# Computing Optimal Snowplow Route Plans Using Genetic Algorithms

T.M. Rao, Sandeep Mitra, James Zollweg
Departments of Computer Science and Earth Sciences
The College at Brockport, State University of New York
Brockport, NY, USA
{trao, smitra, jzollweg}@brockport.edu

Ferat Sahin
Department of Electrical Engineering
Rochester Institute of Technology
Rochester, NY, USA
feseee@rit.edu

***Abstract*—The road network of a small town is represented by a directed graph. Road junctions are the vertices of this graph and each road segment (which has a length and a priority value) is represented by a directed edge. Priority values are numbers 1, 2, etc. with the assumption that 1 is the highest priority. We seek to compute an optimal route map that begins at a particular vertex (the depot) and covers all the edges at least once and returns to the start vertex. The parameters that we wish to minimize are: the total distance covered (thereby minimizing the deadhead miles), the number of u-turns and priority misplacements. In this paper, we propose a Genetic Algorithms-based solution to compute near-optimal route maps in such a graph. Specifically, we have developed a Java software application that generates route maps that minimize a linear combination of the three parameters. We have experimented with reasonably large graphs and obtained good solutions. These solutions are especially useful in snowplow routing for small towns, as plowing costs consume significant portions of the total municipal budgets of these communities. Most of the route planning is currently done manually and routes have evolved over time by experience. In these times of severe budget stress, route planning using our approach can help in performing this essential service in an efficient manner.**

***Keywords: Snowplow Routing, Genetic Algorithms***

## I. INTRODUCTION

The problem of computing an optimal route map in a graph occurs in many applications, ranging from delivering packages, and garbage collection to snowplow routing. Each segment of the route has a length that represents the cost of traversing that segment. Obviously, one wants to minimize the total length. A snowplow has to cover all the road segments. Further, the roads may be assigned priorities 1, 2, etc. implying that priority 1 roads are more important and need to be plowed before priority 2. In addition, as the snowplows are huge trucks, the number of u-turns needs to be minimized as well.

In this paper, we consider the problem of computing efficient routes for a snowplow driver. The road network of a town is represented by a directed graph *G*. The vertices of this graph are the road junctions and the edges are the road segments. Each road segment has an associated 'length' and a 'priority'. Our current implementation does not support multiple parallel edges. We assume that the vertices are numbered *0, 1,* etc. with *0* as the depot. A 'route-map' is simply a closed walk, i.e. a sequence $W = \{v_0, v_1, \dots v_k\}$ of vertices, where each $(v_i\ v_{i+1})$ is an edge, with $v_0 = v_k = 0$ and each edge of the graph appears at least once in *W*. Note that a walk in a graph allows both repeated vertices and edges. A subsequence of the form $\{v_i, v_j, v_i\}$ constitutes a u-turn. The distance covered by a walk, *d(W)* is the sum of the lengths of all the edges present in *W*. If an edge $e_1$ with priority $p_1$ occurs before an edge $e_2$ with priority $p_2$ in *W* and $p_1 > p_2$ (i.e. $e_1$ is less important than $e_2$) it constitutes a priority misplacement. The severity of this misplacement is $(p_1 - p_2)$. For each edge *e* in the walk *W*, with priority *p*, the priority misplacement index (PMI) of *e* is simply the sum $\sum(p - p_i)$ for all edges $e_i$ with priority $p_i$ in *W* that occur before *e* and $p > p_i$. The misplacement index *m(W)* of the walk *W* is the sum of the priority misplacement indices of all edges in *W*. The number of u-turns is denoted by *u(W)*. We have designed and implemented Genetic Algorithms (GA)-based algorithms to compute efficient paths that minimize *d(W), u(W)* and the *m(W)*. We have implemented our algorithms in a Java application called GASPR. This package is implemented using an object-oriented Java framework [9, 10] that we created earlier. Experimentation with graphs of varying sizes has produced very efficient solutions.

This paper is organized as follows: We begin with a short literature survey, and specify the assumptions we make on the type of graphs that our software can handle. We then present a greedy algorithm that computes a single complete walk for the graph and discuss its post-processing strategy. A brief introduction to the GA concepts is presented subsequently. This is followed by a description of how the GA is customized for our specific problem. Experimental results are presented in the following section. Finally, we provide some conclusions and plans for future work.

## II. LITERATURE REVIEW

Our problem is related to the Chinese Postman Problem (CPP) [3]. Dror *et al* [3] consider the cases in which a 'linear precedence relation' exists among priority classes and establish conditions under which a feasible solution exists. Ghiani and Improta [5] study the problem in which a linear precedence relation exists – called the Hierarchical Chinese Postman Problem (HCPP) – and propose a lower-complexity algorithm for circuit computation. Kazemi *et al* [6] have studied the problem of computing a minimum cost traversal of a road network that visits the entire road network. This problem is called the Optimal Traversal Problem (OTP). Eiselt *et al* [4] present a general survey of arc routing problems. They consider several versions of the problem such as undirected CPP, directed CPP and mixed CPP. Cabral *et al* [1] have proposed an algorithm to solve the HCPP by transforming it into a Rural Postman Problem (RPP). In a rural postman problem the edges are classified as 'required' or 'non-required', and only the required edges need be covered. These authors report an implementation of their algorithm and present experimental results on a real-world graph. Toobaie [12] consider the emergency truck routing problem for salting operations. While most papers consider the case with only one postman traversing the graph, Osterhues *et al* [7] consider the *k*-CPP problem, where there are *k* postmen who start and return to the base. The objective is not to minimize the total distance covered by all postmen, but to minimize the maximum distance covered by any postman. Campbell *et al* [2] study a more general problem of 'snow disposal site location and sector assignment' and apply it in the context of an urban snow removal problem for the City of Montreal. The problem we are addressing seeks to minimize the deadhead distance, number of u-turns and the misplacement index.

## III. PREPROCESSING THE GRAPH

We assume that the input graph has no parallel edges or self-loops and has been pre-processed, i.e. connected components are already identified. We also assume that we are dealing with a single connected component. If this is not the case, graph theoretical algorithms should be used to compute such components and the method we outline should then be applied to each component separately. Further, we assume that every vertex has at least one incoming edge and one out-going edge. This assumption is needed to ensure that our walks don't get stuck in a dead-end, and are able to return to the depot. Also, the current version of our software assumes that there is only one plow. The problem of allocating various sections of the graph to each truck is manually handled. We are developing different GA algorithms to perform optimal allocation and route computation for multiple plows.

## IV. A GREEDY ALGORITHM

We use a simple 'greedy' algorithm to compute a single complete walk that starts and ends at vertex 0. The walk is constructed as a sequence of vertices which initially begins as {0}. At every step, the algorithm appends a new vertex to the partially constructed walk. The next vertex to be appended is selected from the list of vertices that are directly connected to the last vertex on the partially constructed walk. The best vertex is selected using only local information by the "bestVertex" algorithm described below. This process repeats until all edges are covered and the walk ends at vertex 0.

The 'visit count' of a vertex is the number of times the vertex has been visited in the partial walk. The 'traverse count' of an edge is the number of times it appears in the walk. Let *NV* and *NE* be the number of vertices and edges in the graph respectively. A linear array called *VVC* (vertex visit count) of length *NV* keeps track of the vertex visits. A two-dimensional array called *ETC* (edge traverse count) with *NE* rows and 3 columns keeps track of the edge visits. In each row of *ETC*, the first column is the source vertex of the edge, the second column is the destination and the third column is the number of traverses. We assume that the graph *G*, and the data structures *VVC* and *ETC* are globally available. We first present the bestVertex algorithm that decides the next best vertex chosen to extend the walk.

```
Algorithm: bestVertex(partial walk W, array A of
        vertices adjacent to the last vertex in W)
    1. b = A[0]; f = last vertex in W,
    2. for each vertex v in A
           a. if (better(v, b)) then b = v.
```

Let $f$ be the last vertex in $W$ and $v_i$ and $v_j$ be two vertices in $A$. We use the following strategy to compare the edges $(f, v_i)$ , $(f, v_j)$ and determine if $v_i$ is better than $v_j$:

```
Algorithm: better (vertex $v_i$, vertex $v_j$)
    1. If $v_i$ causes a u-turn and $v_j$ does not, return false.
       If $v_j$ causes a u-turn and $v_i$ does not, return true.
    2. If $ETC(v_i) < ETC(v_j)$, return true. If $ETC(v_i) >$
       $ETC(v_j)$, return false.
    3. If $priority(v_i) < priority(v_j)$, return true. If
       $priority(v_i) > priority(v_j)$, return false.
    4. If $length(v_i) > length(v_j)$, return true. If $length(v_i)$
       $< length(v_j)$, return false.
    5. If $VVC(v_i) < VVC(v_j)$, return true. If $VVC(v_i) >$
       $VVC(v_j)$, return false.
    6. Return true.
```

The 'greedy' algorithm "computeWalk" is presented below.

```
Algorithm: computeWalk
    1. Set $W = \{0\}$
    2. Initialize VVC to an array of zeros. For each edge
       $(v_i, v_j)$ in G, add a new row to ETC with values
       $(v_i, v_j, 0)$.
    3. While all edges are not covered and not back at
       vertex 0
      a. Let f be the last vertex in W.
      b. Let A be the array of vertices that are
         connected to f by an edge in G.
      c. Let g = bestVertex (W, A).
      d. Append vertex g to W. Increment VVC[g] by
         1. Find the row k of ETC for which ETC[k][0]
         equals f, ETC[k][1] equals g, and increment
         ETC[k][2] by 1.
```

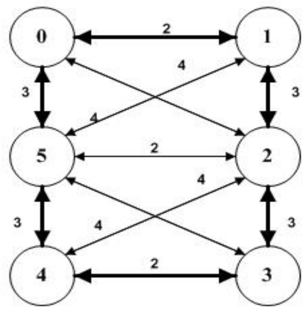

**Figure 1: Graph G1**

**Example 1**: Consider the graph G1 in Fig. 1. The length of each edge is indicated by the number next to it. Thick edges are arterial (priority = 1) and thin edges are neighborhood (priority = 2). Consider the walk *W*= 0-2-3-4-5-0-1-2-4-3-2-1-5-1-0-5-2-5-3-5-4-2-0 (The hyphens have been added for readability). It has *d(W)* = 68, *u(W)* = 3 and *m(W)* = 28.

## V. POST-PROCESSING OF THE WALK

The assumption that there are no 'sink' vertices ensures that there is always a complete walk for the graph. However, the walks constructed by the greedy algorithm above can have many repeated edges, u-turns and priority misplacements. The following post-processing algorithms attempt to improve the walks by eliminating some redundant cycles and sub-walk rearrangements. Eliminating redundant cycles reduces the total distance and sub-walk rearrangement reduces the misplacement index.

**Redundant Cycle Elimination:** Let $c = \{v_i, \ldots, v_i\}$ be a cycle in *W*. 'Eliminating' the cycle *c* consists of replacing the subsequence $\{v_i, \ldots, v_i\}$ by the single vertex $\{v_i\}$. A cycle *c* is redundant if the walk is still complete (i.e. covers all edges) after *c* is eliminated. The algorithm starts at the right end of the walk and looks for a redundant cycle. If such a cycle is found, it is eliminated from *W*. The process continues until no more redundant cycles are found.

**Algorithm: EliminateCycles** (complete walk *W*)

```
1. thereIsALoop = true
2. while (thereIsALoop)
   a. thereIsALoop = false
   b. Starting from the right end of W and scanning left,
      find the first subsequence that is a loop: $\{v_i, \ldots, v_i\}$.
      thereIsALoop = true. If there is no such loop, exit.
   c. If the loop $\{v_i, \ldots, v_i\}$ is redundant, replace the
      subsequence $\{v_i, \ldots, v_i\}$ in W with just the single
      node $v_i$.
```

**Example 2:** Consider the graph in Fig. 1 again. The walk: 0-5-4-5-2-0-1-2-1-0-2-5-3-5-**3-2-3**-5-1-5-**3-2-3**-4-2-4-3-5-0 is a complete walk that has cycles that can be eliminated. For example, consider the two cycles 3-2-3 (shown in the above sequence in bold). If one of the 3-2-3's is replaced by a simple 3, the resulting walk is still complete. On the other hand, the cycle 4-2-4 cannot be replaced by 4 because the edges 4-2 and 2-4 do not occur anywhere else in the walk. The post-process algorithm repeatedly finds such cycles and eliminates them if possible.

**Sub-walk Rearrangement:** Let $s_1 = \{v_i, \ldots, v_j\}$, and $s_2 = \{v_i, \ldots, v_j\}$ be sub-walks of *W* that begin and end with the same vertices; furthermore, assume that $s_1$ and $s_2$ have no vertices in common and $s_2$ occurs to the right of $s_1$ in *W*. We define an "isBetter" algorithm that compares the priorities of the edges in $s_1$ and $s_2$ lexicographically to decide which is better. If $s_2$ is better than $s_1$ then these two subsequences are swapped within *W*. This process is repeated for other pairs of sub-walks, and for each feasible size of the sub-walk.

**Algorithm: isBetter** (subwalk $s_2$, subwalk $s_1$)

```
1. Let $p_1$ and $p_2$ be ordered numerical arrays
   containing the corresponding priorities of the
   edges in $s_1$ and $s_2$ respectively.
2. for k = 1 to min(size of $p_1$, size of $s_2$)
      a. if ($p_2$[k] < $p_1$[k]) return true;
      b. if ($p_2$[k] > $p_1$[k]) return false;
3. Return false.
```

We now present the essential elements of the "RearrangeSubwalks" algorithm. Some of the finer details are skipped in the interest of brevity.

**Algorithm: RearrangeSubwalks** (complete walk *W*)

```
Repeat the following steps for size = 1 to ½ * length of W.
1. start = 0.
2. Find a subsequence $s_1$ of length size beginning at
   start. Find another subsequence $s_2$ after the end of $s_1$
   which begins and ends with the same vertices as $s_1$. If
   there is no such subsequence, go to step 4.
3. If isBetter($s_2$ , $s_1$), then swap $s_2$ and $s_1$ in W.
4. Increment start by 1. If a subsequence of length size
   exists beginning at the new value of start, go to step
   2; otherwise, increment size by 1 and continue with
   the next iteration of the main loop.
```

**Example 3:** Consider the rather poor-quality walk *W* = 0-5-4-3-2-1-0-1-2-3-4-5-**0-2-4-5-**1-2-**0-5-**3-2-5-4-2-1-5-2-3-5-0 in $G_1$. This has a misplacement index = 46. Consider the subsequences $s_1$ = **0-2-4-5** and $s_2$ = **0-5.** Their priority lists are $p_1$ = {2, 2, 1} and $p_2$ = {1} respectively. We find that $s_2$ is better than $s_1$. After swapping these two subsequences we get W = 0-5-4-3-2-1-0-1-2-3-4-5-**0-5**-1-2-**0-2-4-5**-3-2-5-4-2-1-5-2-3-5-0 which has a misplacement index of 42.

## VI. GENETIC ALGORITHMS

Genetic Algorithms [8,9,10,11] (GA) are used to compute good solutions to NP-Hard problems (i.e., problems for which there are no known techniques that can compute a solution in polynomial time). It is well known that all available algorithms to solve such problems take time whose order of magnitude is exponential in terms of the input data size. Evolutionary techniques such as GA are used to compute satisfactory solutions to such problems in a reasonable amount of time, especially for large input data sizes.

In a GA, candidate solutions to a problem are modeled as *individuals* in a *population*. Depending on the problem, the population of a pre-determined size ($N$) is initialized - either randomly or strategically. Each individual has a *fitness value* which represents its quality as a solution. We seek to find the individual with the best fitness. The population goes through a pre-determined number of cycles of evolution. In each cycle, a series of *genetic operators* are applied to modify the population. The standard operators are *evaluate, select, crossover* and *mutate*. Other operators to prevent the population from dying out may be added. In our case, we sort the population in the increasing order of fitness, and choose a preset percentage of top performing individuals for crossover. They are paired randomly and crossed to produce a child population. Some members of the child population go through mutation. The parent and child populations are merged and sorted. The top $N$ are chosen as the next population.

**Individual and Fitness Function:** Individuals in a GA are candidate solutions. To apply GA to our problem we first identify the individual as simply a complete walk that begins and ends at 0. We define the fitness of an individual $W$ as:

$$\text{Fitness}(W) = D * d(W) + U * u(W) * + M * m(W)$$

where $d(W)$, $u(W)$ and $m(W)$ are the total distance covered, the number of u-turns and the misplacement index of $W$ respectively. $D$, $U$ and $M$ are user-selected weights.

**Initialization:** We use a variant of the greedy algorithm of Section IV to generate the initial population randomly. The "computeRandomWalk" algorithm proceeds in a manner similar to the "computeWalk" algorithm. The difference is that, while "computeWalk" is deterministic since it chooses the 'best' vertex from the set of available vertices, "computeRandomWalk" chooses a vertex randomly. The initialization process simply calls this algorithm $N$ times to create an initial population of size $N$.

**Algorithm: computeRandomWalk**
```
1. Set $W = \{0\}$
2. While (all edges are not covered OR not back at
   vertex 0)
   a. Let $f$ be the last vertex in $W$.
   b. Let $A$ be the array of vertices that are
      connected to $f$ by an edge in $G$.
   c. Let $g$ = a randomly chosen vertex from A.
   d. Append the vertex $g$ to $W$.
```

**Evaluation and Selection:** After the initialization, each individual is evaluated using the fitness function defined above. Our *selection strategy* consists of sorting the population in the increasing order of their fitness and selecting the top x% of the population. The choice of percentage is a parameter that is specified in a configuration file and can easily be changed without need to re-compile the software.

**Crossover:** Pairs of individuals are chosen from the selected population to undergo crossover. Our strategy for pairing-up individuals is random. To perform crossover on two individuals $I_1$ and $I_2$, we choose a random start point $i$ and an end point $j$ and identify the corresponding subsequence $v_i$ to $v_j$ in $I_1$. We find a subsequence in $I_2$ that begins at $v_i$ and ends at $v_j$ and swap the two subsequences. This process ensures that the resulting sequences are valid walks. However, they may not be complete. Another variation of the "computeWalk" algorithm called "finishPartialWalk" helps us in completing these walks.

**Algorithm: finishPartialWalk** (partial walk *prefix*)
This algorithm is the same as the "computeWalk" algorithm, except that instead of starting with $W = \{0\}$, this begins with $W$ = *prefix* and initializes the *VVC* and *ETC* arrays based on *prefix*. After the walk is completed, we use the redundant cycle elimination and sub-walk rearrangement algorithms to improve it.

**Algorithm: Crossover** (complete walk *W1*, complete walk *W2*)
```
1. Select a random subsequence $v_i$ to $v_j$ in $W_1$
2. Find the first subsequence in $W_2$ that begins with $v_i$
   and ends with $v_j$. Swap the two subsequences of $W_1$
   and $W_2$, creating two new walks.
3. Complete the new walks using the
   "finishPartialWalk" algorithm.
```

**Mutation:** After applying the crossover operator, individuals might undergo mutation as well. To perform mutation, we select a random subsequence $v_i$ to $v_j$ in walk $W$. We then partition $W$ into three parts: *prefix* (0 to $v_i$), *middle* ($v_{i+1}$ to $v_{j-1}$) and a *suffix* ($v_j$ to end). We delete the middle and compute another sub-walk that connects $v_i$ to $v_j$. We insert this in the middle to create the new mutated individual. Once again, a variation of the "computeWalk" algorithm called "patchPartialWalks" with post-processing helps to accomplish this task.

**Algorithm: patchPartialWalks** (partial walk *prefix*, partial walk *suffix*): This algorithm is very similar to the "finishPartialWalk" algorithm. It finds a walk that joins the last vertex of *prefix* to the first vertex of *suffix*. Once the walk is completed, it is improved using the same post-processing algorithms.

**Algorithm: Mutate** (complete walk *W*)

1. Select a random subsequence $v_i$ to $v_j$ in $W$.
2. Find the subsequences of $W$: *prefix* = 0 to $v_i$, *middle* = $v_{i+1}$ to $v_{j-1}$ and *suffix* = $v_j$ to end.
3. Complete the walk using the "patchPartialWalks" algorithm.

**Replenishment:** At the end of each GA cycle, we examine the current population for diversity. If there are duplicate individuals, we eliminate one of them and replace it with a randomly generated individual using the "computeRandomWalk" algorithm.

## VII. EXPERIMENTAL RESULTS

Many graphs with varying number of vertices and edges were chosen for experimentation. We have used the weights $D = U = M = 1$ in the following experiments. In Table 1, *NV, NE* and *MC* represent the number of vertices and edges in a graph and minimum distance covered by a complete walk respectively. The 'deadhead' of a route $W$ is defined as $d(W) - MC$, i.e. the extra distance covered by $W$ beyond the minimum cover. Thus, if $d(W)$ equals $MC$, then the deadhead is zero. The diagrams for the experimental graphs are shown in Figs. 1, 2 and 3. Thick edges have a higher priority (1) and thin ones have a lower priority (2).

**Graph G1** (Fig. 1: $NV = 6$, $NE = 20$, $MC = 68$)**:** As there is no priority 2 edge adjacent to vertex 0, it is impossible to have $m(W) = 0$. The best we can hope for is deadhead = 0 and $u(W) = 0$. The best solution produced by our GA is: 0-1-2-3-4-**5-0-5**-4-3-2-1-0-2-**5-3-5-1-5-2-4-2**-0. It has $d(W) = 68$ (i.e. deadhead = 0), $u(W) = 4$ and $m(W) = 0$, giving a total fitness of 72. It was attained at iteration 9.

**Graph G2** (Fig. 2: $NV$=20, $NE$=54, $MC$=186) The graph is so constructed that the sub-graph induced by vertices 0 – 9 has only priority 1 edges, and that with vertices 10 – 19 has only priority 2 edges. These two sub-graphs are joined by a single bridge with priority 2. The best solution produced by our GA is optimal: 0-1-2-3-4-7-6-5-4-3-8-7-4-5-6-7-8-9-2-1-0-9-8-3-2-9-0-10-11-18-17-12-11-10-19-18-11-12-13-16-15-14-13-12-17-16-13-14-15-16-17-18-19-10-0. It has $d(W)$ = 186 (i.e. deadhead = 0), $u(W)$ = 0 and $m(W)$ = 0, giving a total fitness of 186. It was attained at iteration 28.

**Graph G3** (Fig. 3: $NV$=32, $NE$=88, $MC$=378) The best solution produced by our GA: 0-1-2-6-7-31-30-31-7-3-2-1-5-6-2-3-7-6-13-30-29-10-29-30-13-6-5-1-0-4-28-29-28-4-5-10-9-8-16-22-23-17-16-8-9-17-23-22-16-17-9-10-11-12-13-14-20-26-27-21-15-14-15-21-27-26-20-21-20-14-13-12-19-18-24-25-19-12-11-18-19-25-24-18-11-10-5-4-0 has $d(W)$ = 378, $u(W)$ =5, and $m(W)$ = 150, giving a fitness of 533 attained at iteration 56.

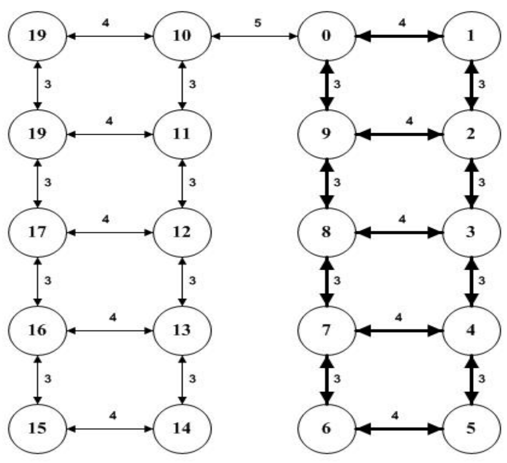

**Figure 2: Graph G2**

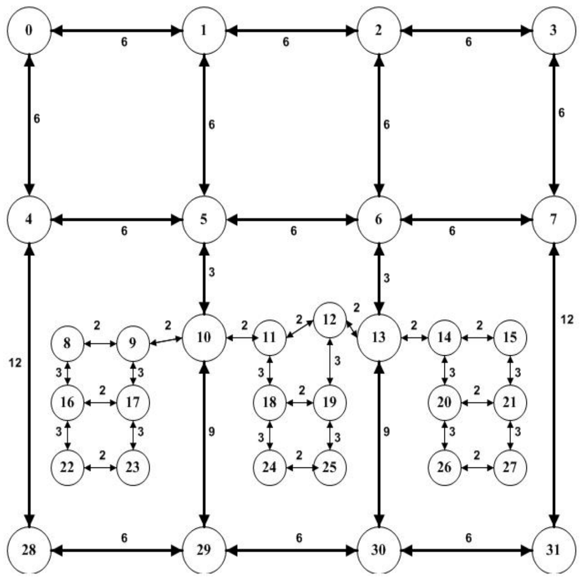

**Figure 3: Graph G3**

**Table 1: Experimental Results**

| | *Graph Characteristics* | | | *GA Solution Characteristics* | | | |
|---|---|---|---|---|---|---|---|
| *Id* | *NV* | *NE* | *MC* | *dead-head* | *u(W)* | *m(W)* | *Iter* |
| Graph10-1 | 11 | 36 | 360 | 0 | 0 | 0 | 22 |
| Graph10-5 | 11 | 76 | 760 | 0 | 0 | 0 | 43 |
| Graphjtt6 | 13 | 31 | 292 | 0 | 4 | 32 | 110 |
| Perin-nbhd | 15 | 38 | 380 | 0 | 5 | 80 | 4 |
| Graph10-2 | 15 | 52 | 520 | 0 | 0 | 0 | 45 |
| Graph11-2 | 16 | 48 | 480 | 0 | 0 | 0 | 31 |
| Graph10-3 | 16 | 54 | 540 | 0 | 0 | 0 | 58 |
| Graph11-3 | 18 | 58 | 580 | 0 | 0 | 0 | 64 |
| Graph10-4 | 22 | 76 | 760 | 0 | 1 | 0 | 97 |
| G5 | 16 | 66 | 258 | 76 | 24 | 49 | 764 |
| Graph9 | 25 | 76 | 760 | 0 | 0 | 0 | 74 |
| G6 | 25 | 96 | 356 | 0 | 13 | 53 | 137 |
| Fpt-major | 28 | 82 | 820 | 0 | 1 | 0 | 193 |
| Brockport | 48 | 154 | 356 | 0 | 15 | 0 | 531 |
| Parma | 71 | 218 | 2180 | 20 | 4 | 0 | 148 |

.

**Experiments with other graphs:** We have conducted experiments with many graphs created from data using real small towns. The results are presented in Table 1. Graph characteristics shown are the number of vertices, edges and minimum cover. Solution characteristics displayed are the amount of deadhead, number of u-turns, misplacement index and the GA iteration at which this solution was attained.

## VIII. CONCLUSIONS AND FURTHER RESEARCH

In this paper, we have presented a GA solution to a snow-plow routing problem. Our application attempts to optimize on three parameters: distance covered, number of u-turns and misplacement index. Our software application, GASPR, computes near-optimal solutions to reasonably large problems in moderate amount of time. The real-world problem, however, is much more complicated than this. Additional problems include the following: optimal partitioning of the graph so that each plow can service one partition, optimal utilization of personnel, equipment etc. A closely related problem is that of side-walk snow clearing. In this problem, even though edges are bi-directional, they need not be cleared in both directions. The segments are narrow enough to assume that clearing them in one direction is sufficient. Further, the plows are rather small and making u-turns is not that difficult. We plan to explore the sidewalk snow clearing problem and also the application of other evolutionary techniques such as scatter search and particle swarm optimization for these problems.

## REFERENCES

[1] Cabral C., Gendreau M., Ghiani, G., Laporte, G. *Solving the Hierarchical Chinese Postman Problem as a rural postman Problem,* European Journal of Operational Research, 155 (2004) 44-50

[2] Campbell, J., Langevin, A., *Operations Management for Urban Snow Removal and Disposal,* Transpn. Res. – A, Col. 29A, No. 5, 359-370, 1995.

[3] Dror, M., Stern, H., Trudeau, P., *Postman Tour on a Graph with Precedence Relation on Arcs,* Networks, Vol. 17, (1987) 283-294.

[4] Eiselt, H., Gendreau, M., Laporte, G., *Arc Routing Problems, Part I: The Chinese Postman Problem,* Operations Research, Vol. 43, No. 2, 1995, 231-242.

[5] Ghiani, G., Improta, G., *An Algorithm for the Hierarchical Chinese Postman Problem,* Operations Research Letters, Vol. 26, (2000) 27-32.

[6] Kazemi, L., Shahabi, C., Sharifzhdeh, M., Vincent, L., *Optimal Traversal Planning in Road Networks with Navigational Constraints,* Proc. 15th Annual Symp. On GIS, Seattle WA, 2007, pp 1-8

[7] Osterhues, A., Mariak, F., *On variants of the k-Chinese Postman Problem,* Operations Research und Wirtschaftinformatik, No. 30, 2005, http://www.wiso.tu-dortmund.de/wiso/or/Medienpool/publikationen/dispap30.pdf accessed 12-15-10.

[8] Luger, G. *Artificial Intelligence: Structures and Strategies for Complex Problem Solving,* Addison-Wesley, 6th Edition, 2008.

[9] Rao, T.M., Mitra, S. *Architecture of a Java Framework for Developing Genetic Algorithms in AI class* in the Journal of Computing in Colleges. Vol. 25 , Issue 6 (2010), 93-99.

[10] Rao, T.M., Rao R., Sahin, F., Tillett J. *Evolutionary Algorithms for the Edge Biconnectivity Augmentation Problem,* IEEE Conference on Systems, Man and Cybernetics, Oct. 5-8, (2003) Washington D.C. pp 1955-1960.

[11] Russel, S. and Norvig, P. *Artificial Intelligence, A Modern Approach,* Prentice Hall, 2nd Edition, 2003.

[12] Toobaie, S., Haghani, A. *Minimal Arc Partitioning Problem,* Journal of the Transportation Research Board, No. 1882, TRB, 2004, pp167-175.