A Comparison of Heuristic Algorithms to Solve the View Selection Problem

Ray Hylock
Management Sciences Department, University of Iowa
Iowa City, IA 52242, ray-hylock@uiowa.edu

A vital component to the success of a data warehouse is the proper selection of views to materialize to efficiently support online-analytical processing (OLAP) queries. Obviously, not all views can be materialized and stored due to space and update time constraints. Thus, the goal of the view selection problem (VSP) is to find a set of views to materialize such that the query response time is minimized given the constraints on space and update time. As this problem is intractable, approximation algorithms are employed. In this paper, we compare several well known heuristic algorithms on an existing health care data warehouse. These algorithms include random search, first improvement local search, best improvement local search, a randomly populated genetic algorithm, and a genetic algorithm populated with local search solutions. Our experiments show genetic algorithms seeded with the results of an initial local search to be superior to all other methods tested.

Key words: view selection problem; data warehousing; heuristics; decision support; genetic algorithms; local search

1. Introduction

For a business to maintain its competitive advantage it is imperative for it to be able to analyze its information sources for trends; typically, this is done through the use of decision support systems and OLAP queries. Likewise, clinical decision support systems can be used to mine healthcare data, leading to improvements in the cost, efficiency, and quality of patient care. These information sources store historical data and are therefore, very large in size and continue to grow over time. These decision support queries are very complex, taking into consideration multiple information sources and various aggregations and combinations, making them extremely time consuming to perform; sometimes too time intensive (Elmasri & Navathe, 2007; Ullman & Widom, 2008). A commonly used technique to reduce the overall query response time, is to precompute and store, in the form of materialized views, the most frequently used queries in a data warehouse.

A data warehouse is a database that collects and stores data from the aforementioned information sources. The goal is to create a single, integrated system by extracting, transforming, and loading (ETL) data from multiple, heterogeneous information sources into a consistent view of the data (Theodoratos & Sellis, 1999). Data warehouses are mainly intended for decision support applications and are optimized for retrieval instead of routine transaction processing (Elmasri & Navathe, 2007). More formally, a data warehouse can be characterized as a subject-oriented, integrated, nonvolatile, time-variant collection of data used to support decision making. Each characteristic is as follows (Inmon, 2002; Kimball & Ross, 2002):

- Subject-oriented: The data warehouse is organized around a key subject (e.g., customers or patient visits)
- Integrated: All data in the data warehouse conforms to one standard. For example, in one database, male and female may be recorded as “m” and “f” whereas in another, “male” and “female”. When records are added to the data warehouse, the domain (or set of values) of each attribute will be checked to make sure all values adhere to the standards imposed by the data warehouse. In this case, if the acceptable values were set to {“m”, “f”}, the second system’s values would be converted from “male” to “m” and “female” to “f”. This ensures data integrity and improves query performance.
- Nonvolatile: The data is loaded and refreshed in a single process and is not updatable by end users. This means new data is generally uploaded in batches and users do not have the ability to edit or delete data.
- Time-variant: Data is stored as a time series so one can analyze the data across time to locate, e.g., trends. The storing of this information is done in advance of the queries. This way, queries posed to the system can be answered directly by the data warehouse instead of being reformulated for each data source and transmitted for execution.

Figure 1 illustrates a typical data warehouse architecture (Gardner, 1998; Gupta & Mumick, 2005; Widom, 1995). As we have covered, the data warehouse integrates information from external information sources to support the query answering process. Breaking it down by components, the monitors monitor the information sources and notify the integrator when a change has been made. The integrator, during allotted times, is responsible for integrating the data from the information sources into the data warehouse. At the top are the users who directly query the data warehouse instead of the individual information sources store historical data, and continue to grow over time.
sources. The users, as mentioned earlier, are querying materialized views. Through the proper selection of table joins and attribute aggregations, a materialized view can significantly improve the speed of the analytical process; from hours or even days to seconds or minutes. Therefore, one of the most important issues in data warehousing is the selection of views to materialize.

Various random search techniques such as random sampling (for a specified period of time, randomly choose a solution and keep the best), iterative improvement (a local search technique allowing only downhill moves for a specified time limit; uses multi-start and keeps the best solution), simulated annealing (similar to iterative improvement, simulated annealing allows uphill moves; also based on a time limit), and two-phase optimization (combines iterative improvements ability to quickly find an area with many local minima with simulated annealing which is better at walking the solution landscape), were tested by (Kalnis, Mamoulis, & Papadias, 2002). Although the test sets from the data warehouse were limited to 12 dimensions (in order to find an optimal solution for comparisons), they experimented with both VSP formulations: the traditional model and the view-maintenance model which minimizes the query response time given space and update time constraints. Their results show that simulated annealing and two-phase optimization are generally better.

A variant of the A* heuristic and an inverted-tree greedy algorithm were compared by (Gupta & Mumick, 1999, 2005) on the view-maintenance formulation. They conclude that the inverted-tree greedy algorithm outperforms the A* heuristic, however, since both of these heuristics were created by the authors, it is difficult to compare their findings with other, more frequently applied heuristics.

This selection process is called the view selection problem (VSP) and is defined as the selection of a set of views (queries) to materialize in order to minimize query response time under a set of given constraints. In this paper, we present various heuristic approaches to solve a VSP instance from a clinical data warehouse developed in an earlier project (Hylock, Street, Lu, & Currim, 2008).

The rest of the article is organized as follows. The remainder of this section gives a brief summary of previous work. Section 2 provides a more thorough background and definition of the VSP. Section 3 describes the heuristic algorithms used. Section 4 presents our results. Finally, we wrap up with our concluding remarks in §5.

1.1. Previous Work
Several articles have been proposed in which genetic algorithms (GA) have been used to solve the VSP. (Zhang & Yang, 1999) take this approach using the traditional VSP model in which the objective is to minimize the query response time plus the update cost, constrained only by the allotted size. They compared GA to a simple greedy algorithm (which it outperformed) to show GA can be used effectively on the VSP. Similarly, (Horng, Chang, Liu, & Kao, 1999) use a GA variant called genetic local search (GLS) and again, formulate the VSP according to the traditional model. GLS builds an initial population through a local search and performs a local search on each subsequent generation (we will further discuss the details of implementing genetic algorithms in §3.4). No comparison with any other algorithm is made. The simple goal of the paper, as with (Zhang & Yang, 1999), is to prove the viability of GA on the VSP.
2.1. Test Queries
Using the bridge group formula and multidimensional design from (Hylock et al., 2008), our test data warehouse has 3,600 possible views, resulting in a runtime of $O(2^{3,600})$. When reviewing the literature, one will notice that hierarchies are not generally included in the formulation of VSP algorithms for the sake of simplicity. Instead, it is assumed that the lowest level (e.g., day) will be materialized. Following this rule, our test dataset will be reduced to 64 views (following the formula $2^6$ from (Hylock et al., 2008)). From the 64 potential views, 50 queries were manually generated insuring no query could be merged with another. That is, if $q_1$ can be answered from $q_2$, then there is no need to include $q_1$. This, however, is still too large to solve optimally in a reasonable amount of time, thus heuristics are still applicable.

2.2. VSP Formal Definition
The VSP, can be formally defined as follows: given a relational data warehouse schema $R$ containing a set of dimensions $D = (d_1, d_2, ..., d_k)$, maximum storage space $S$, maximum update cost $U$, and a workload of queries $Q = (q_1, q_2, ..., q_m)$, select a set of views $M$ such that $M \subseteq L$ (where $L$ is a cube lattice consisting of $|Q|$ vertices) to materialize, whose combined size is at most $S$ and whose combined update cost is at most $U$. Each view $v \in L$ has an associated query frequency $f_v$, an update frequency $g_v$, and a reading cost $r_v$ (which is the number of tuples in $v$). The cost of answering a query $q$ corresponding to a view $v$ is either the number of tuples in $v$ if the view is materialized or the number of tuples in its smallest materialized ancestor (denoted $q(v, M)$). The cost of updating a materialized view $v \in M$ is a percentage $p$ of the number of tuples in its smallest materialized ancestor (denoted $u(v, M)$).

The cost of answering all queries $Q$ in the lattice $L$ given the set of materialized views $M$ is:

$$Q(L|M) = \sum_{v \in L} f_v q(v, M)$$

The storage cost of all materialized views $M$ is:

$$S(M) = \sum_{v \in M} r_v$$

The update cost of all materialized views $M$ is:

$$U(M) = \sum_{v \in M} g_v u(v, M)$$

The model for this problem is as follows:

\[ \text{MINIMIZE: } Q(L|M) \]

\[ \text{SUBJECT TO: } S(M) \leq S \]
\[ U(M) \leq U \]
\[ U(M), S(M) \geq 0 \]

3. Algorithms and Parameters

3.1. Validation Step
As described in §2.2, the update cost is computed for a materialized view based upon a percentage $p$ of the number of tuples in the smallest materialized ancestor. Before validation, the solution is fully built and then checked against the history. If it is an original solution, the validation algorithm (Algorithm 1) is called. The most obvious approach would be to (1) compute all costs associated with the solution and (2) validate the entire solution. Although this produces the correct results, it does so in a naïve manner. That is, it does not take into consideration any of the properties of the solution itself.

A better approach would be to sort the solution by the size of the name in descending order and evaluate the solution one view at a time where both of the constraints (i.e., $S(M) \leq S$ and $U(M) \leq U$) can be checked. The key to this is sorting in descending order. While $S(M)$ does not require any specific order when dealing with ancestors and descendants, $U(M)$ is entirely dependent upon the order. For illustrative purposes, we present the following example: Let $M = \{ABC, C, A, B, AB, D\}$. If we were to set the costs for each one and check for feasibility starting with $ABC$, $U(M)$ would be incorrectly calculated for both $A$ and $B$. The reason being, $AB$ could very well be (and will most likely be) since it is almost assuredly smaller than $ABC$ the smallest materialization ancestor of both $A$ and $B$. This will decrease $U(M)$ if true which may in turn alter an infeasible
solution into a feasible one. That is, if \( |ABC| + 2(p|ABC|) > \mathcal{U} \) (according to a partial solution check), the program will exit; but this might not be true for \( |ABC| + 2(p|AB|) \). Therefore, if we sort \( \mathcal{M} \) to be \( \{ABC, AB, A, B, C, D\} \), this issue no longer exists. It is also worthy of note that intra-lattice level sorting is not necessary. That is, \( \{ABC, AB, D, C, A, B\} \) is just as valid since views in the same lattice level cannot be ancestors or descendants of one another. Therefore, to save computation time, the algorithms building the solutions only sorts based on length of the name (e.g., \( ABC = 3, A = 1 \), etc.).

Algorithm 1 below takes a solution either in the form of the example above (which will be presented as sorted) or in binary (for the genetic algorithm presented later), computes and evaluates the size and update constraints, exits if either constraint is violated with a response of failure or, if valid, computes the query response time for the lattice and returns the objective function value.

**Algorithm 1.** \( \text{ValidateSolution}(s) \)

1. **input:** a solution \( s \) to validate
2. \( Q(\mathcal{L}|\mathcal{M}) := 0 \) // objective function value
3. \( S(\mathcal{M}) := 0 \) // objective function value
4. \( U(\mathcal{M}) := 0 \) // update time constraint
5. \( valid \leftarrow false \)
6. **for** \( i = 1, \ldots, |s| \) **do**
7.     if \( S(\mathcal{M}) := S(\mathcal{M}) + r[|s|] > S \) OR
     \( U(\mathcal{M}) := U(\mathcal{M}) + g[|s|]U(|s|, \mathcal{M}) > U \) then
8.         return \( valid \) and exit
9. **end for**
10. valid := true
11. **for** \( j = 1, \ldots, |\mathcal{L}| \) **do**
12.     \( Q(\mathcal{L}|\mathcal{M}) := Q(\mathcal{L}|\mathcal{M}) + f[|s|]Q(|s|, \mathcal{M}) \)
13. **end for**
14. return \( valid \) and output \( Q(\mathcal{L}|\mathcal{M}) \)

3.2. Random Search

In random search, a random length solution of random vertices (queries) is generated and tested for feasibility. If feasible, the solution is outputted, if not, the process repeats itself until a feasible solution is found. The benefit of this algorithm is speed as it takes little time to find a feasible solution. However, since it is simply picking a random solution from an extremely large search space, the solution quality is generally quite poor. Algorithm 2 shows the pseudo code the random search algorithm implemented.

**Algorithm 2.** \( \text{RandomSearch}(\mathcal{L}) \)

1. **input:** cube lattice \( \mathcal{L} \)
2. \( Q(\mathcal{L}|\mathcal{M}) := 0 \) // objective function value
3. \( s := () \) // solution
4. \( r := \) random number between 1 and \( |\mathcal{L}| \)
5. **for** \( i = 1, \ldots, r \) **do**
6.     \( \{L^k\} \in s \) OR \( k = null \) do

3.3. First and Best Improvement

Since the entire search space is too large to search, improvement algorithms restrict their search to a smaller range; known as a neighborhood. Our neighborhood is defined as follows: Let \( \Phi \) be the solution space. For each solution \( i \in \Phi \), a set \( \mathcal{N}(i) \subseteq \Phi \) is defined as a neighborhood and \( j \in \mathcal{N}(i) \) is defined as a neighbor of \( i \).

Two types of improvement algorithms are applied to the dataset. The first is first improvement (FI) and the second is best improvement (BI). For both, three parameters were varied: the number of neighborhoods \( \mathcal{N} \), the number of neighbors in each neighborhood \( n \), and the amount of variation about the size of the initial solution \( \sigma \). For these experiments \( j \in \mathcal{N}(i) \) where \( |i| - \sigma \leq |j| \leq |i| + \sigma \); that is, the size of the neighbors in \( \mathcal{N}(i) \) are within \( \sigma \) of the size of the initial solution \( i \). For example, if \( |i| = 15 \) and \( \sigma = 2 \), then the size of any \( j \) can be between 13 and 17. This way, our neighborhood is more diverse with the hopes of avoiding neighborhoods with few, if any, feasible solutions. That is, the longer it takes to find a set of feasible solutions, the longer the algorithm runs. And since there are many combinations of two and three views that violate the space and/or update constraints alone; especially when dealing with higher order views (those in the upper half of the cube lattice), having a more diverse neighborhood aids in reducing the overall run time.

The parameter settings were as follow: \( \mathcal{N} = (1, 10, 25, 50, 75, 100) \), \( n = (25, 50, 75, 100) \), and \( \sigma = (0, 2, 5) \). All combinations were tested 10 times (72 combinations for each FI and BI) and an average for each was recorded.

3.3.1. First Improvement

The first improvement algorithm, as described by Algorithm 3, randomly generates a feasible solution \( i \) (for each of the \( \mathcal{N} \) neighborhoods), then it randomly generates up to \( n - 1 \) feasible solutions \( j \) until one better than \( i \) in terms of objective function value is found, then it moves on to the next neighborhood. The overall best solution found is then output. There are two termination criteria, one for the neighborhoods and one for the neighbors, consisting of 10 times either \( N \) or \( n \) (depending upon the loop) to make sure the algorithm terminates within a reasonable amount of time. The value 10 was arbitrarily chosen and can be changed to suit the problem. Line 14 of this algorithm
requires some additional explanation. The reason we restrict the size of a neighborhood to between 2 and $|L| - 2$, is to allow for enough neighbors to search. That is, using our case where $|L| = 50$, a neighborhood of size 0 or 50 yield only 1 solution (i.e., $r_{50}^0 = (50)$ = 1) and a neighborhood of size 1 or 49 have 50 solutions; whereas $2 \leq n \leq 48$ has a minimum of 1,225 combinations. If smaller combinations are allowed, then we may not find any feasible solutions and/or restrict our neighborhood searches to too few in order to realize any benefit from the local search.

Algorithm 3. FirstImprovement($L, \sigma, n, N$)
1: input: the cube lattice $L$, variation $\sigma$, number of neighborhoods $n$, and number of neighborhoods $N$
2: $Q(L|M) := 0$ // objective function value
3: $s_{best} := \emptyset$ // the best solution found so far
4: $s_{tried} := \emptyset$ // the set of solutions already tried
5: $N_{term} := 0$ // termination iterator for $N$
6: for $i = 1, \ldots, N$ do
7: for $j = 1, \ldots, n$ do
8: if $n_{term} > 10 \cdot n$ exit end if
9: found := false // F.I. found
10: $r_{beg} :=$ random number between 2 and $|L| - 2$
11: if $r_{min} := r_{beg} - \sigma < 0$ then $r_{min} := 0$ end if
12: if $r_{max} := r_{beg} + \sigma > |L|$ then $r_{max} := |L|$ end if
13: for $i = 1, \ldots, r$ do
14: if $n_{term} > 10 \cdot n$ || found exit end if
15: $s := \emptyset$ // solution
16: $r :=$ random number between $r_{min}$ and $r_{max}$
17: end while
18: $s := s \cup \{k\}
19: end for
20: sort $s$ from largest to smallest by element size
21: if $s \notin s_{tried}$ then
22: if $Q(L|M) \leftarrow$ ValidateSolution($s$) is valid then
23: if $Q(L|M) \neq 0$ then // if not the first solution
24: if $Q(L|M) < Q(L|M)_{best}$ then // if F.I.
25: $Q(L|M)_{best} := Q(L|M)$
26: $s_{best} := s$
27: found := true
28: end if
29: else // if the first solution
30: $Q(L|M)_{best} := Q(L|M)$
31: $s_{best} := s$
32: end if
33: $s_{tried} := s_{tried} \cup s$
34: end if
35: else
36: $j := j - 1$ // add one back to $j$
37: end if
38: end if
39: end if
40: end for
41: end for
42: $n_{term} := n_{term} + 1$
43: end for
44: $N_{term} := N_{term} + 1$
45: end for
46: output: $Q(L|M)_{best}$ and $s_{best}$ and exit

3.3.2. Best Improvement
The best improvement algorithm, as depicted in Algorithm 4, randomly generates a feasible solution $i$ (for each of the $N$ neighborhoods), then it randomly generates $n - 1$ feasible solutions $j$ and stores the best one, then it moves on to the next neighborhood. The best solution found is then output. Again, there are two exit criteria just as with F.I.

Algorithm 4. BestImprovement($L, \sigma, n, N$)
1: input: the cube lattice $L$, variation $\sigma$, number of neighbors $n$, and number of neighborhoods $N$
2: $Q(L|M) := 0$ // objective function value
3: $Q(L|M)_{best} := 999999999999$ // best O.F. value
4: $s_{best} := \emptyset$ // the best solution found so far
5: $s_{tried} := \emptyset$ // the set of solutions already tried
6: $N_{term} := 0$ // termination iterator for $N$
7: for $i = 1, \ldots, N$ do
8: if $n_{term} > 10 \cdot n$ exit end if
9: $r_{beg} :=$ random number between 2 and $|L| - 2$
10: if $r_{min} := r_{beg} - \sigma < 0$ then $r_{min} := 0$ end if
11: if $r_{max} := r_{beg} + \sigma > |L|$ then $r_{max} := |L|$ end if
12: for $j = 1, \ldots, n$ do
13: if $n_{term} > 10 \cdot n$ \&\& $r_{min} := 0$ end if
14: if $r_{min} := r_{beg} - \sigma < 0$ then $r_{min} := 0$ end if
15: $r :=$ random number between $r_{min}$ and $r_{max}$
16: while $\{k\} \in s$ OR $k = null$ do
17: $k :=$ random number between 1 and $|L|$
18: end while
19: $s := s \cup \{k\}$
20: $n_{term} := n_{term} + 1$
21: $N_{term} := N_{term} + 1$
22: end if
23: sort $s$ from largest to smallest
24: if $s \notin s_{tried}$ then
25: if $Q(L|M) \leftarrow$ ValidateSolution($s$) is valid then
26: if $Q(L|M) < Q(L|M)_{best}$ then // if F.I.
27: $Q(L|M)_{best} := Q(L|M)$
28: $s_{best} := s$
29: end if
30: $s_{tried} := s_{tried} \cup s$
31: end if
32: else
33: $j := j - 1$ // add one back to $j$
34: end if
35: $n_{term} := n_{term} + 1$
36: end for
37: $N_{term} := N_{term} + 1$
38: end for
39: output: $Q(L|M)_{best}$ and $s_{best}$
3.4. Genetic Algorithms

Genetic algorithms (GA) are a variant of evolutionary algorithms. As this algorithm simulates biological processes, the terminology follows suit. GA's begin with an initial population of solutions that are generally randomly generated (known as a generation). Each solution is called a chromosome. The actual chromosome structure is called a genotype which consists of alleles, the individual components of the solution, which have possible values called alleles. The meaning of each solution is known as its phenotype. Simply put, consider a solution to be a binary string of 5 elements. The chromosome is a binary string, the genotype is the binary string of length 5, there are 5 genes with two possible allele values (0 and 1), and the phenotype could then be the binary number's base 10 value (Burke & Kendall, 2005; Goldberg, 1989).

Offspring (future generations) are generated by combining two parents (two solutions in the population). To create children, a crossover point(s) is selected. All genes from the first parent up to and including the crossover point and all genes from the second parent from the gene after the crossover point to the end are combined to make the first child; the second child is simply the remaining genes. For example, if the parents are $P_1 = 10010$ and $P_2 = 01111$ with a crossover point of 3, then offspring $O_1 = 10011$ and $O_2 = 01110$. Next, genetic mutation is applied to the children. This is done probabilistically by flipping a bit with a probability of, e.g., 5% (Burke & Kendall, 2005). The resulting generation of offspring can be merged with the parents or taken alone as the new population for subsequent computations.

The genetic algorithm requires the tuning of four parameters. The first is the population size $\rho$. According to (Burke & Kendall, 2005), a population size of 50 is a good starting point. The second is the crossover rate $\chi$. (Burke & Kendall, 2005) suggest a value of 0.6. In the example above, the crossover point 3 is the result of the crossover rate times the size of the genotype (i.e., $0.6 \times 5$). The third is the mutation factor $\eta$. (Burke & Kendall, 2005) suggest a starting value of 0.05. The fourth is the population strategy. This setting is used to either diversify or intensify the population. That is, by intensifying the population, we are taking an elitist approach in which each resulting generation will be the best solutions from the existing population and offspring. The diversification strategy incorporates a step size in the selection of the population, meaning we take every $k^{th}$ solution in the merged set.

For this paper, two population creation methods were used: (1) randomly generated and (2) the results from a local search. The genotype is a binary string consisting of $|\mathcal{L}|$ elements, where a 0 means the view is not selected for materialization and a 1 means it is selected for materialization. The following parameter settings were used: $\rho = (50, 100)$, $\eta = (0.005, 0.05), \chi = (0.25, 0.6, 0.75)$, and population strategy = (intense, diverse) – for diverse, we use every other $5^{th}$ (unless the results size is less than 5 times the size of the population in which we use the largest possible scalar). Each test was performed 10 times.

3.4.1. Mutation and Crossover

Before delving into the GA algorithms, we first cover the crossover algorithm (Algorithm 5). As stated above, it takes two solutions from the population, implements the crossover, and then mutates the genes. Once completed, it checks to see if it is original, if it is, then it is tested for feasibility and stores it if it is. The process is repeated for every possible combination.

Algorithm 5. Crossover($\mathcal{L}, \mathcal{J}, \chi, \eta$)

1: input: the cube lattice $\mathcal{L}$, initial population $\mathcal{J}$, crossover $\chi$, and mutation factor $\eta$
2: $Q(\mathcal{L}|\mathcal{M}) := 0$ // objective function value
3: $S_{\text{tried}} := \emptyset$ // the set of solutions already tried
4: for $i = 1, \ldots, |\mathcal{J}|$ do
5: for $j = i + 1, \ldots, |\mathcal{J}|$ do
6: $p_1 := \mathcal{J}^i$ // parent 1
7: $p_2 := \mathcal{J}^j$ // parent 2
8: $o_1 := \{p_1^1, \ldots, p_1^{|\mathcal{L}|-|\mathcal{J}|+1}\} \cup \{p_2^{|\mathcal{L}|-|\mathcal{J}|+1}, \ldots, p_2^{|\mathcal{L}|}\}$ /* offspring 1 */
9: for $n = 1, \ldots, |o_1|$ do
10: if $r := \text{random number} \leq \eta$, flip $o_1^n$ bit end if
11: end for
12: $o_2 := \{p_2^1, \ldots, p_2^{|\mathcal{L}|-|\mathcal{J}|+1}\} \cup \{p_1^{|\mathcal{L}|-|\mathcal{J}|+1}, \ldots, p_1^{|\mathcal{L}|}\}$ /* offspring 2 */
13: for $n = 1, \ldots, |o_2|$ do
14: if $r := \text{random number} \leq \eta$, flip $o_2^n$ bit end if
15: end for
16: if $o_k \notin S_{\text{tried}}, k = 1, 2$ then
17: if $Q(\mathcal{L}|\mathcal{M}) \leftarrow \text{ValidateSolution}(o_k)$ is valid then
18: $S_{\text{tried}} := S_{\text{tried}} \cup o_k$
19: end if
20: end if
21: end for
22: end for
23: sort $\mathcal{J} := \mathcal{J} \cup S_{\text{tried}}$ from smallest $Q(\mathcal{L}|\mathcal{M})$ to largest
24: output: $\mathcal{J}^1$ solution $s$ and $Q(\mathcal{L}|\mathcal{M})$

3.4.2. Random Search Genetic Algorithm

The random population generation is quite simple and related to the random search shown in Algorithm 2. The size of the population is first calculated based on the strategy $\delta$. If the strategy is to diversify, then we create $5\rho$ random feasible solutions (5 can be altered to suit the needs of the problem). Then, the resulting solutions are sorted from smallest objective function value to largest. From that list, the initial population is built and Crossover is called.

Algorithm 6. RandomSearchGA($\mathcal{L}, \chi, \eta, \rho, \delta$)

1: input: the cube lattice $\mathcal{L}$, crossover $\chi$, mutation factor $\eta$, population size $\rho$, and strategy $\delta$
2: $Q(L|M) := 0$ // objective function value
3: $pop := \emptyset$ // the population
4: $s_{tried} := \emptyset$ // the set of solutions already tried
5: if $\delta = \text{diverse}$ then $\lambda := 5$ else $\lambda := 1$ end if
6: for $i = 1, \ldots, \lambda \rho$ do
7: $s := \langle \rangle$ // solution
8: $r := \text{random number between 1 and } |L|$ // for $j = 1, \ldots, r$ do
10: while $\{L^k\} \in s$ OR $k = \text{null}$ do
11: $k := \text{random number between 1 and } |L|$ end while
13: $s := s \cup \{L^k\}$ end if
14: else
15: if $s \notin s_{tried}$ then $s_{tried} := s_{tried} \cup s$ end if
19: else
20: $i := i - 1$ // add one back to $i$
21: end if
22: end for
23: sort $s_{tried}$ by $Q(L|M)$ value ascending
24: for $l = 1, \ldots, \rho$ do
25: $pop := pop \cup s_{\lambda l}^\text{tried}$ end for
26: end for
27: output: $Q(L|M)$ and $s \leftarrow \text{Crossover}(L, pop, \chi, \eta)$

3.4.3. Local Search Genetic Algorithm

The local search generated population follows the same algorithmic outline as BI (Algorithm 4). The initial size of the population depends upon the number of neighborhoods and neighbors. The resulting set of solutions is sorted in the same manner as Algorithm 6. In order to build the population, the step size $\lambda$ will have to be calculated. That is, with the random version, we can make sure $5\rho$ solutions are created, however with local search, we may come up short. Therefore, $\lambda$ is the minimum between 5 and the number of solutions divided by the desired population size. This way, we are certain to have a fully populated population (assuming that at least $\rho$ feasible solutions were generated). From that list, Crossover is initiated.

The size of the initial neighborhoods, number of neighbors, and variation about the initial random, feasible solution, were set using the results from first and best improvements, and will be discussed further in §4.4.2.

Algorithm 7. LocalSearchGA($L, \sigma, n, N, \chi, \eta, \rho, \delta$)
1: input: the cube lattice $L$, variation $\sigma$, number of neighbors $n$, number of neighborhoods $N$, crossover $\chi$, mutation factor $\eta$, population size $\rho$, and strategy $\delta$
2: $Q(L|M) := 0$ // objective function value
3: $pop := \emptyset$ // the population
4: $s_{tried} := \emptyset$ // the set of solutions already tried
5: $N_{term} := 0$ // termination iterator for $N$
6: $\lambda := 1$ // solution increment size
7: for $i = 1, \ldots, N$ do
8: if $N_{term} > 10 \times N$ exit end if
9: $n_{term} := 0$ // termination iterator for $n$
10: $r_{beg} := \text{random number between } 2$ and $|L| - 2$
11: if $r_{min} := r_{beg} - \sigma < 0$ then $r_{min} := 0$ end if
12: $r_{max} := r_{beg} + \sigma > |L|$ then $r_{max} := \|L\|$ end if
13: for $j = 1, \ldots, n$ do
14: if $n_{term} > 10 \times n$ exit end if
15: $s := \langle 0 \rangle$ // solution initialized to $|L|$ 0’s
16: $r := \text{random number between } r_{min}$ and $r_{max}$
17: for $i = 1, \ldots, r$ do
18: while $s^k = 1$ OR $k = \text{null}$ do
19: $k := \text{random number between 1 and } |L|$ end while
20: $s^k := 1$
22: end for
23: if $s \notin s_{tried}$ then $s_{tried} := s_{tried} \cup s$ end if
24: if $Q(L|M) \leftarrow \text{ValidateSolution}(s)$ is valid then
25: $s_{tried} := s_{tried} \cup s$ end if
27: else
28: $j := j - 1$ // add one back to $j$
29: end if
30: $n_{term} := n_{term} + 1$
31: end for
32: $N_{term} := N_{term} + 1$
33: end for
34: sort $s_{tried}$ by $Q(L|M)$ value ascending
35: if $\delta = \text{diverse}$ then $\lambda := \min\{5, \lfloor s_{\lambda l}^\text{tried} \rfloor\}$ end if
36: for $l = 1, \ldots, \rho$ do
37: $pop := pop \cup s_{\lambda l}^\text{tried}$ end for
39: output: $Q(L|M)$ and $s \leftarrow \text{Crossover}(L, pop, \chi, \eta)$

4. Computational Results

In the following section, we discuss the computation results of the algorithms presented in the previous section.

4.1. Setup

To establish a lower bound on the objective function value, we computed $Q(L|M)$ given unlimited resources; 1,604,896.28. Likewise, we computed the upper bound given no resources (i.e., computing straight from the base tables); 19,629,434. Although these have no basis on the optimal solution, it can be used as a basis for comparisons.

For these experiments, $\delta$ was set to 30% of the total size of the data warehouse, $\delta = 22,018,130$, and $\mathcal{U}$ was set to 15% of the total size of the data warehouse, $\mathcal{U} = 11,009,065$. The percentages were determined through preliminary experimentation. The update cost percentage $p$ was set to 10% as in (Kalnis et al., 2002).
4.2. Random Search
Random search was performed 20 times; see Table 1. The best solution found was in iteration 12 where $Q(\mathcal{C} | \mathcal{M}) = 2,271,179.05$ and a size of 19 views in the set. However, the average over the 20 runs is more than 4 times larger with a standard deviation of 4,516,539.59. Furthermore, several of the solutions are nearly as poor as not materializing any views at all. Therefore, random search is not a good tool to use when solving the VSP.

Table 1: Random Search Results. Best solution in bold.

| $Q(\mathcal{C} | \mathcal{M})$ | $S(\mathcal{M})$ | $U(\mathcal{M})$ | Size | Time |
|-----------------|--------------|-------------|------|------|
| 18,914,866.25   | 386,100.00   | 105,294.51  | 1    | 16   |
| 12,141,381.89   | 6,047,534.00 | 2,087,794.44| 4    | 0    |
| 3,129,031.27    | 14,369,318.00| 5,660,104.46| 14   | 16   |
| 9,125,159.51    | 9,600,601.00 | 2,658,813.04| 11   | 16   |
| 8,770,809.52    | 10,532,218.00| 8,166,521.07| 13   | 0    |
| 8,784,468.85    | 6,021,371.00 | 3,110,492.63| 6    | 15   |
| 9,415,276.66    | 6,462,018.00 | 2,081,463.21| 4    | 0    |
| 14,244,772.21   | 3,081,230.00 | 1,133,814.78| 5    | 0    |
| 14,198,425.88   | 2,111,654.00 | 549,304.15  | 1    | 0    |
| 2,889,611.71    | 16,934,267.00| 7,332,939.96| 20   | 0    |
| 7,241,295.45    | 7,116,087.00 | 5,089,829.94| 4    | 16   |
| 12,271,179.05   | 19,492,718.00| 6,792,670.88| 19   | 16   |
| 9,835,882.61    | 5,151,494.00 | 2,009,635.55| 7    | 0    |
| 11,010,165.38   | 8,074,453.00 | 4,434,807.36| 10   | 16   |
| 11,488,695.73   | 3,912,494.00 | 2,658,895.26| 7    | 16   |
| 5,067,974.22    | 15,288,346.00| 6,330,704.27| 10   | 0    |
| 4,388,043.34    | 15,415,458.00| 5,649,625.33| 17   | 0    |
| 11,971,873.55   | 1,824,541.00 | 877,347.65  | 5    | 0    |
| 14,427,563.25   | 17,052,242.00| 6,677,615.08| 10   | 0    |
| 20,158,082.21   | 5,909,372.00 | 5,174,491.22| 2    | 0    |
| AVG             | 9,265,177.93 | 8,739,175.80| 3,926,608.24| 8.50 | 5.55 |

4.3. First and Best Improvement
As described in §3.3, there are 72 combinations of parameter settings performed using FI and BI. Appendices A – D have the combined results for all FI and BI experiments. The results in Appendix B show there is a significant drop off in objective function improvement when $N = 10$ and $N = 25$; regardless of the type of improvement used and the variation about the initial solution. Also, Appendix A shows that the running time increases nearly linearly with $N$, with a definite advantage to a variation of +/-5. This can be attributed to a wider variety of solutions in each neighborhood, increasing the overall number of feasible solutions and thus, decreasing the running time. FI was expected to be faster than BI because it does not search the entire neighborhood. However, the difference is not as distinct in smaller sizes of $n$ more than likely because FI searches most, if not all, of the neighborhood in order to find an improved solution. Appendix C shows that both FI and BI approach the size constraint rapidly by around $N = 10$. What is interesting to see is the gap between parameter settings as the variation about the initial solution increases. At $\sigma = 0$, most of the plots are on top of or near the others. When $\sigma = +/-2$ and +/-5, the gap increases with larger sizes of $n$ consuming more space. This can be attributed to a greater number of increasingly diverse solutions being searched. Appendix D is quite different from the previous graphs as no discernable information can be gained about one set of parameters over another. It does, however, shed light on a potential issue with the view-maintenance formulation of the VSP which will be covered in our concluding remarks in §5.

Table 2 shows the top ten worst solutions and Table 3, the top 10 best solutions found. As you can see from Table 2, all of the solutions have $N = 1$. This is not that surprising as the algorithm simply finds the best (or first improvement) solution to a single, random neighborhood. The top 10 best (Table 3), not surprisingly, come from larger neighborhoods with a larger number of neighbors. Also, as expected, the better quality solutions come from BI while the worst come from FI. This is simply attributed to the depth of search performed in each algorithm. One noticeable issue is how poor the worst solutions are. They are not as bad as a purely random search, but the $Q(\mathcal{C} | \mathcal{M})$ distance between the average best and worst solutions is extremely large (581%); showing how sensitive improvement algorithms are to the input parameters. Furthermore, the best solution found is only 2.19% larger than the lower bound computed in §4.1 and consists of a mere 23 views. Although this is purely an artifact of the queries and parameters used for this problem instance, we know that our solution quality cannot improve much further.

Table 2: Worst Individual 10 Overall

| $Q(\mathcal{C} | \mathcal{M})$ | $N$ hoods | $N$ bors Variation | Time (ms) | Method |
|-----------------|------------|-------------------|-----------|--------|
| 13,874,606.47   | 1          | 50                | 2         | 0      |
| 13,726,092.10   | 1          | 75                | 2         | 16     |
| 11,988,744.52   | 1          | 100               | 0         | 703    |
| 11,476,135.02   | 1          | 25                | 2         | 15     |
| 10,827,776.08   | 1          | 25                | 0         | 0      |
| 10,526,204.58   | 1          | 75                | 5         | 578    |
| 7,491,834.66    | 1          | 50                | 2         | 0      |
| 8,370,215.43    | 1          | 75                | 2         | 0      |
| 9,176,553.91    | 1          | 50                | 0         | 0      |
| 9,968,239.59    | 1          | 100               | 0         | 0      |

Table 3: Best Individual 10 Overall

| $Q(\mathcal{C} | \mathcal{M})$ | $N$ hoods | $N$ bors Variation | Time (ms) | Method |
|-----------------|------------|-------------------|-----------|--------|
| 1.640,045.26    | 5          | 50                | 5         | 5.094  |
| 1.644,950.60    | 75         | 100               | 5         | 12,093 |
| 1.647,351.81    | 100        | 100               | 2         | 20,469 |
| 1.656,688.45    | 25         | 100               | 0         | 4,547  |
| 1.658,344.33    | 100        | 100               | 2         | 15,594 |
| 1.674,414.75    | 75         | 25                | 2         | 1,953  |
| 1.678,251.95    | 75         | 75                | 2         | 8,516  |
| 1.680,522.03    | 50         | 75                | 2         | 4,390  |
| 1.683,876.35    | 25         | 50                | 0         | 1,890  |
| 1.684,565.87    | 75         | 25                | 2         | 1,781  |
4.4. Genetic Algorithms

4.4.1. Random Search Genetic Algorithm

As depicted in §3.4, there are a total of 24 possible parameter combinations, each of which were run 10 times with their averages computed. Table 4 shows the worst 10 averages and Table 5, the best 10. What is immediately evident is that 90% of the worst solutions have a population size of 50, 100% have a mutation factor of 0.005, and 90% have a crossover rate greater than 0.25 (strategy is split DIV/INT @ 60/40). Comparing that with Table 5 where 90% of the population is of size 100, there is a much larger crossover rate percentage at 0.25, and 70% intensification strategy rate (mutation rate is split 0.005/0.05 @ 40/60), one can quite confidently conclude that a larger population and mutation factor, smaller crossover rate, and an intensification strategy produce higher quality results for this particular problem instance. It is also interesting to note the $Q(L|M)$ gap between the best and worst 10. The difference between the average best and worst solutions is 42%. This is still large, but much smaller than seen using either improvement algorithms.

<table>
<thead>
<tr>
<th>Table 4: Worst Individual 10 Overall</th>
</tr>
</thead>
<tbody>
<tr>
<td>Worst $Q(L</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 5: Best Individual 10 Overall</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best $Q(L</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>10</td>
</tr>
</tbody>
</table>

4.4.2. Local Search Genetic Algorithm

Before we begin, we will discuss the local search parameter settings. Using the results in Appendix B (and as mentioned in §4.3), one can see an almost asymptotic like leveling of objective function values around 25 neighborhoods with 25+ neighbors. Also, the amount of variation has little to do with the success of the algorithm in terms of objective function value. However, when comparing the times in Appendix A, a variation of +/- 5 is noticeably faster. Therefore, it was decided to perform two tests: (1) 25 neighborhoods, 25 neighbors, and +/- 5 variation (denoted Local Search – Genetic Algorithm 25) and (2) 50 neighborhoods, 50 neighbors, and +/- 5 variation (denoted Local Search – Genetic Algorithm 50).

4.4.2.1. Local Search – Genetic Algorithm 25

Along with the parameters stated above, the 24 possible parameter combinations were tested just as with the random GA version. Table 6 shows the worst 10 averages and Table 7, the best 10. What is immediately evident is that 90% of the worst solutions have a mutation factor of 0.005, a strategy of diversification, and crossover rate greater than 0.25 (population is split 50/100 @ 60/40). Comparing this with Table 7 where 100% of the mutation factors are 0.05, 90% of the strategies are intensification, and 60% of the crossover rates are 0.25 (population is split 50/100 @ 40/60), one can deduce that a larger mutation factor, smaller crossover rate, and an intensification strategy, produce higher quality solutions for this particular problem instance. It is also interesting to note, as with random GA, the $Q(L|M)$ gap between the best and worst 10. The difference between the average best and worst solutions is a mere 16%; nearly one-third the size of random GA. Furthermore, the best solution found is only 1.85% larger than the lower bound computed in §4.1 and consists of only 24 views.

<table>
<thead>
<tr>
<th>Table 6: Worst 10 Overall Local Search – Genetic Algorithm 25</th>
</tr>
</thead>
<tbody>
<tr>
<td>Worst $Q(L</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 7: Best 10 Overall Local Search – Genetic Algorithm 25</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best $Q(L</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>10</td>
</tr>
</tbody>
</table>

4.4.2.2. Local Search – Genetic Algorithm 50

This section follows the same parameter testing as in the previous section. Table 8 shows the worst 10 averages and Table 9, the best 10. Similar to the previous tables of worst solutions, 80% of the population are of size 50 and 90% have a mutation factor of 0.005 and crossover rate greater than 0.25 (strategy is split DIV/INT @ 60/40). If we compare these observations to Table 9 where 100% of the strategies are intensification, 80% of the crossover rates are 0.25, and 70% of the populations are 100 (mutation factor is
split 0.005/0.05 @ 40/60), it is easy to see that a larger population, smaller crossover rate, and an intensification strategy creates solutions of better quality for this particular problem instance. Like the others, it is interesting to note that the $Q(L|M)$ gap between the average best and worst 10 is the smallest seen at 10%; 60% tighter than Local Search – Genetic Algorithm 25. Furthermore, the best solution found is only 1.21% larger than the lower bound computed in §4.1 and consists of a mere 28 views.

<table>
<thead>
<tr>
<th>Table 8: Worst 10 Overall Local Search – Genetic Algorithm 50</th>
</tr>
</thead>
<tbody>
<tr>
<td>Worst</td>
</tr>
<tr>
<td>-------</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>10</td>
</tr>
</tbody>
</table>

4.5. Algorithmic Comparisons

First, we will show a comparison of all algorithms attempted. As you can see from Table 10, the use of genetic algorithms combined with local search produces not only the best solutions and average solutions, but does it in 13%-25% of the time compared to local search alone. The randomly populated genetic algorithm, as expected, performs poorly compared to the other non-random algorithms, but significantly outperforms random search alone; however, as shown in §4.4.1, this algorithm is, although not producing the best $Q(L|M)$ or best average $Q(L|M)$, it does construct a tight cluster of solutions. Intuitively, random search with genetic algorithms is a direct improvement over random search and therefore should be no less than its predecessor. This is also the case for first improvement/best improvement and the genetic algorithm with local search.

Second, Table 11 shows the percent of average objective function values that are better in the algorithms listed across the top than the best average for the algorithms listed on the side. As you can see, all algorithms, regardless of the parameter setting, are better than random search. Also, 51.39% of first improvement solutions and 66.67% of best improvement solutions outperform the best average solution for the random search genetic algorithm. Furthermore, the best algorithm, as expected, was the local search genetic algorithm with parameters settings of $\rho = 100, \eta = 0.05, \chi = 0.25$, strategy of intense, $N = 50, n = 50$, and $\sigma = +/- 5$.

Lastly, Table 12 shows the best average solution overall compared with the population size of 50 variant. The reason for this comparison is to see if the improvement in objective function value is worth the added time. The first two columns of Table 12 are the average $Q(L|M)$ and time values for the 10 iterations used in general testing. The objective function value is decreased by 1.038% if we use a population size of 50, but the time is reduced by 55.391%. Since our sample size is only 10, we ran these two experiments again at 1,000 iterations to reduce sampling error; the results of which can be seen in the third and fourth columns. As you can see, the $Q(L|M)$ benefit gap was reduced to a mere 0.575% while the time difference remained fairly steady and substantial at 50.668%. It is therefore, our conclusion that using genetic algorithms with a local search initialized population with the parameters listed below, is the best for this problem instance when compared to those algorithms tested.

Parameter settings: $\rho = 50, \eta = 0.05, \chi = 0.25$, strategy = intense, $N = 50, n = 50$, and $\sigma = +/- 5$.

<table>
<thead>
<tr>
<th>Table 9: Best 10 Overall Local Search – Genetic Algorithm 50</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 10: For each method attempted, the best solution, average objective function value, average solution size, and average time in milliseconds are shown</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best $Q(L</td>
</tr>
<tr>
<td>----------------</td>
</tr>
<tr>
<td>Random Search</td>
</tr>
<tr>
<td>First Improvement</td>
</tr>
<tr>
<td>Best Improvement</td>
</tr>
<tr>
<td>Random Search – Genetic Algorithm</td>
</tr>
<tr>
<td>Local Search – Genetic Algorithm 25</td>
</tr>
<tr>
<td>Local Search – Genetic Algorithm 50</td>
</tr>
</tbody>
</table>

Improvement parameters are as such: (neighborhoods, neighbors, variation). Genetic algorithm parameters are as such: (population, mutation, crossover, strategy). 1 Average over 20 samples, 2 (75, 100, 5), 3 (100, 100, 2), 4 (75, 50, 5), 5 (100, 0.005, 0.75, diverse) – this value is an anomaly as the section average was only 1,961,200.90, 6 (100, 0.05, 0.6, intense), 7 (100, 0.05, 0.25, intense), 8 (100, 0.05, 0.75, intense), 9 (100, 0.05, 0.25, intense)
Table 11: The percent of average objective function values that are better in the algorithms listed across the top than the best average for the algorithms listed on the side

<table>
<thead>
<tr>
<th>First Improvement</th>
<th>Best Improvement</th>
<th>Random Search – Genetic Algorithm</th>
<th>Local Search – Genetic Algorithm 25</th>
<th>Local Search – Genetic Algorithm 50</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random Search</td>
<td>100.00%</td>
<td>100.00%</td>
<td>100.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td>First Improvement</td>
<td>-</td>
<td>8.33%</td>
<td>0.00%</td>
<td>33.33%</td>
</tr>
<tr>
<td>Best Improvement</td>
<td>0.00%</td>
<td>-</td>
<td>0.00%</td>
<td>4.17%</td>
</tr>
<tr>
<td>Random Search – Genetic Algorithm</td>
<td>51.39%</td>
<td>66.67%</td>
<td>-</td>
<td>79.17%</td>
</tr>
<tr>
<td>Local Search – Genetic Algorithm 25</td>
<td>0.00%</td>
<td>0.00%</td>
<td>-</td>
<td>25.00%</td>
</tr>
</tbody>
</table>

Table 12: Average Q(L) and time values for the local search genetic algorithm using neighborhoods of size 50, neighbors of size 50, and variation of +/- 5 for 10 iterations and 1,000 iterations

| Percent improvement | Average Q(L|M) (10 iterations) | Average Time (ms) (10 iterations) | Average Q(L|M) (1,000 iterations) | Average Time (ms) (1,000 iterations) |
|---------------------|-------------------------------|-----------------------------------|-----------------------------------|-------------------------------------|
| (100, 0.05, 0.25, intense) | 1,681,566.84 | 3653.10 | 1,693,883.81 | 3,409.30 |
| (50, 0.05, 0.25, intense) | 1,699,015.48 | 1629.60 | 1,703,622.79 | 1,681.87 |
| Percent improvement | -1.038% | 55.391% | -0.575% | 50.668% |

4.6. Local Search – Genetic Algorithm Extension

In this section, we perform additional experiments using, as its base, the parameters defined at the end of §4.5. The goal is to determine the extent to which increasing the size of N or π affects the solution quality compared to the time required for the algorithm to run, and to one another.

First, we ran 5 experiments using the following parameters: ρ = 50, η = 0.05, χ = 0.25, strategy = intense, N = 1,000, n = 100, and σ = +/- 5; the results can be seen in Table 13 below. Comparing the average solution to the average of the 10 best in Table 9 (i.e., Q(L|M) = 1,641,717.43 and Time (ms) = 3,107.9), we get a decrease in objective function value of 0.6% at a time increase of 26.080%.

Second, we ran 5 experiments with the same parameter setting except N = 100 and n = 1,000; see Table 14. Comparing the average solution to the average of the 10 best in Table 9, we get a decrease in objective function value of 0.28% at a time increase of 21.807%.

Comparing these two extensions, the objective function value in the first is 0.32% better than the second, but takes 19.51% more time. This seems to indicate that increasing N before N may produce higher quality results for the time taken. However, both extensions fail to improve on the results found in Table 9. Therefore, it is our judgment that the parameters listed at the end of §4.5 are more than adequate to solve this problem given the tradeoff between solution quality and time.

5. Conclusions

A data warehouse is designed to integrate multiple, heterogeneous information sources to facilitate decision support and data analysis. One major component to such a design is the selection of views to materialize to decrease query response times. In this paper, we applied three heuristic techniques (random search, first and best improvement, and random search and local search based genetic algorithms) to the VSP.

Through extensive parameter tuning and experimentation we have shown local search initialized genetic algorithms to be the favorable choice for our example problem instance, and provided the preferred parameter settings in §4.5. As we only tested one instance, it is not clear if the algorithm is uniformly superior to the others; further research needs to be conducted before any generalizations can be claimed. Along with (Horng et al., 1999; Zhang & Yang, 1999), our experiments prove GA to be a powerful heuristic on the VSP and champion its further usage.

As stated in §4.3, the view-maintenance formulation of the VSP presents several issues. In §1.1, the warping of the traditional VSP formulation into the view-maintenance formulation was discussed. The advantage of the view-maintenance model over the traditional model is limiting the update time. However, by removing this cost from the objective function, we are no longer able to minimize the combined query response time and update cost. This may explain the wild results shown in Appendix D where the effects of updating have become an afterthought. That is, updating a set of materialized views is generally measured in hours or days whereas query response times in minutes or seconds. While not every query is asked at the same time, materialized views are generally updated in a single process. Current thinking hinges on the fact that there is an allotted update window and whether or not we use all or a portion of it is of little importance. Conversely, a slight decrease in update time (measured in hours) associated with a slight increase in query response time (measured in seconds) will have an overall positive impact on system performance. That is, reducing the update time allows for additional query processing time. Of course there is a tradeoff between these times as update frequencies play an important role, but a
hybrid model between the traditional and view-maintenance VSP formulations may prove invaluable.

In the future, we plan to explore the heuristic search results on various datasets (some with known optimal solutions) as well as the implementation of other heuristics. Furthermore, we also intend to tackle the hybridization of the VSP model formulation.

References
Appendices
Appendix A: Running times in milliseconds for each variation (x-axis: the size of the neighborhoods)

![Graph showing run time in milliseconds for each variation](image)

- **Run Time (ms) @ 0 Variation**
- **Run Time (ms) @ +/- 2 Variation**
- **Run Time (ms) @ +/- 5 Variation**
Appendix B: $Q(L|M)$ values for each variation (x-axis: the size of the neighborhoods)
Appendix C: $S(\mathcal{M})$ values for each variation (x-axis: the size of the neighborhoods)
Appendix D: $\mathcal{U}(M)$ values for each variation (x-axis: the size of the neighborhoods)

### U(M) Value @ 0 Variation

- FI-25 neighbors
- FI-50 neighbors
- FI-75 neighbors
- FI-100 neighbors
- BI-25 neighbors
- BI-50 neighbors
- BI-75 neighbors
- BI-100 neighbors

### U(M) Value @ +/- 2 Variation

- FI-25 neighbors
- FI-50 neighbors
- FI-75 neighbors
- FI-100 neighbors
- BI-25 neighbors
- BI-50 neighbors
- BI-75 neighbors
- BI-100 neighbors

### U(M) Value @ +/- 5 Variation

- FI-25 neighbors
- FI-50 neighbors
- FI-75 neighbors
- FI-100 neighbors
- BI-25 neighbors
- BI-50 neighbors
- BI-75 neighbors
- BI-100 neighbors