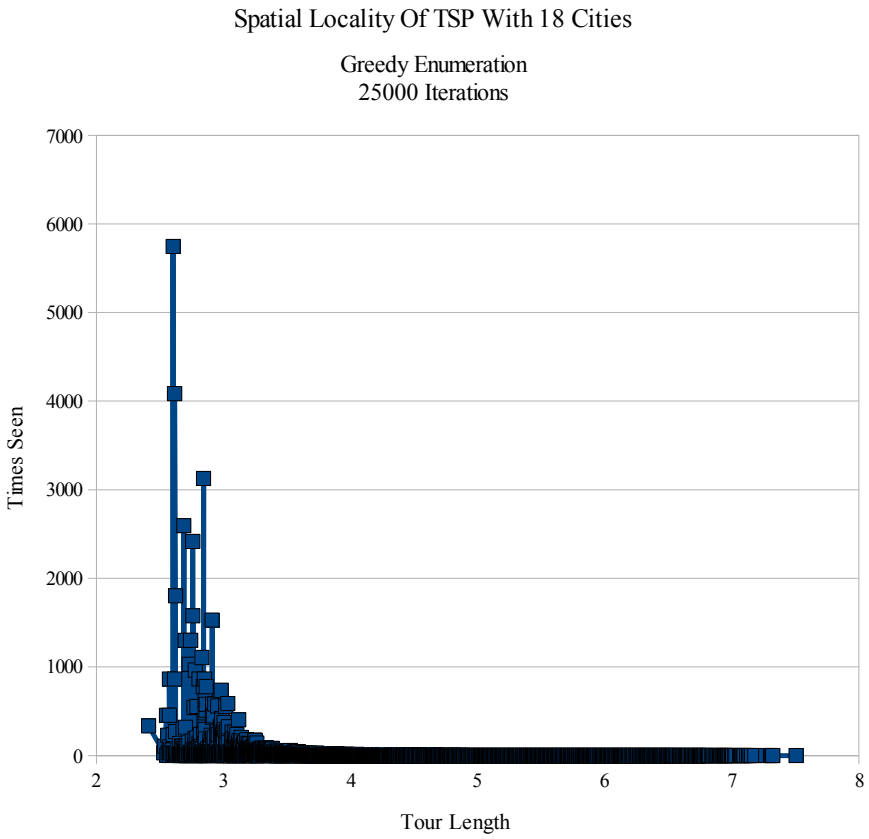


**Figure 8.** Spatial Locality Of TSP With 18 Cities Using GE

### 6.1.2 Temporal Locality

As mentioned above, temporal locality measures the distribution of data items in the data stream with respect to time. This means that the amount of time that elapses between collisions is studied, where a collision is defined as encountering a tour that has already been encountered in the past.

Figure 10 shows the temporal locality of TSP with 8 cities using RR. Notice that a small number of steps between collisions is seen much more often than a large number of

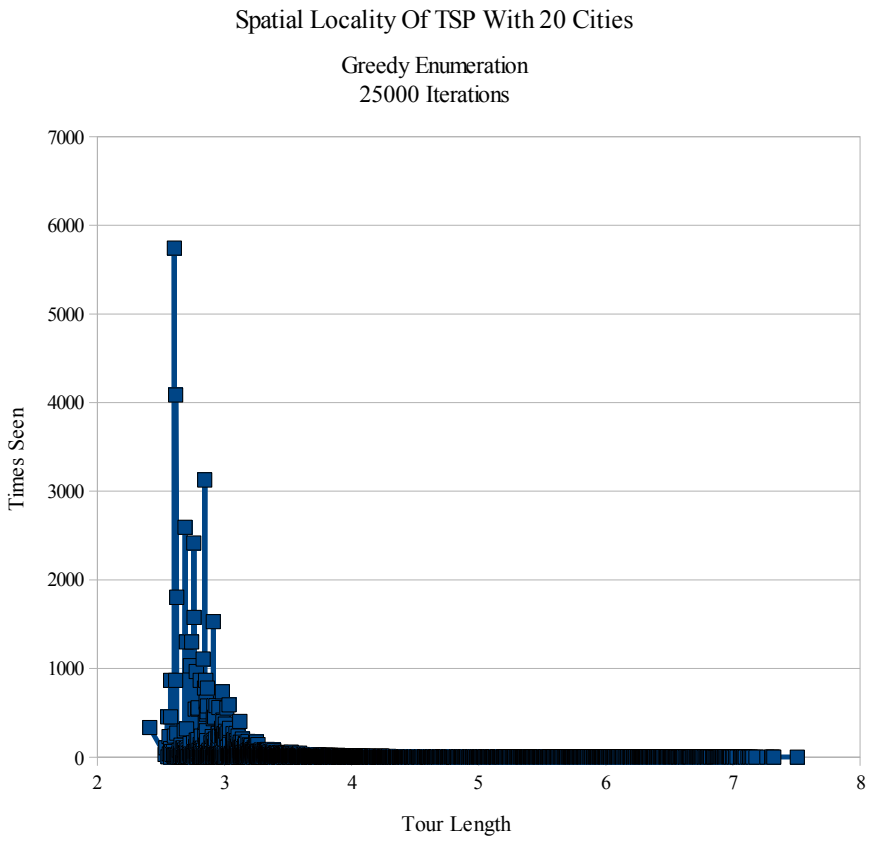


**Figure 9.** Spatial Locality Of TSP With 20 Cities Using GE

steps between collisions. This suggests that the Least Recently Used (LRU) cache replacement policy is a good replacement policy to examine once a cache is added to IHC. LRU keeps the data items in the cache that have been seen most recently by removing the least recently used data item once the cache becomes full and a new data item is encountered. This data does not guarantee that LRU will perform well, however it does warrant further research. Regardless, Figure 11 shows that RR begins to lose good temporal locality just as fast as it loses good spatial locality. As is the case with spatial locality, RR loses good temporal locality by the time it gets to problems with 12 cities

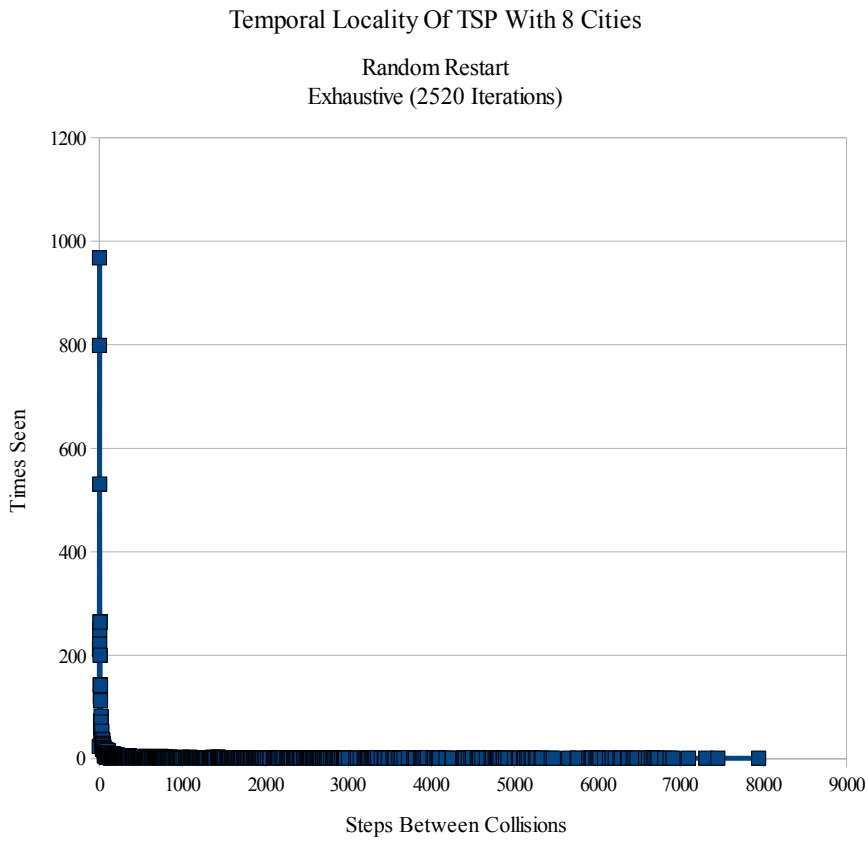


**Figure 10.** Temporal Locality Of TSP With 8 Cities Using RR

using 25,000 iterations. However, GE maintains good temporal locality even longer than it maintains good spatial locality. Figure 12 shows that GE maintains good temporal locality up to 26 cities using 25,000 iterations. Again, this shows that GE is better suited to the addition of a cache than RR because it maintains good temporal and spatial locality across a broad set of problem sizes. However, Figure 13 shows that by 28 cities with 25,000 iterations, GE loses good temporal locality as well. Regardless of the fact that both GE and RR exhibit poor temporal locality at times, the data suggest that there are definitely times when adding a cache would improve the time performance of IHC with

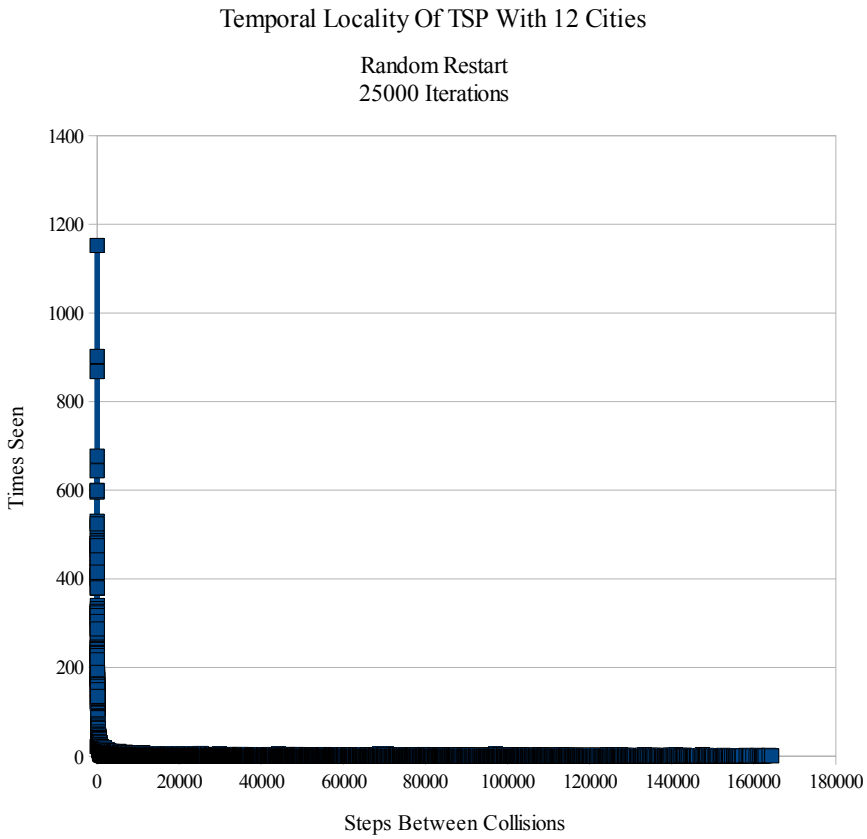


**Figure 11.** Temporal Locality Of TSP With 12 Cities Using RR

both GE and RR.

### 6.2 Iterative Hill Climbing With a Cache

After examining data pertaining to locality, it becomes clear that the addition of a dedicated cache to IHC is worth studying. As long as a good percentage of the domain is examined, the level of both temporal and spatial locality remains high, which should equate to good performance with a cache.

In order to convert a tour into a unique address that can be used by the cache, a

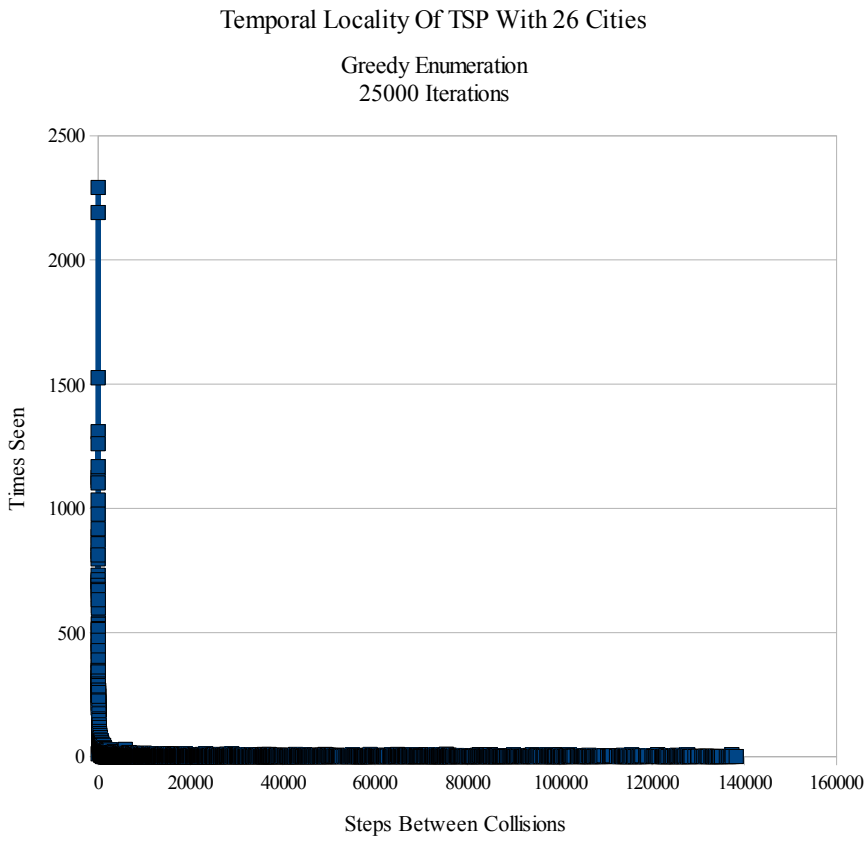


**Figure 12.** Temporal Locality Of TSP With 26 Cities Using GE

hashing algorithm must be used. In our case, the MD5 algorithm is used. MD5 is used because it converts data of any size into a fixed-sized 128-bit hash and the chance of converting two data values into the same hash is incredibly small [17]. It is important to convert the data into a fixed size because tours can be any length, and by being able to convert tours of any size into a fixed size it is possible to examine problems with a small number of cities as well as problems with a large number of cities. The process of converting a tour into an address consists of first converting the tour into an MD5 hash. Once the tour is converted into an MD5 hash, the number of bits necessary to address the

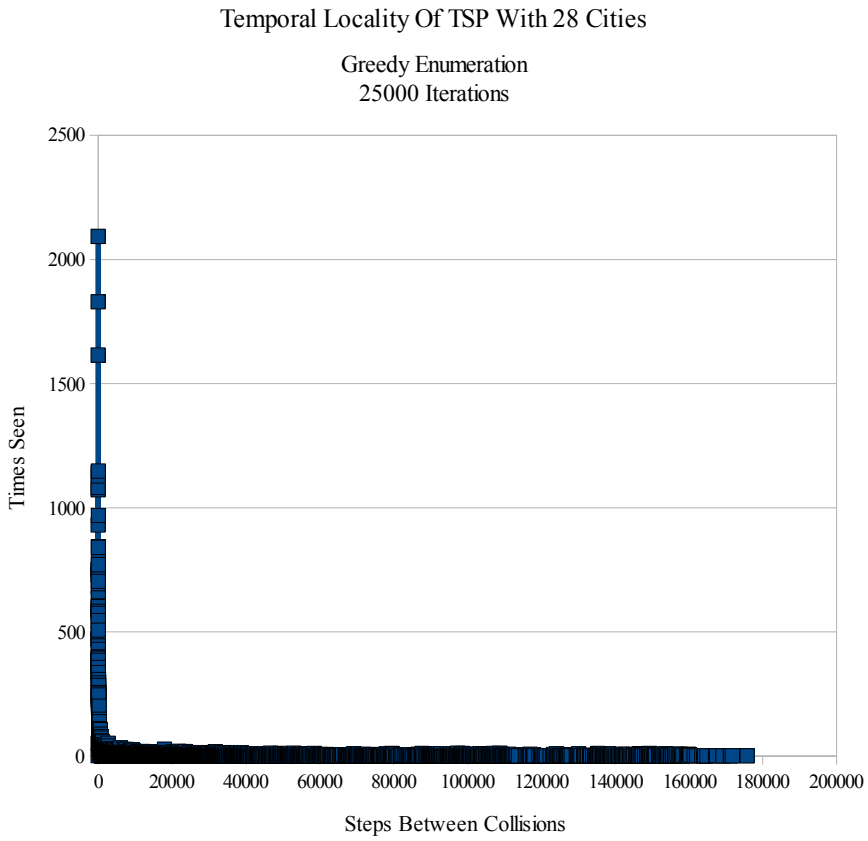


**Figure 13.** Temporal Locality Of TSP With 28 Cities Using GE

cache are removed from the most significant bits of the MD5 hash. Once this address is found, the remaining bits of the MD5 are stored in the cache. When a lookup is performed, the address is calculated the same way and the remaining bits are compared.

It was mentioned above that LFU and LRU are good replacement policies to study. Additionally, the random replacement policy is also a good policy to study in order to provide a good comparison. Thus, in order to study caching in IHC, a dedicated cache is implemented with IHC and data are gathered using LFU, LRU, and random replacement. Initially, the size of an optimal cache is not known. However, since it is

assumed that the cache described will be able to be implemented in hardware, reasonably standard cache sizes are studied in order to find an optimal cache size.

The first step towards finding the optimal size for the cache is to determine the level of associativity that the cache will be implemented with. As previously discussed, associativity is the number of storage locations that a single address can use. Since modern hardware typically uses either 4, 8, or 16-way associativity, these are the values that are used in the following experiments.

Once the level of associativity has been determined, the optimal number of sets must be found for each level of associativity used. The optimal number of sets for each level of associativity is found by running cached IHC with all 3 replacement policies using caches that have the same level of associativity but different numbers of sets. The smallest number of sets that has the lowest miss ratio is considered to be optimal for that particular level of associativity. Miss ratio is defined as the ratio of times that the cache was searched and the desired item was found to the total number of times that the cache was searched. Figure 14, Figure 15, Figure 16, Figure 17, Figure 18, and Figure 19 show the results of these experiments. Figure 14 shows the miss ratio for RR with 4-way associativity, Figure 15 shows the miss ratio for RR with 8-way associativity, and Figure 16 shows the miss ratio for RR with 16-way associativity. It can be seen in all three of these figures that the miss ratio eventually reaches a point where adding more sets does not result in a smaller miss ratio regardless of the replacement policy used. Though the number of sets that is optimal for each level of associativity is different, the total size of the optimal cache for each level of associativity is the same. Similarly, Figure 17 shows

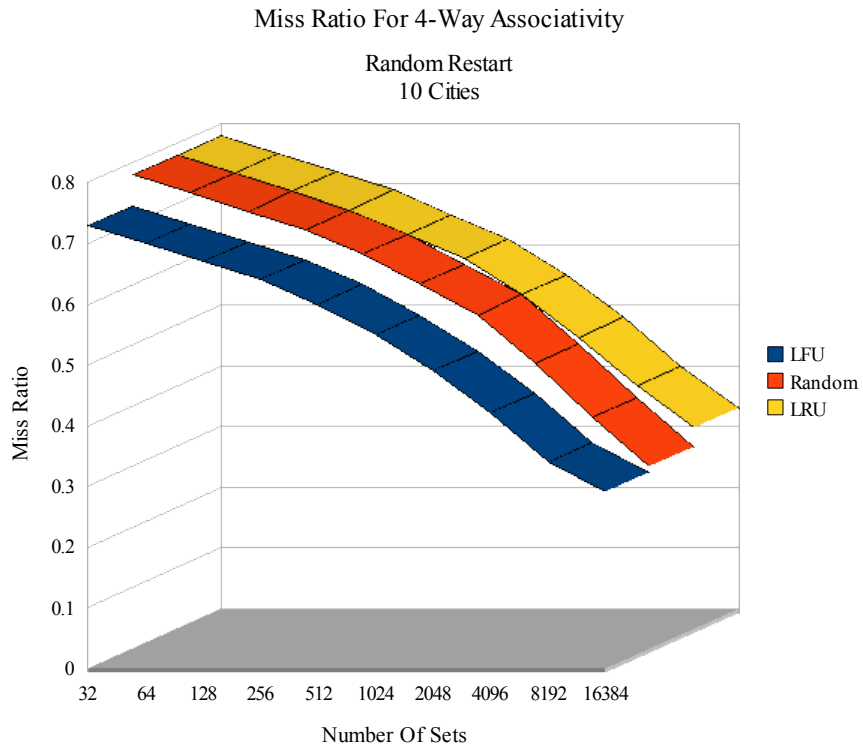


**Figure 14.** Miss Ratio For 4-Way Associativity Using RR

the miss ratio for GE with 4-way associativity, Figure 18 shows the miss ratio for GE with 8-way associativity, and Figure 19 shows the miss ratio for GE with 16-way associativity. Like RR, the miss ratio for GE eventually reaches a point where adding more sets does not result in a smaller miss ratio regardless of the replacement policy used. Also, like RR, though the number of sets that is optimal for each level of associativity is different, the total size of the optimal cache for each level of associativity is the same. In fact, the optimal number of sets for each level of associativity is the same regardless of whether GE is used or RR is used. Therefore, for the following experiments, 4,096 sets are used with 16-way associativity, 8,192 sets are used with 8-way

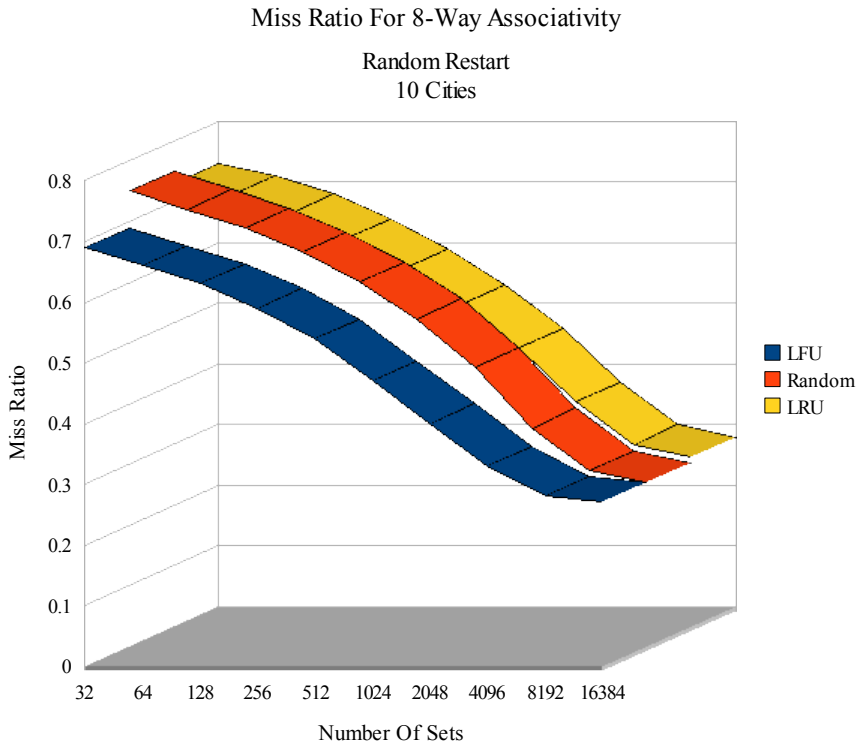


**Figure 15.** Miss Ratio For 8-Way Associativity Using RR

associativity, and 16,384 sets are used with 4-way associativity.

### 6.3 Execution Speedup

Having found the optimal cache size, non-cached version of IHC can be compared to cached versions of IHC in order to determine if the time performance of IHC can be improved by adding a dedicated cache. The estimated number of instructions required in each instance is calculated assuming that a dedicated cache is implemented in hardware. Based on our implementation, it is believed that a highly efficient implementation of the 2-Opt algorithm would require 40 instructions per butterfly. Additionally, it is assumed

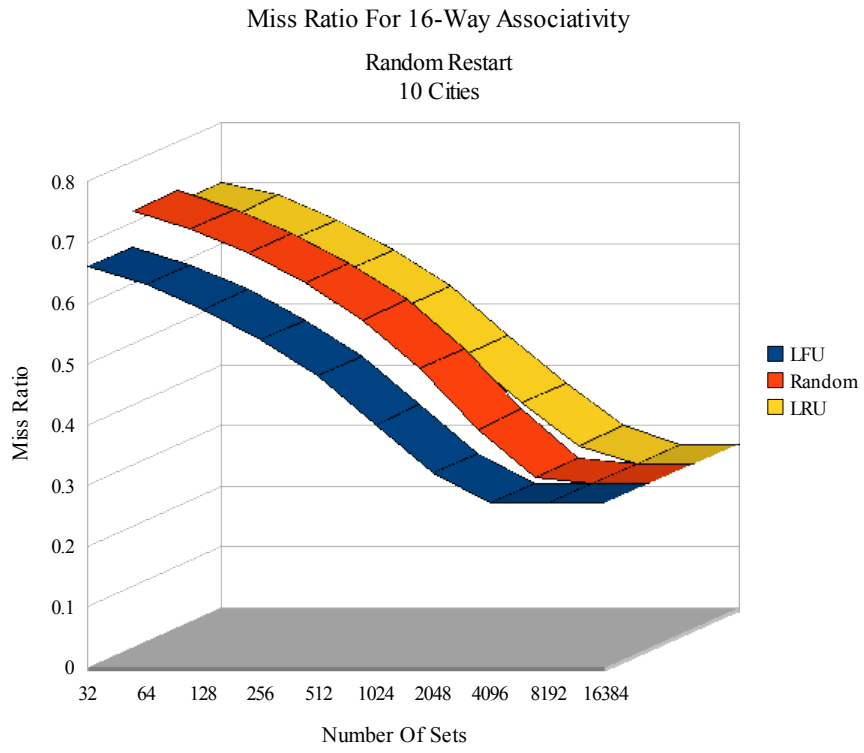


**Figure 16.** Miss Ratio For 16-Way Associativity Using RR

that a cache which is implemented in hardware requires 2 instructions per cache hit, 20 instructions per cache miss, and 55 instructions per hash.

Figure 20 shows a comparison of cached and non-cached versions of IHC using 25,000 iterations on a problem with 10 cities. In Figure 20, RR is used to generate initial configurations. It is easy to see that the cached versions require much fewer instructions than the non-cached version. In fact, the cached versions of IHC required about 70% fewer instructions than the non-cached version of IHC regardless of the replacement policy used. This is a significant improvement, especially considering the fact that the total size of the cache used in these is experiments is only about 1MB. With current

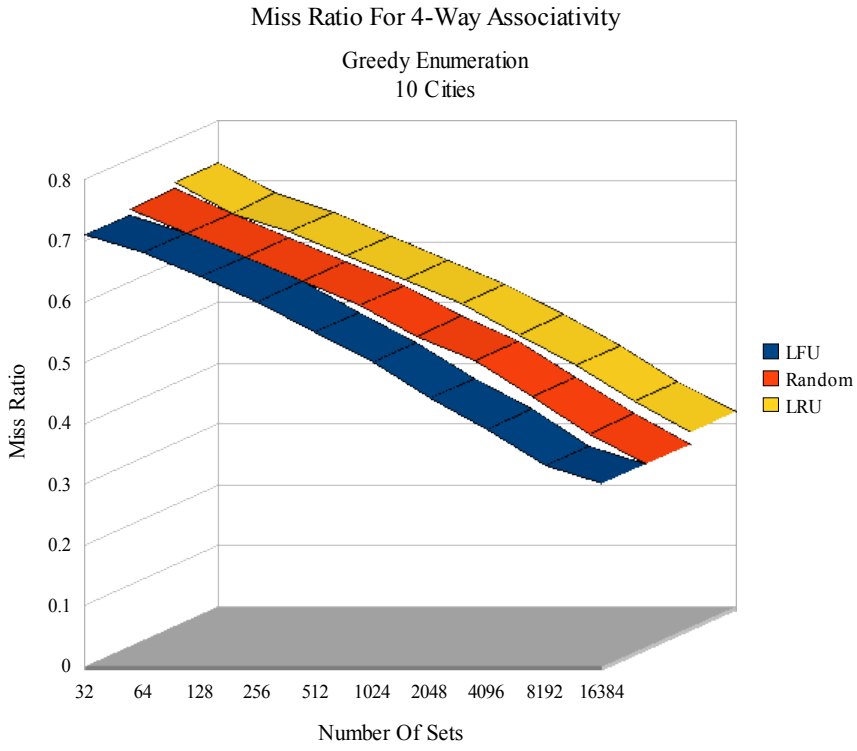


**Figure 17.** Miss Ratio For 4-Way Associativity Using GE

technology, a much larger dedicated cache is possible, and an even greater level of improvement may result from using a larger cache.

While these numbers appear very promising, the locality data show that the level of locality decreases as the percentage of the domain examined decreases. Therefore, data must be gathered that considers the possibility that only a small percentage of the domain will be examined. Figure 21 shows the estimated number of instructions required with the same number of iterations as Figure 20 but using more cities. It is clear that, in Figure 21, the improvement is smaller than the improvement in Figure 20. This is somewhat expected based on the locality data. Figure 22 shows that if the number of cities is

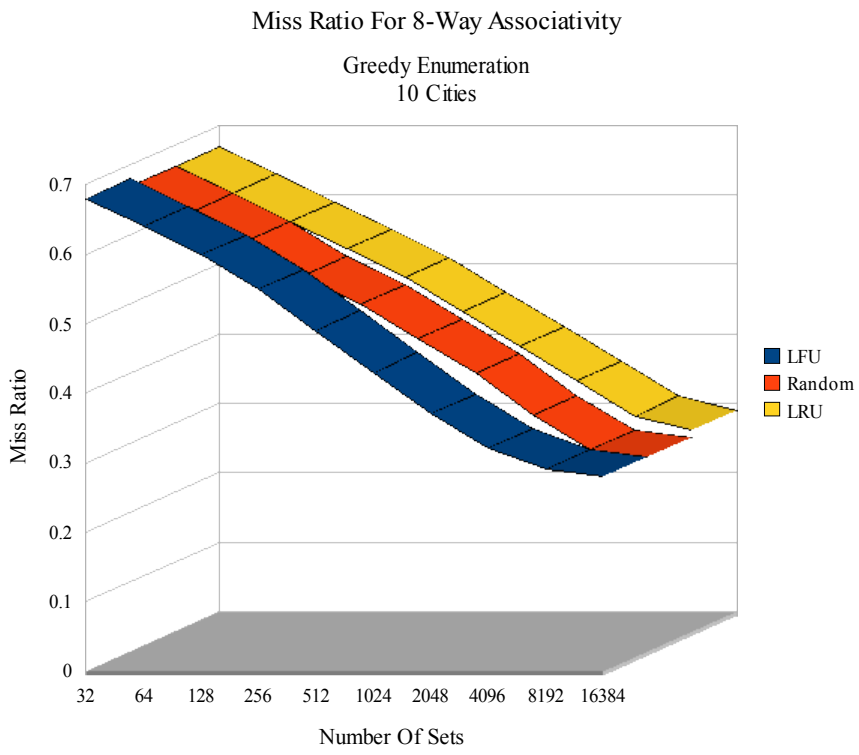


**Figure 18.** Miss Ratio For 8-Way Associativity Using GE

increased again while the number of iterations remains the same, the improvement becomes even smaller. In this figure, the improvement gained is less than 20%. This level of improvement probably isn't worth the extra effort required to implement a cache. Thus, as the locality data suggest, adding a cache to IHC when using RR is only beneficial when a large percentage of the problem domain is examined. When the percentage of the problem domain that is examined with IHC when using RR is relatively small, there is little or no benefit provided by the addition of a cache.

Though IHC using RR quickly loses locality, IHC using GE does not. Recall that while RR loses good temporal and spatial locality at around 12 cities with 25,000

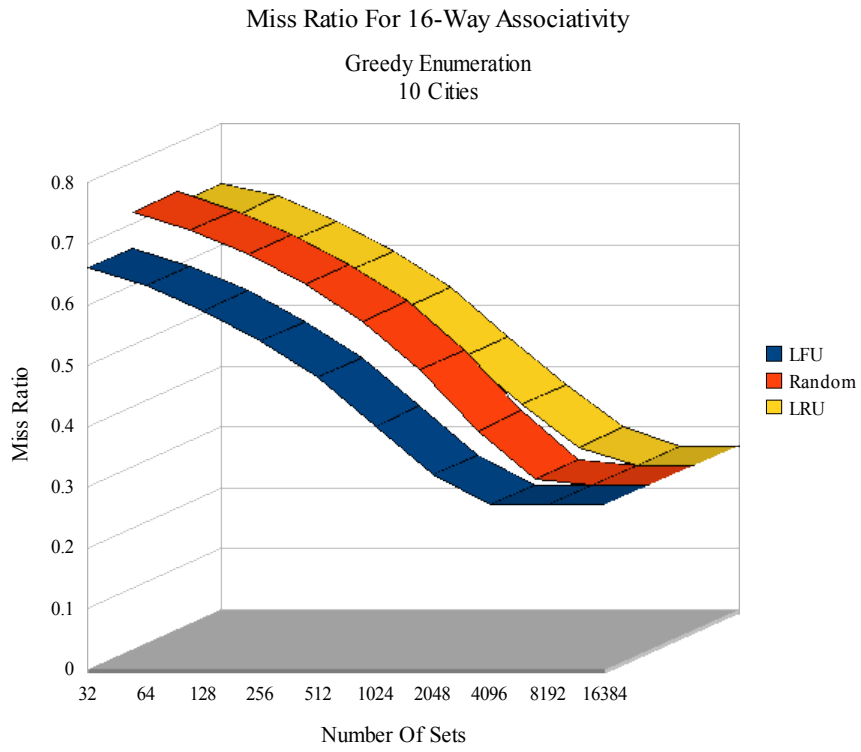


**Figure 19.** Miss Ratio For 16-Way Associativity Using GE

iterations, GE exhibits good temporal and spatial locality when using roughly twice as many cities with the same number of iterations. Thus, it is expected that GE will benefit from the use of a cache in a broader range of problems than RR. Figure 23 supports this hypothesis. Notice that, in Figure 23, 30 cities are used with 25,000 iterations. While the speedup with RR was only about 20% using 16 cities and the same number of iterations, GE exhibits an 80% speedup using a greater number of cities. In any case, the locality data imply that the speedup associated with adding a cache with GE will eventually disappear as the number of cities increases and the number of iterations remains the same. However, Figure 24 shows that adding 10 cities and using the same number of

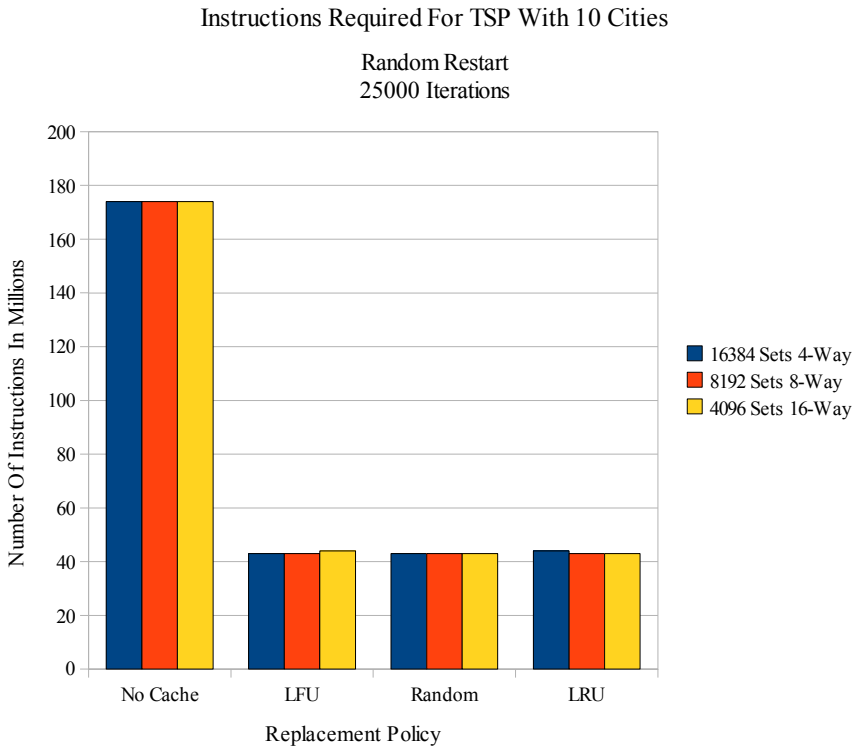


**Figure 20.** Instructions Required With 10 Cities Using RR

iterations with GE results in nearly the same percentage of speedup. While this doesn't mean that the percentage of speedup won't eventually reach a point where it is not worth the additional trouble of implementing a cache, it does show that it doesn't happen nearly as quickly as it happens with RR. Also, this does not mean that GE finds better solutions than RR. All it means is that, in a wider variety of problems, GE can benefit more from the addition of a cache than RR with respect to time performance.

### 6.4 Comparison of Solutions

In order to compare the quality of the results returned by the cached and non-

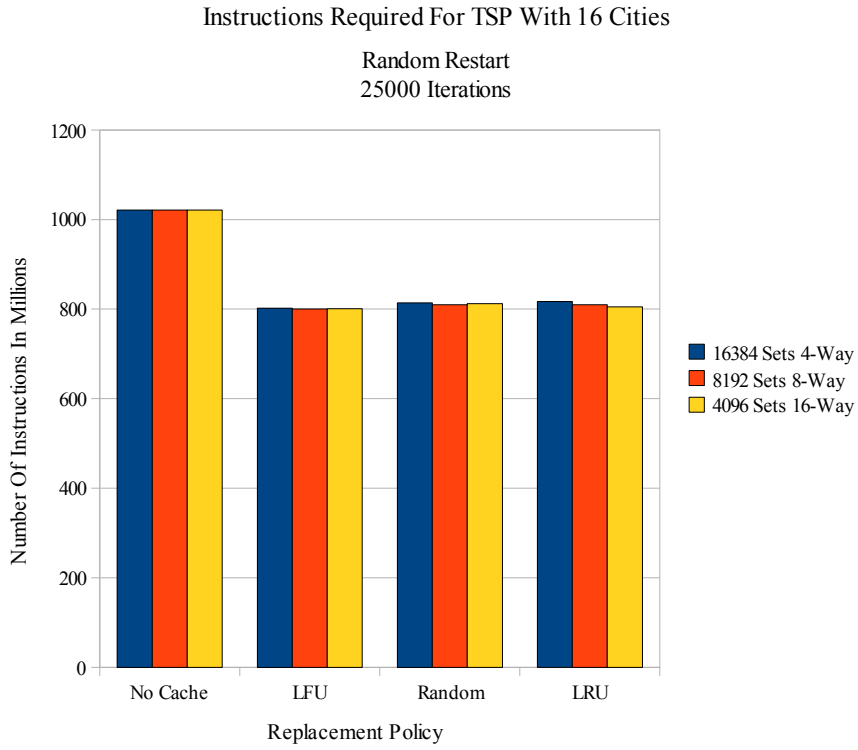


**Figure 21.** Instructions Required With 16 Cities Using RR

cached versions using GE and RR, additional data are gathered. These data are presented in Table 1. For this experiment, a problem with 50 cities is run 5 separate times using 25,000 iterations. The final results returned by both GE and RR after 25,000 iterations are presented, as well as the final results returned by GE after taking into account the speedup that is provided by adding a cache. It should be noted that, in this case, there was no speedup provided by adding a cache with RR, so those results are not presented. Table 1 shows that GE never produced a better result than RR, even after taking the speedup provided by the cache into account. This is somewhat surprising, considering the fact that the cached version of GE was able to examine nearly four-times as many tours as the two

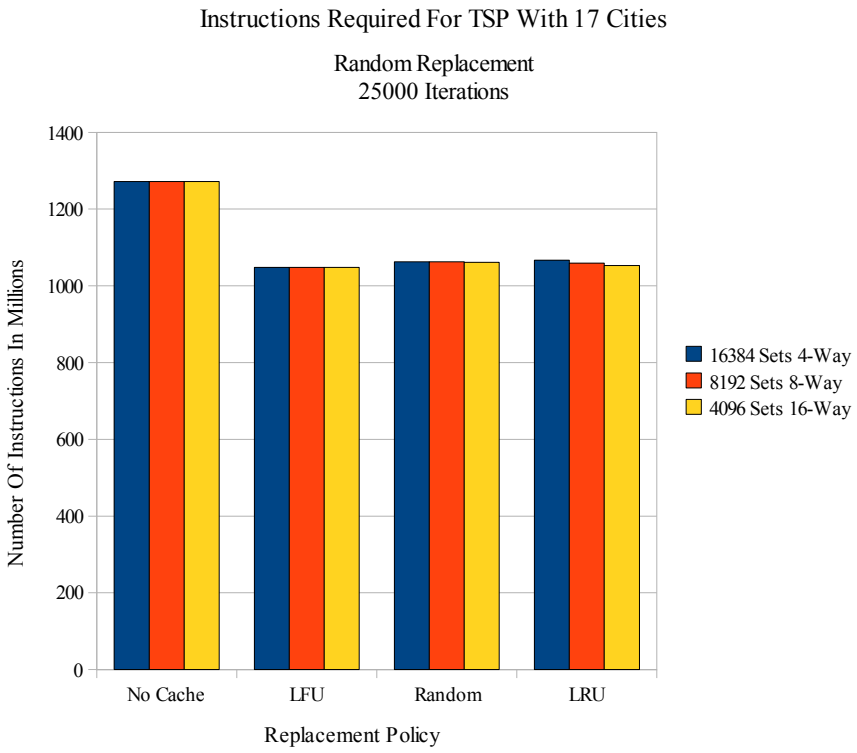


**Figure 22.** Instructions Required With 17 Cities Using RR

non-cached versions. Regardless, there was a case where the cached version using GE provided a better solution than the non-cached version using GE, which proves that a cached version using GE can return better solutions than a non-cached version of GE. Also worth noting is the fact that the cached version using GE, in a couple cases, returned the same solution as both non-cached versions while using only a quarter of the computation time.

### 6.5 Summary of Results

Many interesting things can be said about the data that are presented in this

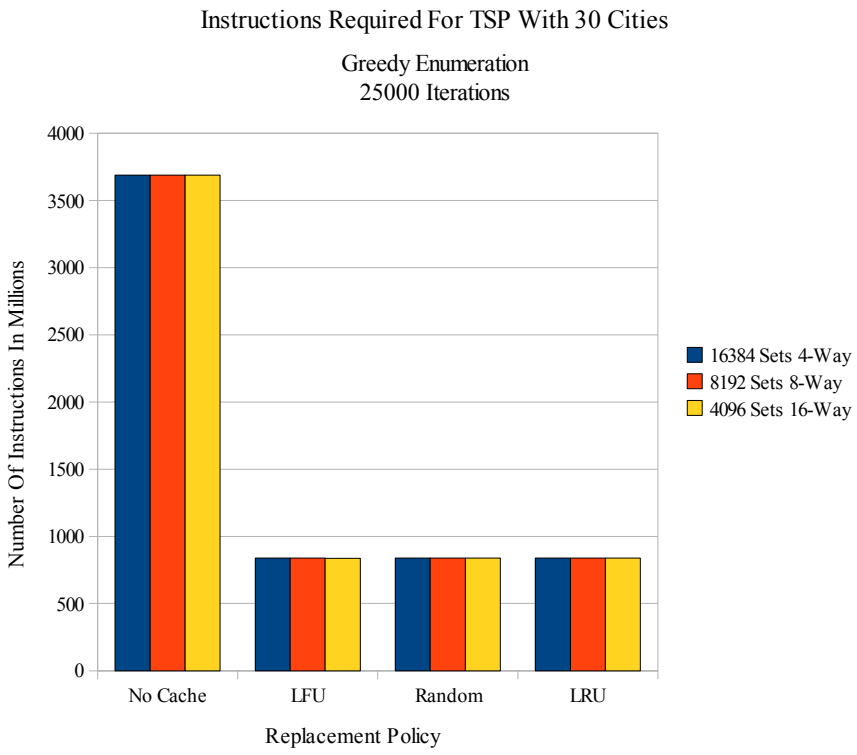


**Figure 23.** Instructions Required With 30 Cities Using GE

chapter. The locality data show that IHC exhibits good locality at times. While GE maintains locality across a broad variety of situations, RR does not. This finding is confirmed by the speedup data. The speedup data also confirm that GE is well-suited for the addition of a cache. In fact, the point at which GE stops benefiting with respect to computation time from the addition of a cache is not found in this thesis. This is very good, especially since the data that compare the results returned by the cached and non-cached versions of IHC using GE and RR show that a cached version of IHC using GE can return a better solution than a non-cached version of IHC using GE. These data also show that a cached version of IHC using GE can return the same solution as a non-cached

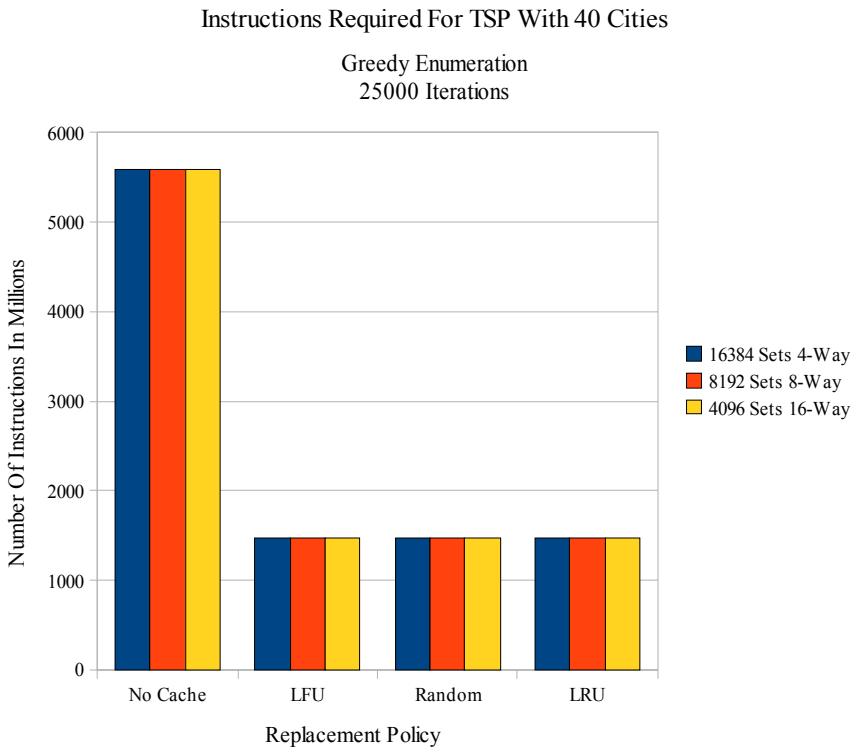


**Figure 24.** Instructions Required With 40 Cities Using GE

version of IHC using RR in a quarter of the time. Though these data have not been statistically verified, and more data need to be gathered before definitive conclusions can be drawn, it seems as though using a cached version of IHC using GE offers a lot of improvement over both a non-cached version of IHC using RR and a non-cached version of IHC using GE.

One final point of interest is the fact that all three cache replacement policies performed nearly identically. One possible reason for this is that the size of the cache used in this thesis is very large compared to the size of the problems used. However,

additional data need to be gathered in order to confirm or deny this hypothesis.

**Table 1.** Comparison of Solutions

| Problem # | RR | GE | Cached GE |
|---|---|---|---|
| 1 | 1.68740 | 1.68740 | 1.68740 |
| 2 | 1.87940 | 1.92543 | 1.92543 |
| 3 | 2.18191 | 2.25531 | 2.22627 |
| 4 | 2.43169 | 2.44108 | 2.44108 |
| 5 | 1.72302 | 1.72302 | 1.72302 |

# Chapter 7

# CONCLUSIONS

In many of the instances examined in this thesis, adding a cache to IHC improves the time performance of the algorithm regardless of whether GE or RR is used to generate initial configurations. The data presented in this thesis show that a version of IHC implemented with a cache can perform better with respect to time than a version of IHC implemented with no cache. At times, the cached version ran four-times faster than the non-cached version. This means that the same result is found in a quarter of the time by using a cache. In addition, this thesis shows that a cached version of IHC is able to produce better results than a non-cached version of IHC. On the other hand, there are cases where adding a cache to IHC results in a decrease in performance compared to a non-cached version. Thus, consideration should be used when determining whether or not a cache should be implemented with IHC in a particular instance. In most cases, a cache shouldn't be used with RR due to the fact that RR loses good locality so quickly. However, GE is very well-suited to benefiting from the addition of a cache. In fact, the point at which GE no longer benefits with respect to time from the use of a cache is not found in this thesis.

Additional research needs to be performed concerning caching in IHC. For

instance, the point at which GE no longer benefits with respect to time from the use of a cache needs to be studied. The locality data gathered in this thesis suggest that IHC using GE will eventually stop benefiting from the use of a cache. However, this point is not found in this thesis. Additionally, more information concerning the quality of the results returned by both GE and RR should be studied using both cached and non-cached versions of IHC. No cases where GE returned a better solution than RR, whether cached or non-cached, are found in this thesis. However, that does not mean that they do not exist. Finally, additional tests should be run in order to validate these results.

## LITERATURE CITED


[1] Xi, B., Liu, Z., Raghavachari, M., Xia, C., & Zhang, L. (2004). A smart hill-climbing algorithm for application server configuration. *Proceedings of the 13th international conference on world wide web*. (pp. 287-296).

[2] Russell, S. J., & Norvig, P. (1995). *Artificial intelligence: A modern approach.* Prentice-Hall.

[3] Zilberstein, S. (1996). Using anytime algorithms in intelligent systems. *AI Magazine*, 17(3), 73-83.

[4] Allen, D., & Darwiche, A. (2003). Optimal time-space tradeoff in probabilistic inference. *Proceedings of the 21st international joint conference on artificial intelligence.* (pp.969-975).

[5] McMahan, H., & Gordon, G. (2007). A fast bundle-based anytime algorithm for poker and other convex games. *Proceedings of the 11th international conference on artificial intelligence and statistics.* (pp. 323-330).

[6] Ramos, T., & Cozman, G. (2005). Anytime anyspace probabilistic inference. *International Journal of Approximate Reasoning*, 38(1), 53-80.

[7] Hertel, P., & Pitassi, T. (2007). An exponential time/space speedup for resolution. *Electronic Colloquium on Computational Complexity*, 46, 1-25.

[8] Ganapathy, G., Ramachandran, V., & Warnow, T. (2004). Better hill-climbing searches for parsimony. *Algorithms in bioinformatics.* (pp.245-258).

[9] Garey, M. R., & Johnson, D. S. (1979). *Computers and intractability: A guide to the theory of NP-completeness.* W.H. Freeman.

[10] Stone, H. S. (1993). *High-performance computer architecture.* Prentice Hall.

11] Applegate, D. L., Bixby, R. E., Vasek, C., & Cook, W. J. (2007). *The traveling salesman problem: A computational study.* Princeton University Press.

[12] Johnson, D. S., & McGeoch, L. A. (1997). The traveling salesman problem: A case study in local optimization. *Local search in combinatorial optimization.* (pp. 215-310).

[13] Kirkpatrick, S., Gelatt, C. D., & Vecchi, M. P. (1983). Optimization by simulated annealing. *Science*, 220, 671-680.

[14] Vose, M. D. (1999). *The simple genetic algorithm: Foundations and theory.* MIT Press.

[15] Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). *Compilers: Principles, techniques, and tools.* Addison-Wesley.

[16] Vanichpun, S., & Malkowski, A. M. (2004). Comparing strength of locality of reference – popularity, majorization, and some folk theorems. *23rd Annual Joint Conference of the IEEE Computer and Communications Societies.* (pp. 838-839).

[17] Wang, X., & Yu, H. (2006). How to break MD5 and other hash functions. *24th Annual International Conference on the Theory and Applications of Cryptographic Techniques.* (pp.19-35).

[18] Pearl, J. (1984). *Heuristics: Intelligent search strategies for computer problem solving.* Addison-Wesley.

# VITA

David Alan Karhi was born in Austin, Texas on February 15, 1984. He attended Texas State University-San Marcos and received the degree of Bachelor of Science in Computer Science with a minor in Mathematics in August 2006. Immediately after receiving his bachelor's degree, he entered the Graduate College of Texas State University-San Marcos.

Permanent Address: 11105 Chateau Hill

Austin, TX 78750

This thesis was typed by David Karhi.