Plowing the Streets of Helena An Exploration of Algorithmic Solutions to Connected Graph Traversal Optimization

Kylan Johnson
Carroll College, Helena, MT

Follow this and additional works at: https://scholars.carroll.edu/mathengcompsci_theses

Part of the Applied Mathematics Commons, Theory and Algorithms Commons, and the Urban Studies and Planning Commons

Recommended Citation
https://scholars.carroll.edu/mathengcompsci_theses/81

This Thesis is brought to you for free and open access by the Mathematics, Engineering and Computer Science at Carroll Scholars. It has been accepted for inclusion in Mathematics, Engineering and Computer Science Undergraduate Theses by an authorized administrator of Carroll Scholars. For more information, please contact tkratz@carroll.edu.
Plowing the Streets of Helena
An Exploration of Algorithmic Solutions
to Connected Graph Traversal Optimization

by

Kylan Neal Johnson

April 24, 2003

Submitted in Partial Fulfillment of the Requirements for Graduation with Honors
to the Department of Mathematics, Engineering, and Computer Science, Carroll
College, Helena, Montana.

Dr. Mark Parker – Director

Professor Phil Rose – Reader

Joan Stottlemeyer – Reader
# Table Of contents

Abstract................................................................................................................................. iii
Acknowledgements ................................................................................................................ iv
Introduction .................................................................................................................................. 1

Chapter One – Solution Methods ......................................................................................... 5

  OVERVIEW ................................................................................................................................. 5
  CRITERIA TO OPTIMIZE .......................................................................................................... 5
  ASSUMPTIONS .......................................................................................................................... 7
  DESCRIPTION OF ALGORITHMS
    Basic Priority ......................................................................................................................... 9
    Look Ahead ............................................................................................................................ 9
  PLOWING STRATEGY ............................................................................................................. 12

Chapter 2 – Computer Implementation ................................................................................. 13

  Using the Main Menu ............................................................................................................. 13
  USING THE PLOWER ............................................................................................................. 14
  PLLOWING VISUALS ............................................................................................................. 16
  USING THE MAPPER ............................................................................................................. 17
  THE MAPPER BUTTONS ......................................................................................................... 17
  EDITING VISUALS ................................................................................................................ 20

Chapter 3 – Structure ............................................................................................................. 21

  STRUCTURE COMPOSITION .................................................................................................. 21

Chapter 4 – Class Functions .................................................................................................. 24

  STREET ..................................................................................................................................... 24
  INTERSECTION ....................................................................................................................... 25
  SNOWPLOW ........................................................................................................................... 26
  DRIVER ...................................................................................................................................... 28
  MAPPER ................................................................................................................................... 30
  MENU ........................................................................................................................................ 31

Chapter 5 – Analysis of Algorithmic Results ................................................................. 32

  MEASURING PERFORMANCE ............................................................................................... 32
  USAGE OF ALGORITHMS ........................................................................................................ 34
  ANALYSIS OF THE MAPS ....................................................................................................... 36
  ALGORITHMIC FLAWS ........................................................................................................... 38

Chapter 6 – Future Works and Improvements ......................................................................... 40

Bibliography ............................................................................................................................... 42

Appendix – The Maps ................................................................................................................. 43
Abstract

Due to fierce winter conditions, Montana streets can become blocked with snow and ice. It is up to the snowplow operators to ensure the safety of travelers across the state by plowing and sanding roads before traffic becomes congested and accidents occur. Through algorithms and graph theory a plan can be made to optimally plow the streets so that snow and ice do not interfere with the daily commute.

Through the use of two algorithms, this thesis addresses the task of optimal plowing. The first is a Greedy Algorithm, the other a Look Ahead algorithm. The Greedy Algorithm assesses the priorities of every street connected to the current intersection and chooses to plow the highest priority street. The Look Ahead algorithm allows the user to input how many streets beyond local streets are to be considered. For example, the program could look many streets into the future to seek out a street that may not have been plowed yet but is surrounded by other streets that are. Algorithms can be used in any user-defined sequence.

Users are also able to create, edit, and save their own maps through a graphical user interface. The editing program allows users to create and delete both streets and intersections as well as edit properties such as position and name on existing intersections and streets. This interface allows the user to enter more complex maps, such as the sectors of Helena for more realistic optimization.
Acknowledgements

I would like to thank many people for helping me with my thesis and for their continued encouragement as long as I’ve known them.

- **Dr. Mark Parker** – Thank you for being my advisor, giving me guidance, and coping with my numerous revisions.

- **Professor Phil Rose** – Thank you for being a reader of my thesis and for everything you’ve done for me these last four years.

- **Joan Stottlemeyer** – Thank you for being my reader and coping with the oddities of code in English terms!

- **Ann Marie Miner** – Thank you so much for being understanding of the many, many long nights coding and for letting me carry my laptop everywhere!

- **My Parents** – Thank you for all your support you’ve *always* given.
Introduction

Optimal snowplow routing is yet another application of combinatorics and graph theory. Streets and Intersections can be represented by numbers on a computer to effectively apply modern technology and mathematics in order to determine optimal snowplow paths.

The task of optimally routing a snowplow around city streets is not easy. The problem is NP-Complete, which means that there is no known deterministic method that will run in polynomial-time to solve such a problem. The fastest algorithms currently for NP-Complete used are no better than exponential or \( O(k^m) \) where \( k \) and \( m \) are real numbers.

A problem is NP-Complete when it is shown to hold properties of existing NP-Complete problems, a process called reduction. In this case, snowplow routing can be reduced to an NP-Complete problem called the Chinese Postman Problem (CPP). The CPP asks for the shortest tour of a graph, which visits each edge at least once. One can easily see the analogy to a postman who must deliver a docket of letters along specific paths the fastest way possible. Since my snowplow routing problem and CPP attempt similar goals using similar criteria, the task of snowplow routing is reduced to CPP and is in NP-Complete.

In addition to being NP-Complete, the task of optimal snowplow routing can be thought of as a form of graph traversal. A map can be expressed as a directed multi-graph where the streets are the edges and the intersections are the nodes of the graph. A directed multi-graph is a graph where the nodes have multiple edges between them, each
with a direction. Any street has between one and eight lanes with each lane an edge in our directed graph system.

The goal of a snowplow path is to travel every lane once while still traversing every lane on the graph. This requirement resembles an Eulerian Path. Recall that an Eulerian Path is a path that travels each edge of a graph once and only once while still traveling every edge. Because of these similar properties, I consider an Eulerian Snowplow path to be optimal because we must visit every lane once but multiple times are a waste. Thus, the problem of snowplow routing is just finding an Eulerian Path on our graph.

It is easy to discover the mere existence of an Eulerian Path, given a graph. If the graph contains only two nodes that connect an odd number of edges, the graph has an Eulerian Path. However, we are not asking for an answer to the existence of an optimal route but an actual route that will not take too long. If we just wanted to know the existence of such a path, a reduction to an Eulerian path could yield our yes or no answer. The reduction could show the existence of an Eulerian Path and thus a snowplow path, regardless of cost. It is highly likely, though, that the graph will not contain an Eulerian Path. The probability of only two nodes connecting an odd number of vertices is small given the number of nodes on a city map and the basic layout of city design.

Nevertheless, I wish to actually find a path and minimize the time spent plowing, so I have developed a heuristic that will yield a sub-optimal solution. Sub-optimal algorithms are algorithms that may yield optimal results but should come close to the optimal solution in the worst case. Given the level of difficulty of such a problem, finding an answer that is pretty good (sub-optimal) is a very acceptable solution, and a
Plowing the Streets of Helena

decent solution is better than no solution at all. To achieve a sub-optimal solution, I allow the streets to be traveled. A Street is traveled if there are no more lanes on that street that need to be plowed.

There are a few types of algorithms to choose from. The first and most straightforward is the Greedy Algorithm. A Greedy Algorithm is a flavor of graph traversal algorithm that makes decisions based solely on the cost of the edges that could be selected next. Thus, if a plow is at an intersection, it will choose the street that has the highest priority that hasn’t been fully plowed. Should all the streets be fully plowed, a Greedy Algorithm will generally pick a street at random, as it has no future knowledge of map status. My Greedy Algorithm is called Basic Priority and follows the same guidelines. The user can select an intersection where the plow will begin. From there the plow will make decisions based solely on priority with 1 being the highest and 3 being the lowest ordinal ranking.

A shortcoming of Greedy Algorithms that must be addressed is the lack of intelligence, specifically the algorithm’s inability to see into the future. I have created an algorithm called Look Ahead, a more intelligent algorithm that can look a user-defined number of streets into the future in order to determine the path of highest priority. The goal of the algorithm is to find a successive number of streets for each potential street choice that offers the highest priority. Lower priority streets may then be plowed before high priority streets as they may lead to a string of high priority streets versus high priority streets that may lead to priority 3 streets. Once Look Ahead has accumulated the totals for each of its current local paths, it then selects the highest priority and proceeds with the plowing. In Look Ahead, cycles are allowed. It is realistic to say that a plow
should circle a city block if that city block holds high enough priority. The plow will only make the cycle if the algorithm concludes that the cycle remains the best route. This should generally be the case if the algorithm looks far enough into the future.
Chapter One – Solution Methods

This chapter gives an overview of the criteria and assumptions used to optimize a map. Then, a brief summary of both algorithms is given where the general goals and logic of each algorithm are discussed. Last, a strategy to optimize is given and will be described in more depth in Chapter 5.

Overview

Our objective in plowing is the completion of one goal: to efficiently plow every street on the map. The concept of “efficiently” is somewhat unclear and must be defined for my specific problem as snowplow routes can be optimized according to different criteria. We could attempt to plow in the shortest time where each street takes specific times to be plowed and traveled. Road attributes such as grade, direction, and traffic can also be used as costs for optimization. My algorithms look solely at priority. The assignment of road priorities considers the many aspects of roads. For example, residential roads do not get much traffic relative to downtown streets and are thus prioritized low. However, streets around schools, which may be located in residential areas, have a higher traffic total and need a higher priority. These special cases allow priorities to encompass the special properties streets possess.

Criteria to Optimize

Every optimization involves the weight of costs versus rewards. To plow efficiently, one must weigh the rewards that come with each decision of plowing and
attempt to combine the most rewards with the fewest costs. The major reward criteria are as follows:

- **Space Cleared** – Every Street plowed is one more street that we don’t have to plow. We want to plow as many streets in a row as we can, because if we travel a street, we in a sense wasted an iteration. This problem is somewhat hard to avoid since pockets of lower priority streets will pop up due to plowing high priority streets first. This reward justifies plowing a lower priority street just because we’re in the area, a common practice in actual snowplowing.

- **Time Spent** – Algorithms can optimize according to time spent traveling a specific choice. My algorithms do not consider time that the plow would spend plowing, but the analysis of my algorithms includes the factor of time when choosing an appropriate algorithm for a plowing goal.

- **Straight Cost** – My particular problem emphasizes a street’s priority. Generally, the higher priority streets must be completed first and are thus given special consideration in every algorithm. Since priority 1 streets are the roads that most people travel, clearing them is essential.

Each kind of algorithm handles these criteria in its own way, and both have merits according to these criteria. Priority 1 streets must be clear, and Basic Priority will ensure they are clear because the only basis of plowing a street is its priority. The algorithm will naturally seek out high priority streets. Look Ahead combines Straight Cost and Space Cleared. Instead of one path to consider, numerous paths are conglomerated to form a single priority. Future choices are considered in hopes of finding the highest priority path
over time. As stated earlier, time spent is only considered when analyzing the time the algorithm took to calculate a path.

**Assumptions**

When considering the task of graph traversal, we have many variables to consider: plow width, plow speed, starting point, weather conditions, and numerous other variables. The following assumptions are implemented in my algorithms:

1. The snowplow used will be the standard Mack truck plow. I will also assume that the snowplow can plow one street lane per pass. Hence, a four-lane street will take four passes. In addition, the plow makes perfect plows every time, meaning the driver will accomplish all that he was trained to do with regards to positioning and consistency in his plowing.

2. The plows will sand whenever necessary. Since sanding is assumed to have been done correctly, it is not considered when selecting a street.

3. The operational scenario is that of a community that has just seen a major storm but is not experiencing snow when plows are operating. I assume that the plow, on one pass, makes sufficient progress on that lane for a car to travel safely.

4. A plow can complete a designated route without stopping to fill up its gas tank. This assumption is reasonable due to the sheer size of the tank and the length of the respective snow routes in Helena. Whenever a plow is used, the previous driver preps the plow for usage, meaning the tank is filled with gas and the plow itself is clear of any damages so as to provide maximum plowing efficiency.
5. Each main road is one of three priorities: Major Snow Route (1), High Traffic Collectors (2), Problematic Streets (3). Other streets such as common residential streets and small streets such as alleyways are given no priority and thus not considered.

6. Unplowed streets will be given higher priority than plowed streets. An open priority 2 street will be given priority over a fully plowed priority 1 street, assuming there are no more priority 1 streets.

7. Lanes remaining will separate incomplete streets with the same priority. For example, should a plow come to an intersection where it has two choices, both priority 1 streets but one has been plowed, the unplowed street is taken. Should both streets be of equal priority and plowed, the decision will be algorithm dependant. If two or more streets have the same priority and are unplowed, the decision is also algorithm dependent.

8. A road has no direction, meaning that there is no such thing as a one-way street. Plows can travel in any direction.

Description of Algorithms

When traversing the graph, a snowplow will follow one of two algorithms: Basic Priority and Look Ahead. Both algorithms use two arrays defined in Snowplow to store choices and priorities, called “names” and “priority.”
Basic Priority

When the plow is using Basic Priority, the decisions are rather simple. When the plow is at an intersection, the algorithm will look at the possibilities given to the plow at that intersection. This decision excludes the street that the plow arrived on, as that choice is somewhat redundant and it is unreasonable for a plow to make a U-turn that easily. Basic Priority will base its next decision on straight priority of the local choices with the selection being the lowest ordinal number (e.g., 1). As stated in assumption 5, an unplowed street is given precedence over a plowed street. The first priority 1 street found in the list of candidates would be the one chosen. Should the plow come to an intersection that has all its streets plowed, the next street will be chosen by the C++ random function because the streets can all be thought of as equal.

The algorithm will start with a user designated intersection and find the first candidates for that intersection using a function called FirstChoices. Next, the algorithm will look for the highest priority street available, starting with priority 1s and moving to priority 3s. Should the algorithm find a street that hasn’t been plowed, it will plow the first street it finds that meets its current priority criteria. If no such street can be found, the street is picked at random, regardless of priority and is traveled rather than plowed.

Look Ahead

The purpose of the Look Ahead algorithm is to consider the options available beyond the current intersection and select the path that will produce the best overall results over many future plows. Look Ahead allows a snowplow to look forward a user-
defined number of streets to see what would happen if the plow traversed a particular street.

The graph can be thought of as a tree, with each node being an intersection, the root of the tree as the current intersection, and every branch of the tree is a choice that can be made after the previous branch has been traveled. One must perform a breadth first search of the tree, selecting the highest priority street at each level. As the branch selections are made, the priority of each selection is added to an original total for each possible selection of the root.

Figure 1 illustrates an example of a tree. Suppose the plow comes upon an intersection after plowing Benton with Euclid, priority 1 and Custer, priority 3. Euclid is the first obvious choice. But what if the plow knew that two streets away (two levels down the tree), were Logan and Hannabolt, two priority 1 streets whereas taking Euclid would yield Realto and Cali, two priority 3 streets. By taking a lower priority street, the plow nets a higher output. Not only does this algorithm make practical sense, but it will also emulate the actual decisions a snowplow operator might make. Experienced snowplowers will know the map they traverse and be able to think three or more streets ahead so that they can hit a shortcut.
The user can specify which intersection the algorithm starts, the number of streets it will travel or plow, and the number of streets that Look Ahead will look beyond the local choices. First, Look Ahead will "look" down every option of the current intersection. At every intersection looked at, the first street found with the highest priority is chosen and its priority added onto a running total for the original decision. When all branches of the original set of streets have been looked at, the algorithm sorts the totals to determine the lowest number and thus the highest priority path. If all streets have been plowed or there is a tie, the algorithm searches two more streets ahead than the user-defined number and chooses the last, lowest choice. I.e., if the priority choices were 15, 16, and 16, the street with the second 16 would be the choice.

Once a street has been appended to the list, its priority is incremented by four meaning a priority 1 street would now have a priority of 5. This incrementation decreases the appeal of that street to the algorithm as that street has already been seen if not fully plowed. Should Look Ahead not find a street to travel or plow, the possible streets are either dead ends or they have the same priority. Look Ahead then looks two additional streets beyond the user-defined look-ahead number to break the tie.

Figure 2 uses Look Ahead 2. Our street choices for our starting intersection are austin1, tyrone, and austin2. Since we are using Look Ahead 2, the algorithm will look two streets past the local choices. If we were to plow austin2 (2), we would then plow gary (3), then am (2) for a total of 7. If we were to plow tyrone (1), we

Figure 2 -- Look Ahead Example
would then plow felix1 (2), then am (2) for a total of 5. Last, if we were to plow austin1 (2), we would then plow powers (1), then shawn (1) for a total of 4. The algorithm will choose austin1 because it offers the best overall result.

**Plowing Strategy**

Through the combined use of Basic Priority and Look Ahead, a reasonable plowing route can be obtained. Each algorithm offers unique properties that, when combined, emulate a plow out on the roads.

Realistically, if a plow is in a somewhat desolate region of the map where there are few high priority streets left, other streets can be considered. Residential loops and distributor streets, though not as high priority, could still be plowed while the plow is in the vicinity and in real practice, a plow operator would take this type of route. This philosophy might detract from the speed at which the plow may operate; however, other optimization factors such as customer satisfaction and space cleared must be considered.

When beginning to plow, we can use Basic Priority to seek out high priority streets. In common configurations of city snow routes, there will usually be a string of successive priority 1 streets that must be plowed, thus making Basic Priority a good algorithmic choice. When that route has been exhausted, pockets of unplowed streets such as residential and minor contributory streets must be found and reached in the shortest distance possible. A solution is to use Look Ahead to seek out the remaining streets as quickly as possible. We shall see in Chapter 5 that this technique can work, but only some of the time.
Chapter 2 -- Computer Implementation

The real power of the program can be seen in the application itself. Users are given many choices regarding plow starting points, number of streets to plow before stopping, and type of algorithm to use. Not only is plowing a map easy, but creating an original map is also simple. A user can choose to create a new map or edit one he or she made earlier. This section will deal with the specifics of how to use the editing and the plowing modes.

Using the Main Menu

The main menu serves as the navigation for the entire program. The buttons launch the specific features of the plowing program.

- **New File** – This button will launch a new editing window in which the user can create a map.

- **Open file** – This button will launch a previously made file in the plowing program. The user will be prompted on the C++ input box to enter a filename. The request will be made until the user enters a correct name or decides not to proceed.

- **Edit a Map** – This button will launch a previously created file in the editing program. The user will be prompted on the C++ input box to enter a filename.
The request will be made until the user enters a correct name or decides not to proceed.

- **Start Plowing** – This button will launch the last saved file in the plowing program. Should the user not save, the button will yield a “File Not Found” response.

Should the user mistakenly select to Open or Edit a file, he or she must enter an incorrect name at the prompt. The program will then ask the user if he or she wishes to continue. The user should select “n” and go back to the Main Menu.

**Using the Plower**

The main menu activates the Plower by selecting a file with the “Open File” button or with the “Start Plowing” button on the main menu. The initial screen the user sees will be similar to the one shown here. The user can control the Plower through the button interface at the right of the window. Streets with more lanes are wider while streets with fewer lanes are narrower.
• **Number Pad** – The numbers to the right of the main buttons. The user first presses a number combination (greater than zero) followed by the button of the attribute he or she wishes the to apply number to.

• **Startingwith** – Indicates the intersection the plow will start or switch to. The value must be a number of an intersection on the map.

• **BasicPriority** – When the Plower has legal plowing parameters, the button will make the Plower use Basic Priority on following iterations.

• **Look Ahead** – When the Plower has legal plowing parameters, the button will make the Plower use Look Ahead on following iterations.

• **Number to Look Ahead** – This button is used to define the number of streets the plow will look beyond the local streets when using the Look Ahead Algorithm. The number to look ahead must be less than 20.
• **Number Of Streets** – This button is used to define the number of streets the current plowing algorithm selection will plow or travel with the plowing number less than or equal to 10,000.

• **End** – This button clears the current plowing session and resets everything for a new plow.

**Plowing Visuals**

When we begin to plow, streets will begin to change colors. Black streets have not been touched at all; blue streets have been seen by the plow but have more than one lane left; red streets have one lane remaining; green streets have no lanes remaining. As the streets are plowed, the width of each street reduces and the color of each street changes with each iteration. The graph is completed when all streets are green.

The current intersection of the plow flashes to make the plow easier to see and follow as it plows. Also, every time a street is traveled or plowed, the Plower displays the most current decision made regarding a street being plowed or traveled.
Using the Mapper

The Mapper is launched when the “Edit A Map” button is pressed on the main menu or when “New File” is pressed. The program is designed to let the user make maps simply and easily. Its interface resembles the Plower’s, with buttons along the right side of the window. Each button offers a unique functionality that allows the user to make maps the way he or she envisions them. The white area is called the stage and is where the users create their maps.

The Mapper Buttons

These buttons control functions that create, edit, and delete both streets and intersections from existing or new maps.

- **Create Intersections** – When this button is pressed, Create Intersection mode launches. Any mouse click on the white stage that is not on an intersection will generate a new numbered intersection on the screen.
- **Create Streets** – When this button is pressed, Create Streets mode launches. Any mouse click must be directed at an intersection. The first two intersections that are selected are locked in as a street. After the pair selection, the user must then go to the input screen where he or she is prompted to enter the street’s name, priority, and number of lanes.

- **Edit Intersections** – When this button is pressed, Edit Intersection mode launches. Any mouse click must be directed at an intersection and once an intersection has been selected, the next mouse click will reposition the intersection and update the streets if the click is on the stage. The intersection will stay selected until a legal click is made, another intersection is selected, or another mode is launched. If another intersection is clicked, it becomes the selected intersection. Changing modes erases the selection.

- **Edit Streets** – When this button is pressed, Edit Streets mode launches. Any mouse click must be directed at a specific street. Error tolerance on street selections varies, so the user is encouraged to click as closely to the desired street as possible. Once a street has been selected, the user must click Edit Streets again.
to edit a street’s properties on the C++ input screen. The user is prompted to change the street’s name, priority, and number of lanes, or to do nothing.

- **Save Map** – When this button is pressed, Save mode launches. The user is redirected to the input window to enter the desired filename (without spaces) for the map. All maps are saved in the Maps folder that is local to the project folder.

- **End Session** – When this button is pressed, the current map-editing window will close, deleting all unsaved data. The user should go to the Main Menu to make the next selection.

- **Delete Selection** – The user can delete both intersections and streets by first selecting the object the user wishes to delete with that objects editing mode; e.g., to delete a street, select the street with Edit Streets and then hit the Delete Selection button. Warning: there is no undo button, so the user should delete at his or her own risk!
- **Checkboxes** – There are four checkboxes to choose from. The top two boxes control the Snap-To-Grid feature. The left box displays the grid when checked and is broken into $\frac{1}{2}$-centimeter boxes. The right check box activates the Snap-To-Grid option and any points created or moved will snap to the closest corner on the grid when the checkbox is selected. The bottom two checkboxes control the display of names and parameters on the graph. The bottom left box controls the display of number of lanes and priority, and the bottom right box controls the display of the street names.

**Editing Visuals**

The Editing window has a couple of features that differentiate it from the Plowing window. Streets have three labels that tell the street’s name, priority, and number of lanes. As we can see from Figure 8, priority is labeled in light gray (green on screen), lane number in dark gray (blue on screen), and intersection numbers are black.

![Figure 8 – Example of Map-Editing Visuals](image-url)
Chapter 3 — Structure

This chapter describes, in greater depth, the pieces that make up each class within the program. These classes are Street, Intersection, Driver, Mapper, Snowplow, and Menu.

Structure Composition

Maps are centered around two main data structures, streets and intersections that represent the streets and intersections of the specific road system we are optimizing. Streets inherently hold the main focus of the structure because they make up the path we are trying to find. However, intersections know what streets they connect through an array of six street names and serve as a “resting place” for the plow to make its next decision. Because intersections connect specific algorithmic decisions, they serve as the true base structure for my program.

Streets are constructed of the following:

1. Two integer indices into an array of intersections. Since Streets cannot exist without intersections, this is a reasonable constructor. Intersections must be stored in an array or vector, and there must only be one within the program.

2. The name of the street, stored as a “string.”

3. Two integer costs associated with the street:
   a. Priority on a scale of 1-3 with 1 being most important.
b. The number of lanes on the street. Since I assume that a plow will be used that plows a lane at a time, the number of lanes would equal the number of passes a plow must make to completely plow the street.

**Intersections** are constructed of the following:

1. An x and y coordinate so the point can be easily referenced and both coordinates are floats.
2. An integer representing the number of streets that the intersection connects. I assume that an intersection will connect no more than 6 streets.
3. The constructor contains the names of the streets, stored in an array of “strings.” All indices in the array that do not have a street are set to “NULL.”

**Snowplows** are constructed of the following:

1. A string containing the filename of the Driver instance using the snowplow.
2. A pointer to the SimpleWindow that Driver has created.
3. A pointer to a vector of Streets and a pointer to a vector of Intersections. Pointers are used to allow one group of information to exist rather than copying the data into the instance of Snowplow. Vectors allow for dynamic storage and expansion of the data.
Drivers are constructed of the following:

1. A “string” that contains the filename that the instance of Driver is plowing.
2. Integers containing the number of streets and intersections on the map.
3. The “string” that contains the title for a Driver window.
4. The dimensions for the window as floats.
5. Position of the window relative to the top left corner of the screen.

Mappers are constructed of the following:

1. The “string” that contains the filename that the instance of Driver is plowing.
2. The integers containing the number of streets and intersections of the map.
3. The “string” that contains the title for a Mapper window.
4. The dimensions for the window as floats.
5. The position of the window relative to the top left corner of the screen.

Menus are constructed of the following:

1. The “string” that contains the title for a Menu window.
2. The dimensions for the window as floats.
3. The position of the window relative to the top left corner of the screen.
Chapter 4 — Class Functions

This chapter gives a brief description of each function of each class. The name of the file is given as a header and is followed by the constructor, facilitators, and mutators of each file.

Street

Constructor
Street (int one = 0, int two = 0, string name = "NULL", int priority = 2, int lanes = 2);

- Street parameters are defaulted to zeros, null strings, a priority of 2, and 2 lanes

Facilitator

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int GetIntersectionOne()</td>
<td>Returns an index into the array of intersections of a specific street intersection.</td>
</tr>
<tr>
<td>int GetIntersectionTwo()</td>
<td></td>
</tr>
<tr>
<td>String GetStreetName();</td>
<td>Returns the name of the street.</td>
</tr>
<tr>
<td>int GetPriority()</td>
<td>A user can also get the priority, number of lanes, and assigned handle of the street.</td>
</tr>
<tr>
<td>int GetNumberOfLanes()</td>
<td></td>
</tr>
<tr>
<td>int GetHandle()</td>
<td></td>
</tr>
</tbody>
</table>

Mutators

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void SetPriority(int priority)</td>
<td>Since Streets have default parameters and must be made on the fly, these functions set the parameters once the user enters them.</td>
</tr>
<tr>
<td>void SetName(string name)</td>
<td></td>
</tr>
<tr>
<td>void SetNumberOfLanes(int lanes)</td>
<td></td>
</tr>
<tr>
<td>void SetHandle(int handle)</td>
<td></td>
</tr>
<tr>
<td>void IncPriority()</td>
<td>Used by snowplow.h when plowing streets using Look Ahead. IncPriority increments the road priority by 4 every time the road traversed.</td>
</tr>
<tr>
<td>void RestorePriority()</td>
<td></td>
</tr>
<tr>
<td>void SetIntOne(Intersection Int)</td>
<td>Since Streets have default parameters, these functions set the intersections of the specific street when they are ready. Used specifically by Update streets in Mapper.h to update modified streets in each intersection</td>
</tr>
<tr>
<td>void SetIntTwo(Intersection Int)</td>
<td></td>
</tr>
</tbody>
</table>
Intersection

Constructor
Intersection(double x = 0, double y = 0, int Streets = 0, string Names[6]=NULL);

- Intersection also has a default constructor with everything initialized to 0 and NULL. Note the Names array stores the names of the specific intersection’s connected streets.

Facilitators

float GetXCoordinate()  // Returns the x or the y coordinate of a specific intersection.
float GetYCoordinate()

Position GetPosition()  // Returns the coordinates as a position.

int GetNumOfStreets()  // Returns the number of streets an intersection connects.

int GetHandle()  // Returns the handle the intersection was given upon creation.

string GetStreet(int index)  // Returns the street name from the intersection’s StreetNames array (names of streets it attaches) at the given index.

string GetStreetNames()  // A user can get a single string that contains all the streets an intersection connects, separated by a space. This is used mainly for display purposes.

Mutators

void SetHandle(int handle)  // Sets the handle because a handle is not set until a street has been created.

void SetCoords(Position position)  // Used whenever the coordinates for the intersection change. Used specifically when a user edits the position of an intersection.

void AddStreetToInt(string Name)  // Allows street names to be added to the existing names, removed from the existing names, or changed. A name would need to be changed if the user edits a Street’s name.

void RemoveStreet(string Name)

void ChangeStreet(string Old, string New)  // An intersection also keeps track of how many streets are connected to it, for various reasons. These functions change the number of streets the intersections think they have.

void IncrementStreets()

void DecrementStreets()
Plowing the Streets of Helena

Snowplow

Constructor
Snowplow(string name = "Charlie", SimpleWindow *Window = &SimpleWindow(), vector<Street> *St = NULL, vector<Intersection> *In = NULL);

- The snowplow’s name is defaulted to Charlie, Window to a basic simplewindow, and the pointers to the vectors are null.

Facilitators

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SimpleWindow *GetWindow()</td>
<td>Returns a pointer to the SimpleWindow of the instance. It’s always called MyWindow.</td>
</tr>
<tr>
<td>void MoveForward()</td>
<td>These are various functions that control a linked list. We can move the current pointer forward and backward; we can display all the streets in the linked list; we can move the pointer to the first instance of the string “streetname”; or, just move the pointer to the end of the list.</td>
</tr>
<tr>
<td>void MoveBackward()</td>
<td></td>
</tr>
<tr>
<td>void ShowStreets()</td>
<td></td>
</tr>
<tr>
<td>void CurrentStreetTo(string streetname)</td>
<td></td>
</tr>
<tr>
<td>void ToEnd()</td>
<td></td>
</tr>
<tr>
<td>Street GetStreet(int index = 0)</td>
<td>Returns the street at the given index or with the given street name from the program’s street vector. We can also get an intersection at a given index in the programs intersection vector.</td>
</tr>
<tr>
<td>Street GetStreet(string name)</td>
<td></td>
</tr>
<tr>
<td>Intersection GetIntersection(int index = 0)</td>
<td></td>
</tr>
<tr>
<td>Position GetIntOnePos(Street street)</td>
<td></td>
</tr>
<tr>
<td>Position GetIntTwoPos(Street street)</td>
<td></td>
</tr>
<tr>
<td>int FindFrom(string origin, string current)</td>
<td>Find From is used to find the intersection the plow originated at given a street previously traveled (origin) and street currently traveled (current). Find To uses the name of the last street plowed and the intersection started with (origin) to determine what intersection the plow is currently at. Both functions initialize the names and priority arrays in Driver. First Choices assumes no previous street and is used to start the plowing algorithm while Find Choices finds all the choices for the current intersection, considering the previous street plowed. Displays a message in the Driver window.</td>
</tr>
<tr>
<td>int FindTo(int origin, string current)</td>
<td></td>
</tr>
<tr>
<td>void FindChoices(int intersection, string streetname = &quot;NULL&quot;)</td>
<td></td>
</tr>
<tr>
<td>void FirstChoices(int startingwith = 1)</td>
<td></td>
</tr>
<tr>
<td>void Message(string message)</td>
<td></td>
</tr>
</tbody>
</table>
### Mutators

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void AppendPath(string streetname)</code></td>
<td>Appends a new path to the linked list with a given street.</td>
</tr>
<tr>
<td><code>void DeletePath()</code></td>
<td>These functions allow various linked list functionality to delete single and groups of paths as well as replace paths with different paths. May be useful for future revisions but not heavily used in this version.</td>
</tr>
<tr>
<td><code>void ReplacePath(string replacement)</code></td>
<td>StreetPlowed decrements the number of lanes a street has left in the Plowed vector. The Plowed vector shows the number of lanes for each street on the graph. GetPlowed returns the integer in Plowed at the index given. GetLastPlowed returns the private data member LastPlowed that tells the index of the intersection the plow was at last. The status is False when streets are left unplowed and True when the graph has been completed. NumberPlowed() returns the number of streets completed. Iterations() returns the number of times any algorithm has plowed or traveled a street in the current session. TotalLanes() returns the total number of lanes that are to be plowed.</td>
</tr>
<tr>
<td><code>void ReplacePath(string existing, string replacement)</code></td>
<td>The Greedy Algorithm that is based solely on priority and randomly selects streets for tiebreakers. Uses FindChoices to update name and priority arrays.</td>
</tr>
<tr>
<td><code>void DeleteFrom(string street)</code></td>
<td>Uses the Look-Ahead technique to populate the names and priority arrays.</td>
</tr>
<tr>
<td><code>void StreetPlowed(string plowed)</code></td>
<td>The intelligent algorithm that looks a user-defined number of streets into the future to make a selection.</td>
</tr>
<tr>
<td><code>int GetPlowed(int index)</code></td>
<td></td>
</tr>
<tr>
<td><code>int GetLastPlowed()</code></td>
<td></td>
</tr>
<tr>
<td><code>bool GetStatus()</code></td>
<td></td>
</tr>
<tr>
<td><code>int NumberPlowed()</code></td>
<td></td>
</tr>
<tr>
<td><code>int Iterations()</code></td>
<td></td>
</tr>
<tr>
<td><code>int TotalLanes()</code></td>
<td></td>
</tr>
</tbody>
</table>
Driver

Constructor
Driver(string filename, int streets=0, int intersections = 0, const string &Title = "Plow My Streets", const float WindLength = 27.0, const float WinHeight = 19.0, const Position &WinPosition = Position(0.5, 0.5));
- Driver’s main goal is to create the window for the plowing. The constructor requires a filename because a plow cannot function without a map.

Facilitators

SimpleWindow *GetWindow()
BitMap& GetBmp(const int index)
void FindStreetIndex(string streetname = "")
Street GetStreet(int index=0)
Street GetStreet(string name)
Intersection GetIntersection(int index=0)

int GetReturnArray(int index=0)
int GetPlowed(int index=0)

int MouseClick(const Position &mouse)
void Message(string message)

Mutators

void CreateMap()
void CreateIntersections()
void CreateStreets();
void CreateCosts()
void CreateBitMaps()
void NameStreets()

GetWindow() returns a pointer to the window of the instance which inside the class is called MyWindow. GetBmp returns the Bitmap at the given index in the Bmp vector.
Find the two indexes for each of the intersections associated with the given streetname. Specifically, used when creating streets from a file in driver.h.
Both GetStreets return a street given an index or a name. Index is most common, as the index is usually known. GetIntersection can only use an index because the Intersections are not named.
GetReturnArray returns the int at an index in the ReturnArray array. This array stores the indexes of the intersections for each street that is created in CreateStreets. FindStreetIndex populates ReturnArray. GetPlowed just returns the requested index of the Plowed array.
The main function that controls everything the user does. This function interprets the user mouse clicks and responds accordingly. MouseClick calls most functions.
Writes a message or instruction for the user at the bottom of the screen.

These functions draw the map, create the intersection and street vector, add the costs to the existing streets, create and load the vector of bitmaps which are used for buttons, and create the vector of street names.
The driver looks at three text files to create all the vectors. ParseStreets takes a line at a time from the streets file as the names of the streets on the map. ParseIntersection takes lines from the intersection file and turns them into Intersections. Last, ParseCosts deciphers lines from the cost file. Used in CreateStreets, CreateIntersections, and CreateCosts.

Assesses and returns the number of streets that have been plowed.

Controls the blinking cursor on the Driver main window using SimpleWindow’s TimerCallback function.
Mapper

Constructor
Mapper(string filename = "NULL", int streets=0, int intersections=0, const string &Title = "Make Some Maps", const float WindLength = 28.0, const float WinHeight = 20.0, const Position &WinPosition = Position(0.5, 0.5));

- Mapper's main goal is to create the window for the editing. Unlike Driver, the filename can be NULL as the user could create a new map. Everything else is comparable to Driver.

Facilitators

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int MouseClick (const Position &amp;mouse)</td>
<td>The main function of Mapper. Takes a user mouse click as an input, interprets the intent of the mouse click, and acts accordingly.</td>
</tr>
<tr>
<td>SimpleWindow *GetWindow()</td>
<td>Returns the window of the instance. Again, the name is MyWindow.</td>
</tr>
<tr>
<td>BitMap&amp; GetBmp(int index)</td>
<td>GetBmp returns the specified bitmap from the Bmp vector. GetIntPic returns the instance of the bitmap representing an intersection. It is used to help determine if a mouse click occurred in the IntPic.</td>
</tr>
<tr>
<td>BitMap&amp; GetIntPic(int index)</td>
<td>These functions are used to determine where the mouse click occurred. IsInsideInt tells if the click was in an IntPic and returns the index of the pic or -1 if no IntPic was clicked. IsInsideSt performs calculations to see if a click was directly between two points and returns the index of that street. It returns -1 otherwise. InsideRect is used by IsInsideSt to broadly categorize two points. It calculates the rectangle generated by two points and determines if the click is in that area.</td>
</tr>
<tr>
<td>int IsInsideInt(const Position &amp;mouse)</td>
<td>Returns the number of Streets, Intersections, and finds the index in the street vector that has the name “Name”.</td>
</tr>
<tr>
<td>int IsInsideSt(const Position &amp;mouse)</td>
<td>A check to make sure the name that was just entered by the user has not already been used.</td>
</tr>
<tr>
<td>bool InsideRect(Position one, Position two, Position point)</td>
<td></td>
</tr>
<tr>
<td>int GetNumStreets()</td>
<td></td>
</tr>
<tr>
<td>int GetNumInts()</td>
<td></td>
</tr>
<tr>
<td>int GetNameHandle(string Name)</td>
<td></td>
</tr>
<tr>
<td>string GetFileName()</td>
<td></td>
</tr>
<tr>
<td>Bool NameNotUsed(string name)</td>
<td></td>
</tr>
</tbody>
</table>
Menu

Constructor
Menu(const string &Title = "Main Menu",
const float WindLength = 10.0,
const float WinHeight = 12.0,
const Position &WinPosition = Position(7.5, 7.5));

0 The job of menu.h is to create a Main Menu for the user to make selections throughout his or her session.

Facilitators

int MouseClick(const Position &mouse);
SimpleWindow *GetWindow();

BitMap& GetBmp(const int index);
void CreateBitMaps();

bool GetMapStatus();
string GetFileName();
string GetFileDestination();

bool CheckFile(string name);

MouseClicked interprets the click of a user and launches the specific mode the user selected. GetWindow returns the window of the instance called MyWindow. CreateBitMaps loads the button images into the Bmp vector. GetBmp returns the bitmap at the index of the Bmp vector.

GetMapStatus determines if a map was saved in this session and permits the loading of the most recently saved map. GetFileName retrieves that file and GetFileDestinations retrieves the folder path for the files.

CheckFile scans the destination file and makes sure the file exists before the program attempts to open it.
Chapter 5 -- Analysis of Algorithmic Results

This chapter will discuss the performance of both Basic Priority and Look Ahead algorithms, specifically concerning the number of lanes an algorithm plows versus the lanes both plowed and traveled.

Measuring Performance

Routes will be measured in terms of efficiency. An optimal Snowplow route (Eulerian Path) is found when we plow all lanes and travel none. Any streets we travel hurt our efficiency because we traveled a street when we should have or could have been plowing. The formula for efficiency is defined as follows:

\[
Efficiency = \frac{\text{Total Lanes Plowed}}{\text{Total Lanes Plowed} + \text{Total Lanes Traveled}}
\]

Another aspect of algorithm performance is speed of computation. Simplicity lends its hand to the speed of Basic Priority. In the worst case, Basic Priority will perform 18 compares per street plowed. Recall that intersections can connect as many as six streets together and that the names of these streets are stored in an array. If the first five streets have been fully plowed and the last street is a priority 3, Basic Priority will see if there are any priority 1’s, then 2’s, and finally 3’s, reaching the final street on the last try.

Look Ahead performs a great deal more compares than Basic Priority. Essentially, Look Ahead performs Basic Priority along each leg of its look into the future because it picks the best choice at each leg. The only difference is that it picks the smallest priority overall, meaning all choices at each intersection are considered. For
example, if we performed Look Ahead 3 (three beyond the locals) and every intersection had six streets, we would do a large number of steps. For each local street, we'll scan 18 streets, making the worst possible number of compares 6*18 or 108 compares.

Remember that 108 compares is just for one decision in the worst case.

The complexity of Look Ahead increases linearly for the number of streets looked ahead and the number of streets at each stage in the look ahead. To elaborate, assume the worst-case scenario happens every time. The algorithm must then perform six searches, one for each local choice. In other words, the algorithm performs the same calculations six times. For each local street, Look Ahead will look three beyond. This means that six streets will be considered, three times. Our final number of compares is therefore $3 \times 6 \times 6 = 108$. We can generalize the formula in the worst case:

$$NumberOfCompares = 6 \times LookAhead \times NumOfLocals$$

This formula means that, the number of compares depends on the number of streets the algorithm looks ahead and the number of streets at each look. We can generalize the formula for all cases:

$$NumberOfCompares = \sum_{i=0}^{NumOfLocals} \sum_{LookAhead} NumOfStreets$$

The number of compares is the sum of the cumulative number of decisions each local street uses. For each local street, one compare is required plus however many streets are at each decision beyond.
Usage of Algorithms

Both Basic Priority and Look Ahead perform better under certain conditions: priority dispersion, map layout, and priority status.

- **Priority dispersion** pertains to the sequential nature of priorities on the map. For instance, maps generally have a string of streets that are all of high priority called a snow route. These streets are not scattered about the map but rather sequential and able to be reached by one another.

- **Map layout** refers to the positioning of the streets. Algorithms behave differently if the map is the map in grid form such as the central part of Helena versus jumbled such as downtown Helena. Grid maps tend to have a sequential snow route pattern while jumbled maps are quite disorganized.

- **Priority status** refers to the current plowed status of the map. Algorithms will behave differently if unplowed streets are more difficult to find.

Basic Priority performs quite well in terms of priority dispersion. If street are strung together, Basic will follow that route. This situation is especially useful when high priority street must be cleared on time. Basic Priority will assure that only high priority streets are considered and plowed first. Look Ahead is not as rigid as Basic is in sticking to designated snow routes. Look Ahead generalizes paths too much to stick to the snow routes and thus plows snow routes only when it sees fit.

Map layout is best suited to Basic Priority when normal layouts apply. Normal refers to a grid form of dispersion where an intersection has four streets and there are main streets that hold high priority. When streets become jumbled and designated snow
routes lose continuity, Basic will choose the best choice in a possible disjoint pattern. Since Look Ahead doesn’t care about structure, it performs very well with maps that have no set pattern.

Priority status gives the algorithms the most trouble. Basic priority picks a random street at an intersection if there are no unplowed streets left at that intersection so it may be quite some time before it finds a new set of unplowed streets. This phenomenon makes nearly complete maps quite hazardous to Basic. Look Ahead can only look so far. If the algorithm did not see any streets to plow, it will look a little farther. If it still doesn’t find a street, it will go on the current standing of the map.

Recall that street priorities are incremented upon plowed or traveling a lane. Completed streets thus have different priorities that break ties between plowed streets. Should the algorithm see only plowed streets, it could take awhile to escape the mass of plowed streets.

The main problem with both Basic Priority and Look Ahead is their inability to close out a plowing session. In analysis of a log session for both algorithms, streets are not traveled excessively until late in the session. Basic Priority is very simple, yet it will only choose a plowed street if it has no other option. It therefore doesn’t break down until the plowed status of a map is quite cluttered. Look Ahead, surprisingly, gives similar results only later than Basic. Look Ahead leaves 3-5 streets scattered about the map on a regular basis and won’t find them for many iterations.
Analysis of the Maps

Helena is delineated into 5 plowing regions: Northside, Westside, Central, Eastside, and Downtown. I will use these maps to demonstrate and explore the various ways my algorithms can be used. The strategies I explore are as follows:

- All Basic – Use only Basic Priority to optimize
- Look Ahead – Use one of many Look Ahead values such as 5, 10, or 15
- Use Basic and Look Ahead – Use Basic to start and finish with Look Ahead
- Mixture of Look Ahead – Use different Look Ahead values during plowing, i.e., use Look Ahead 4 to start and then Look Ahead 15 to finish

Figure 9 shows a table of the various plowing strategies. The random nature of Basic leaves too much room for sloppy plowing, making the poor results of using just Basic expected. A great surprise is that of Look Ahead 5; of all techniques, Look Ahead 5 is consistently atop the list. It is counterintuitive to think that looking ahead a smaller number of streets would perform better than looking ahead 15 streets. Reasons for this phenomenon will be explained in a following section. The reader should note that mixtures of algorithms were not performed many times. It is feasible that a better order could be found, but these results are based on usual output. Actual maps of Helena are included as an appendix.
Plowing the Streets of Helena

Downtown | Eastside | Westside | Central | Northside
---|---|---|---|---
Lanes | | | |
All Basic | 1600 | 1648 | 1382 | 1029 | 178
LA 2 | 916 | 651 | 566 | 529 | 39
LA 5 | 547 | 567 | 582 | 626 | 38
LA 10 | 703 | 660 | 540 | 479 | 38
LA 15 | 617 | 650 | 749 | 554 | 38
Mixture Basic/LA | 600 | 589 | 833 | 650 | 51
Mixture LA | 720 | 560 | 537 | 540 | 39

Figure 9 -- Table of Plowing Results

When finding a snowplow path, one must consider time spent optimizing as well as results. Basic Priority was always finished with its thousands of iterations before a Look Ahead was done with its hundreds. This result is of course due to the simplicity of Basic. This property becomes more important as the size of the maps grows. Maps the size of Chicago would take a considerable amount of time to optimize. We could run Basic Priority maybe 5 times and Look Ahead once (each using the same amount of time), then compare the answers to determine the best.

User intent also drives the optimizing process. Realistically, Basic should be used first so the priority 1 streets are open for travel. Look Ahead may complete the plowing faster, but remember that travelers are theoretically using the streets as they are plowed. If this is the case, Basic Priority must be used and iterations must be compromised so major streets are open for travel. Look to the table to see that a mixture of Basic and Look Ahead performed well.
Algorithmic Flaws

Look Ahead has an interesting flaw. Recall that looping is allowed in the algorithm. As the algorithm looks ahead, the streets chosen can be streets already seen, as long as their priorities are good enough. Look Ahead values become very interesting because looking ahead a great number such as 15 gives the algorithm more of a chance to loop. This means that it might not be as effective as originally planned and future revisions of the algorithm could improve this minor flaw.

A far subtler flaw in Look Ahead is the non-recognition of plow turning. Figure 10 gives an example of the flaw. Assume the plow is at the top left corner, and we are using Look Ahead 3. The algorithm will look right, seeing two unplowed streets and two plowed streets. However, looking left also yields two plowed streets and two unplowed streets, equaling the same priority total as the right choice. Depending on which choice was considered first, the street to the left could be chosen because the streets look equal even though the correct choice was obviously the unplowed street.

The main hindrance of the given algorithms is their inability to close out a session. The solution is the creation of another algorithm. As a future work, a Shortest Path algorithm should be devised. This algorithm will clean up the remaining streets left by the other two algorithms by taking the shortest path between the current location of
the plow and the closest incomplete street. This algorithm should significantly reduce the
number of iterations for each session.
Chapter 6 – Future Works and Improvements

There are always improvements that can be made to any program. While the current state of the program achieves much, there are a few improvements to consider for a future revision.

- The program is coded in SimpleWindows, a graphics library created in the mid-nineties. It was chosen due to class exposure, the smaller need for high tech graphics, and the small learning curve. New graphics libraries such as OpenGL and DirectX offer updated functionality that could further boost both the aesthetic and functional aspects of the program. Work can be done to update display features in the thesis and extend the user-friendly nature of the program further.

- Snowplows are constructed in such a way that new algorithms can be made to control the plows. In the future, many different flavors of algorithms can be added to the program to offer more diverse and specialized solutions.

- Current algorithm performance can be improved. Both algorithms perform exceptionally well until a small number of unfinished streets remain. Another algorithm, such as a shortest path algorithm, can be devised to finish the job of the existing algorithms.

- Blizzard conditions such as severe icy roads and high amounts of falling snow during plowing were not considered. The program would be even more realistic if there were changing weather as the plow operates. This change would mainly involve changing the Plowed vector which tells if a street has been plowed or not.
• Street name conventions could also be extended further to include linking multiple intersections to form one street. For example, Montana Ave. in Helena connects many intersections, but is only called Montana. In this program, the street is called Montana1, 2, etc. Currently, every street has a unique name to distinguish the paths apart. Should linked list compilation be changed to handle based instead of name based, names could be similar.

• Mapper editing functionality could be extended further with more complex graphics libraries. Streets could be named on the window itself instead of the Microsoft C++ input window. Groups of intersections could be linked and joined at once in case there was a really long street.
Bibliography


Appendix – The Maps

This appendix gives the five graphs of Helena, representing the five sections of Helena snow routes. All graphs were made with my Mapper program. The graphs have no street names because the map becomes too cluttered. In addition, these maps were composed from other maps provided by the city of Helena that were not completely up to date. Streets may be missing.

Downtown
Central
Plowing the Streets of Helena

Westside
Eastside
Plowing the Streets of Helena

Northside