Algorithm Design & Analysis for Scheduling and Motion Planning of iRobot Roomba

Jade Cheng

December 2008

University of Hawaiʻi at Mānoa
Department of Information and Computer Sciences
2500 Campus Road
Honolulu, HI 96822

Algorithm Design & Analysis for Scheduling and Motion Planning of iRobot Roomba

Jade Cheng

Department of Information and Computer Sciences
University of Hawaiʻi at Mānoa
Honolulu, HI 96822

yucheng@hawaii.edu

December 02, 2008

Abstract

The paper is concerted with Roomba's novel feature, which optimizes the summation of priorities through a vertex-weighted, undirected mesh graph. The paper presents a brut-force algorithm to solve this problem. The paper provides a comprehensive problem formulation, algorithm design, and the complexity analysis for the algorithm. We further prove the correctness of the algorithm and the fact that it is unlikely for us to find a polynomial algorithm that solves this problem because the worst exists can be converted into to a well-known NP-complete—the longest path problem. The algorithm proposed in this paper runs at an exponential time.

1 Introduction

The paper is concerned with the optimization of the summation of priorities in a vertex-weighted, undirected mesh graph. This is potentially an implementation of a novel feature for Roomba. The engineers need to manage this issue in real-life, in order to prepare the next generation of Roomba to deliver a more desirable service.

Hereon we present a mathematical description of the challenge presented above as a problem formulation, algorithm design, and complexity analysis. We make the assumption that a Decision Researcher with a strong mathematical background is the target reader.

2 Problem Formulation

We investigate the input and output of this problem set. It is clear this problem falls into the category of optimization problem. In mathematics and computer science, an optimization problem is the problem of finding the best solution from all feasible solutions. In mathematics, in particular, the term optimization refers to the study of problems in which one seeks to minimize or maximize a real function by systematically choosing the values of real or integer variables from within an allowed set.[1]

An optimization problem can be represented in the following way:

Given: A function $f: A \to R$ from some set A to the real numbers.

Constraint: A set of inequalities $g_1, g_2, \dots g_n$: $C_i < g_i < C_i'$, where $i = 1,2, \dots n$, C_i and C_i' are two sets of real numbers

Sought: An element x_0 in A such that $f(x_0) \geq f(x)$ for all x in A (maximization).

In our problem set, the expression could be written as:

Given: A function $f(\sigma) = \sum_{k=1}^{r} p_{i_k j_k}$: $V = \{v_{i,j} \mid 1 \leq i \leq m \wedge 1 \leq j \leq n\}$ denotes the vertices of a $m \times n$ mesh graph, $\mathcal{U}_{m,n}$; $E = \{(v_{i,j}, v_{k,l}) \mid 1 \leq i \leq m \wedge i \leq j \leq n \wedge 1 \leq k \leq m \wedge 1 \leq l \leq n \wedge [(i = k \wedge |j - l| = 1) \vee (|i - k| = 1 \wedge j = l)]\}$ denotes the edges of $\mathcal{U}_{m,n}$; $p_{i,j} \in \mathbb{N}$ for each $v_{i,j} = V$ denotes the priorities of the vertices, where $\mathbb{N}$ denotes the set of natural numbers.

Constraint: One inequality $g = \sum_{k=1}^{r} c_{i_k, j_k} + d(v_{a,b}, v_{i_1, j_1}) + \sum_{k=1}^{r-1} d\left(v_{i_k, j_k}, v_{i_{(k+1)}, j_{(k+1)}}\right) + d(v_{i_r, j_r}, v_{a,b})$: $g \leq C$, where $1 \leq i_1 \leq m \wedge 1 \leq j_1 \leq n$; C as a constant denotes to battery capacity.

[1] http://en.wikipedia.org/wiki/Optimization_problem

Sought: A sequence σ of r *distinct* vertices $v_{i_1,j_1}, v_{i_2,j_2}, \cdots, v_{i_r,j_r}$ which maximizes the objective function f(σ), subject to the following constraint g, where $1 \leq i_1 \leq m \wedge 1 \leq j_1 \leq n$.

2.1 Preliminaries

The problem formulation described above is based on a series of preliminaries and simplifications. Our problem is mostly concerned to maximize the summation of priorities of vertices on a known mesh graph, which indicates a continuous map of a house/apartment's floor plan. Implementation of this program is feasible in theory. Some of the simplified constraints are, however, need to be further studied before any real-life application can be conducted.

We will discuss several major preliminaries of our problem formulation in the following sections. Of course, there are other conceivable simplifications that are not going to be discussed in this paper.

2.1.1 Roomba stores the complete map

In the algorithm presented in this paper, all the computations are conducted and finished off-line. It means that the best solution is a known when the robot starts its mission. The robot follows an already computed path to carry out the cleaning duty. Through out this path, the first portion is traveled in the robot's traveling mode, which consumes less battery life per distance. The second portion is traveled in the robot's cleaning mode, which consumes a combination of cleaning battery consumption and a traveling battery consumption. The third, also the last portion of the trip is traveled in the robot's traveling mode. The last section of traveling takes the robot to its home base where it starts the trip.

Since the calculation is conducted and finished off-line, it requires the robot to have an intact memory of the entire map for a particular house/apartment's floor plan. Conceivably, this computation can be done during the battery charging stage.

As we discussed in the previous paper of this project, path planning is a one of the major topics in the field of mobile robot navigation. Autonomous mobile robots are used in various applications such as in automatic freeway driving, cleaning of hallways, exploration of dangerous regions, etc. These applications demand robust and adaptable methods for path planning.

Path planning can be divided into two categories, one is global path planning, where there exists a priori knowledge of the complete working area; and the other is local path planning, where the working area is uncertain. Global path planning includes configuration space method, potential field method and generalized Voronoi diagram. The planning is done off-line, and the robot has complete knowledge of its working area and its path when it starts. Local path planning methods use ultrasonic sensors, laser range finders, and on-board vision system to "perceive" the environment; planning is done on-line[2].

As we can see, in this problem formulation, we do not consider the local path planning while just focus on the global path planning. We take the entire map of the working area as a part of the input.

[2] I.J. Nagrath, Laxmidhar Behera, K. Madhava Krishna, and K. Deepak Rajasekar, *Real-time navigation of a mobile robot using kohonen's topology conserving neural network*

Once the decision is made off-line, the robot does not have a flexibility of changing the plan when it is in the working mode.

In real life, however, we need to take into consideration of the human interference. No matter how well the engineered robot memory and navigation systems are there is no promise that the map stays the same as it is when it was mapped by the robot. For example, the furniture is mapped as barriers. Based on the preliminaries of this paper, to the robot perspective, the furniture behaves just as other barriers, such as the wall. The furniture however can move from place to place even during cleaning. Therefore, in real life, there is no absolute up-to-date off-line map. But, in our paper we only focus on global path planning and consider it as a simplification.

We use the following scenario as an example (Figure 1). The single block barrier object can be a chair. We use dashed line for traveling mode movements and solid line for cleaning mode movements. In the first map, the chair is at the position as shown. The computations are based on this map. The best solution path is calculated as shown. Then, the chair might be accidentally moved to a position that is on the solution path. Since the computations are done off-line, the robot would still follow the out-of-date solution path. Our problem formulation does not, however, provide a alternative option when the robot hits the unexpected barrier shown on the right side of the figure.

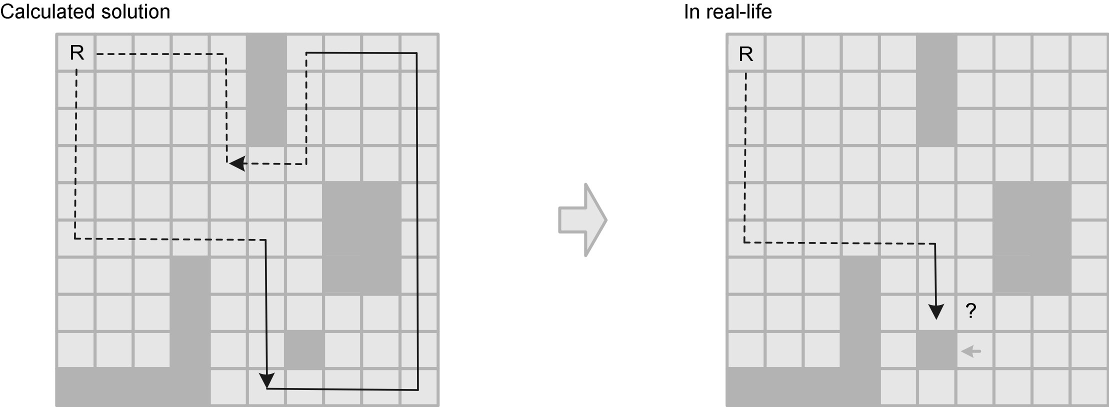


Figure 1. Simplification 1: Roomba stores the entire map, which should not change afterward.

2.1.2 Roomba does not switch between cleaning and traveling modes within one trip

This problem formulation is also based on the assumption that the robot is not capable of resuming cleaning mode within one trip. This means that the cleaning mode movements can be represented by one continuous stretch of a solid line. The solution provided by this problem formulation does not include the case when the robot needs to travel to a new area and resume cleaning in the new area.

During cleaning, the robot takes the vertices that are not cleaned. We assume once the vertex is cleaned the robot refused to go over it again in its cleaning mode. Therefore, if the robot finishes cleaning within a certain area, even if it still has more than enough battery life to move to a new area to clean more, based on our assumption, it would not take this action. In other words, once the robot starts one round of cleaning, it will continue until the end of its battery life is reached or the

vertices around its current position are all cleaned; once it ceases the cleaning mode, there is only one action that the robot takes—traveling home.

In real life, however, as we discussed in the first paper, the robot has a built in feature which allows the machine to stop cleaning when it finishes one area. The robot would travel to a new area and resume cleaning. This is directed by the lighthouse navigation system. The robot does not have the knowledge whether the entire map is covered or not. It requests the lighthouse to give the further instruction. Before the lighthouse signal, the robot has no idea where the entrance of next area is. It would have to travel to that entrance as it receives the lighthouse signal. During this travel, the robot uses its traveling mode. The robot may re travel a stretch of path that has already cleaned.

Let's use the following scenario as an example (Figure 2). In real-life, the robot would be informed about the other area on the map which is shown as the right side of the grid. The robot ceases the cleaning mode and travels to the new area where it turns on the cleaning mode and continues cleaning. The total coverage of the vertices' priorities is the summation of the two stretches of paths. This is, however, not within the consideration of our problem formulation. We assume that the robot computes the entire path off-line and manages to cover the most amount priorities by taking a single run powered in its cleaning mode. Once it stops cleaning, it goes back to the home base. The single solid path shown on the right figure is supposed to be the best solution based on our assumption. If we assume the two figures below show cases that consume battery lives within the input battery capacity, it is possible that the left figure provides a better summation of covered priorities comparing the single solid line on the right figure. But based on the preliminary of this problem formulation, we consider the right output as our best solution, while the left figure is an invalid output.

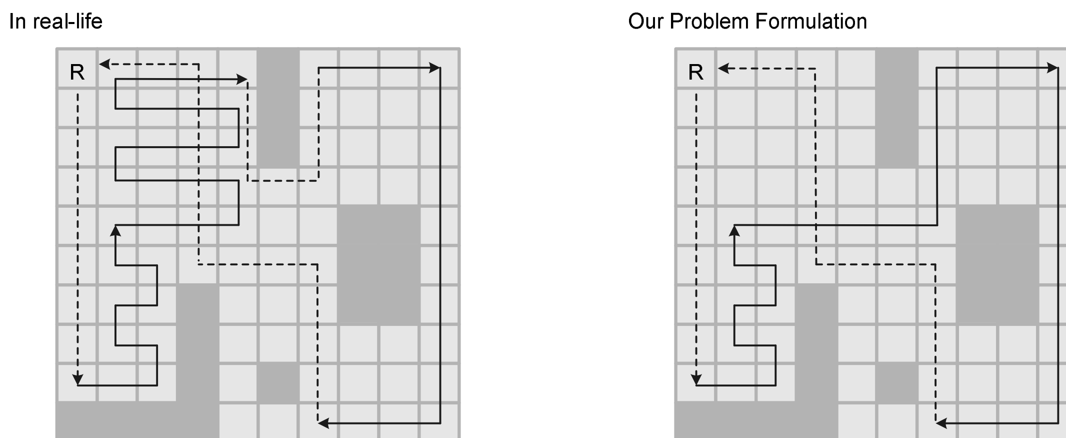


Figure 2. Simplification 2: Roomba does not cease and resume cleaning within one trip.

2.1.3 Roomba cleans with a floor coverage of one

There is another simplification of our problem formulation that needs to be pointed out. In this paper we assume a strict restriction of floor coverage of one. As we discussed in the previous paper of this project, the regular floor coverage of Roomba is three. It is not a fixed number for different surface conditions. The robot has a built-in mechanism to detect the dirtiness of a certain area. Generally, the assessment is based on the amount of dirt it picks up as it cleans. Once the robot

detects a dirtiness crossing a certain level, it increases the floor coverage of this area. This parameter is also user adjustable for some of the higher end machines.

However, on the other hand, we can also consider this factor as part of the input. It means that the input battery consumption for each vertex contains the multiplication factor of the floor coverage of this certain vertex. Therefore, for our problem formulation, we impulse a strict floor coverage of one for the input grid map along with its vertices' battery consumptions and priority values. Intuitively, we can use the following scenario as an example.

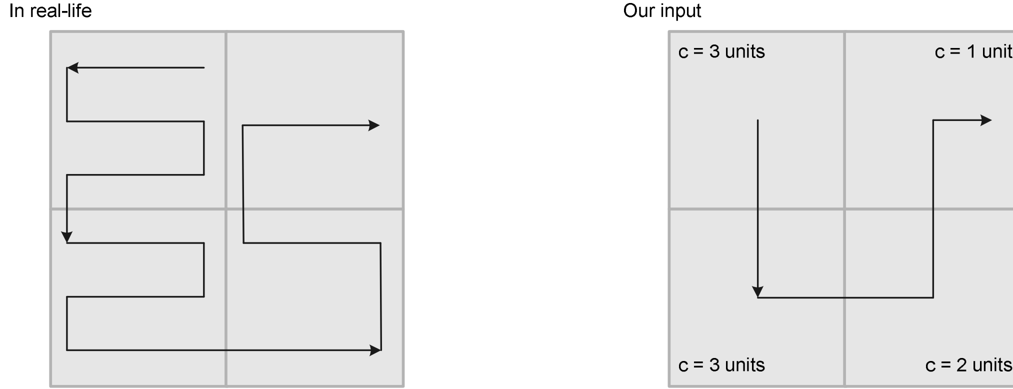


Figure 3. Simplification 3: Roomba cleans with a strict floor coverage—one.

2.2 Detailed Problem Formulation

In this section, we will talk into the algorithm formulation of the branch problem and the heuristic distance algorithm that we are going to use in the final implementation of this project.

2.2.1 Problem formation for the branch problem

Obviously, the cleaning starting point does not have to be where the root is. Once the starting point is chosen, our program should compute the best path that starts at that point. The selecting of the starting point is, therefore, one of the determinants of the output. Since the robot is capable of traveling to whichever starting point that is chosen, we can divide the problem into the number of possible starting points. In other words, we convert the formulated problem into the following.

Given: A function $f(\sigma) = \sum_{k=1}^{r} p_{i_k j_k}$: $V = \{v_{i,j} \mid 1 \le i \le m \wedge 1 \le j \le n\}$ denotes the vertices of a $m \times n$ mesh graph, $\mathcal{U}_{m,n}$; $E = \{(v_{i,j}, v_{k,l}) \mid 1 \le i \le m \wedge i \le j \le n \wedge 1 \le k \le m \wedge 1 \le l \le n \wedge [(i = k \wedge |j - l| = 1) \vee (|i - k| = 1 \wedge j = l)]\}$ denotes the edges of $\mathcal{U}_{m,n}$; $p_{i,j} \in \mathbb{N}$ for each $v_{i,j} = V$ denotes the priorities of the vertices, where $\mathbb{N}$ denotes the set of natural numbers.

Constraint: One inequality $g = \sum_{k=1}^{r} c_{i_k,j_k} + (v_{a,b}, v_{i_1,j_1}) + \sum_{k=1}^{r-1} d\left(v_{i_k,j_k}, v_{i_{(k+1)},j_{(k+1)}}\right) + d(v_{i_r,j_r}, v_{a,b})$: $g \le C$, where i_1, j_1 are fixed indexes; C as a constant denotes to battery capacity.

Sought: A sequence σ of r *distinct* vertices $v_{i_1,j_1}, v_{i_2,j_2}, \cdots, v_{i_r,j_r}$ which maximizes the objective function f(σ), subject to the following constraint g, where i_1, j_1 are fixed indexes.

Once the starting point is chosen, the traveling cost from the home base to this starting point can be calculated. We will discuss the algorithm of conducting this calculation in the following section. We subtract this part of battery life from the input battery capacity. We compute the feasibility of each step by comparing the expected battery consumption of taking this step to the battery life that is current available. If the resulting battery life is a positive number, this step is conceptually possible to be a part of the solution path. If this is the case, we shall continue on this path and take the next possible step. The end cases come when we enter a dead end by taking a certain step or the battery life can not sustain any more cleaning. When we encounter either of the base cases, we gather one possible solution path. We compare the priorities collected in this path with the priorities collected in other possible solution paths. The path with the highest priorities summation wins and shall be kept as the output of this particular branch problem. We further compare all of the best solutions of paths start from all possible starting points. The overall best solution is formed.

2.2.2 Problem formulation for the traveling mode

As we discussed in as our first preliminary, the computation concerted with this paper is conducted 100% off-line. The robot remembers the entire map as a part of the program's input. The local path planning and on-site navigation is not within the scope this paper. For a continuous map, there are some well-known distance heuristic algorithms. If we consider the map is a mesh graph with no obstacles, Manhattan distance is a good candidate. If we take into consideration of the obstacles on the mash graph, A* algorithm is an ideal candidate. In the implementation of A* algorithm, we can still use Manhattan distance to provide the distance heuristic values, which update themselves as the program runs. In the following sections, we will discuss these two algorithms in more details and take a look at their practical implementations for this project.

Manhattan distance

The definition of Manhattan distance is the distance, paths, lines, etc. which are always parallel to axes at right angles. For example, a path along the streets of Salt Lake City or the moves of a rook in chess[3]. In a plane with $p1$ at $(x1, y1)$ and $p2$ at $(x2, y2)$, it is $|x1 - x2| + |y1 - y2|$. Basically, it provides the shortest path from point to point on a grid graph with out allowing cutting corners under the preliminary that there is no obstacles block the way in the graph. Manhattan distance fits our needs based on this problem formulation. At a certain position, (i, j), the robot has four possible options for its next step, left $(i - 1, j)$, right $(i + 1, j)$, up $(i, j + 1)$ or down $(i, j - 1)$. It does not travel following the diagonals. For example from graph position (i, j) to $(i + 1, j + 1)$.

[3] Eugene F. Krause (1987). Taxicab Geometry. Dover. ISBN 0-486-25202-7

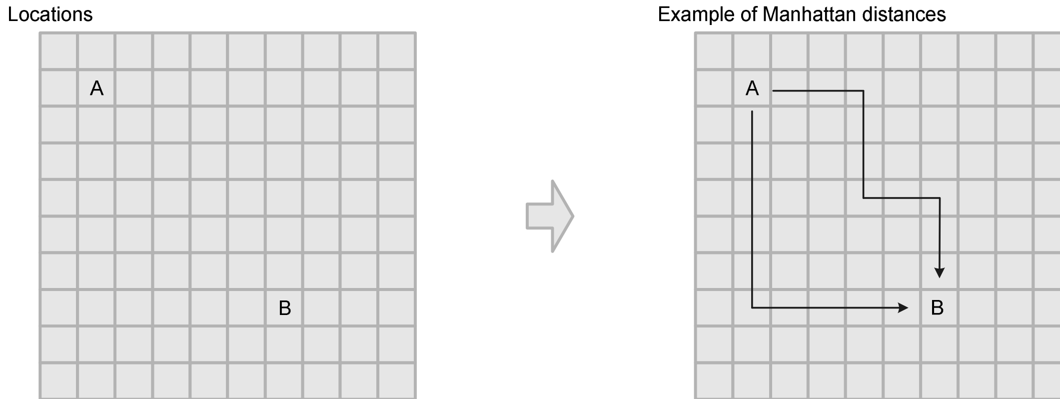


Figure 4. Two examples of Manhattan Distance

A* Algorithm

In computer science, A* is a best-first, graph search algorithm that finds the least-cost path from a given initial node to one goal node. It is a variant of Dijkstra's algorithm, which is more suitable for a continuous grid-based map[4]. The following functions denotes the values that the nodes store to apply A* algorithm. The order of visiting is determined by the value of function $f(x)$ for each node.

$g(x)$: the actual shortest distance traveled from initial node to current node

$h(x)$: the estimated (or "heuristic") distance from current node to goal

$f(x)$: *the* sum of $g(x)$ and $h(x)$

Let's put this implementation into a simplified example (Figure 5). We have the robot sitting at position A, and the destination is position B. The starting point could be the docking station and the destination could be the calculated cleaning starting point. The starting point could also be the calculated end point and the destination could be the docking station. As we can see, the trip is symmetric. The calculated starting point could also be taken as the calculated end point or vice versa. We have two blocks colored with solid gray representing the obstacles in between the robot and its desired destination. If A and B are located in two different rooms, we can think of the obstacles as the walls. If A and B are located in the same room, we can think about the obstacles as the furniture. The obstacle is, however, not allowed to change once the room is mapped as we discussed in the preliminary section.

In this example (Figure 5), the number shown as $f = x_1$ represents the current $f(x)$ for this grid, the number shown as $g = x_2$ represents the local $g(x)$ for this grid, and the number shown as $h = x_3$ represents the heuristic distance value of the grid to the destination grid B, $h(x)$. For this implementation, we used Manhattan distance to compute the estimate distance. As we discussed in the previous section, Manhattan distance fits our need. The value is the best possible distance from a particular node to the destination node. If there is any Manhattan distance path available, the robot would eventually take that path as the algorithm executes. If there is no Manhattan distance

[4] http://en.wikipedia.org/wiki/A*_search_algorithm

path available to get to the destination, the robot would adjust the value of the actual $g(x)$ for the nodes on the way, and try get to the destination taking the shortest way around the obstacles.

The figures used shading to represent the closed set, which includes the nodes that are already explored. The open set is shown as nodes without shading. They contain function values and parent pointers. The nodes in the open set are waiting to be examined. They are stored in a priority queue based on their objective function value, $f(x)$. They are polled out one after another and placed into the closed set. At the same time, we add their directly connected neighbors that are not in the open set into the open set, and update their $g(x)$ and $f(x)$ values. The final figure demonstrated the solution path with solid black arrows. The final solution is drawn based on the parent pointer from the destination node. We trace back from B. Every node on the way is a member of the solution path. We finish gathering the nodes of the solution when we reach the starting point A.

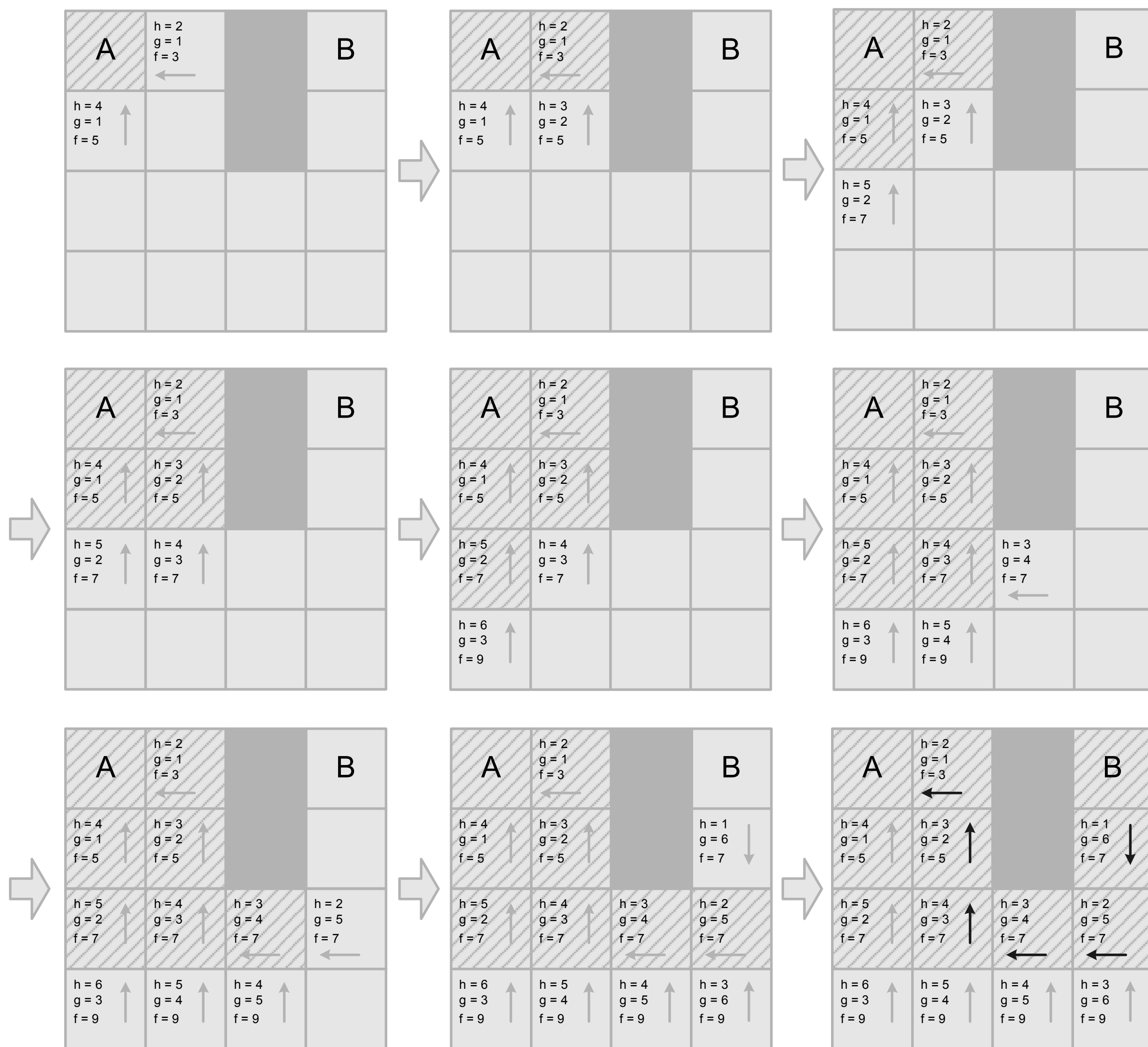


Figure 5. The execution of A* algorithm that takes Manhattan Distances as the heuristic distances

2.2.3 Conclusions of problem formation

So far, we discussed the main thoughts of the algorithm that we are going to use to compute the best solution for the branch problem and the algorithm for the traveling mode. Combining these two, we formed a solution algorithm to maximize our final objective function $f(\sigma) = \sum_{k=1}^{r} p_{i_k j_k}$. The program would use the A* algorithm to calculate the distance and the battery consumption, which is linear proportional to the distance, from the home base to the chosen starting point. Then it tries all four possible steps while computing the feasibility of each step. This is done by keep computing how much battery life it needs to travel home from a certain position. The branch best solutions are gathered through this computation. Then, the final solution is chosen from the branch best solutions.

In our final solution data collection, there are several components. It includes the summation of the priorities, which is supposed to be the maximum possible based on the input. It includes the battery consumption of this trip, which is smaller but should be somewhat close to the input battery capacity. Last but not the lest, it includes a sequence σ of r distinct vertices, $v_{i_1, j_1}, v_{i_2, j_2} \cdots, v_{i_r, j_r}$ which maximizes the objective function, $f(\sigma) = \sum_{k=1}^{r} p_{i_k j_k}$, subject to the constraint, inequality $\sum_{k=1}^{r} c_{i_k, j_k} + \left(v_{a,b}, v_{i_1, j_1}\right) + \sum_{k=1}^{r-1} d\left(v_{i_k, j_k}, v_{i_{(k+1)}, j_{(k+1)}}\right) + d\left(v_{i_r, j_r}, v_{a,b}\right) \leq C$. Optionally, it could also include the information of the path it travels from the home base to the starting point and the path from the end point back to the home base. These paths are not conserved as the required output. They are, however, helpful to build the final applet product as the user can see how the robot avoids the obstacles on the way while still manages to take the shortest possible path from place to place.

2.3 Property of Optimum Solutions

The algorithm we present in this paper provide an absolute solution for a routing maximizing problem. It is different from an approximation algorithm that provides desirable heuristic solutions, although the latter approach is more commonly conducted in real-life. We will later discuss in the algorithm complexity analysis section that the run-time of this algorithm is not polynomial. It grows very exponentially as the problem size grows. It is conceivable that there exist approximation algorithms to partially solve the problem with runs at a reasonable speed. The run-time is, however, not the first priority of this design. The ultimate goal for this algorithm design is the correctness of the final single solution.

As we can see, the algorithm is a brut-force algorithm that traverses all possible paths to find the solution path. Before actually examining the rest of the possible vertices, we do not have the knowledge of whether certain vertices are not worth visiting. The final output is generated after coving all possible steps of all vertices that is under examination. Therefore, if we can prove that the algorithm successfully traverse the entire graph within it battery life allowance and returns the longest possible path, we prove the correctness of the optimization property.

Proof of graph coverage for the brute-forth algorithm

Suppose we have more than enough battery capacity to travel as many vertices as we need in a mesh graph. This is the case, when we get to traverse the entire graph with no constraints.

Therefore, we need to prove the graph is completely traversed and the longest path is return as the output of our algorithm if this case occurs.

Suppose there exists one vertex v that is not traversed the whole time when the algorithm executes with a sufficient battery capacity input. The vertex can be one of the three vertices, a corner vertex that has two neighbors, an edge vertex that has three neighbors, or a middle vertex that has four neighbors. Based on the assumption, the vertex v is never examined. In other words all paths that cover it neighbors stop at where the neighbors are.

This implies one of the base cases is hit while examining the neighbor vertices. Since we also assumed that the battery capacity that we are considering here is more than enough, the only base case left is that the neighbors are facing dead ends. That is to say, there are no available further steps to take on those paths. All of the adjacent vertices are already covered on the paths or the boundaries are reached. That is how the paths all stop at the positions where the neighbors are.
We observe a conflict. As part of the assumption, vertex v is never examined. The fact is any of the paths that cover v's neighbors can not stop at where the neighbors are because a next step is always available as long as v is still un-explored.

Therefore, we proved that the graph is completely covered in the algorithm presented in this paper.

Simplified example of the graph traversal

Let's consider a mesh graph with six vertices (Figure 6). We can see that all vertices are traversed. All possible paths in the graph are examined. If we plug this graph into the algorithm with a sufficient battery capacity, all path solutions would be eventually compared and the solution with the greatest summations of priority would be chosen as the final output.

In this example, we use shading to indicate traversed vertices. We use dark color solid line arrow to represent the currently examining vertex. We use light color solid line arrow to represent a pointer point to the previous vertex on a particular path. The leaf graphs in the tree structure describe all possible paths that the graph can product with the top light most vertex as the starting point.

In this example, the longest paths are the ones that cover all vertices in the graph. It does not have to be the case. The longest path might hit the end case before covers all vertices. Even if the longest paths do not contain certain vertices, those vertices are covered by other paths as we proved above.

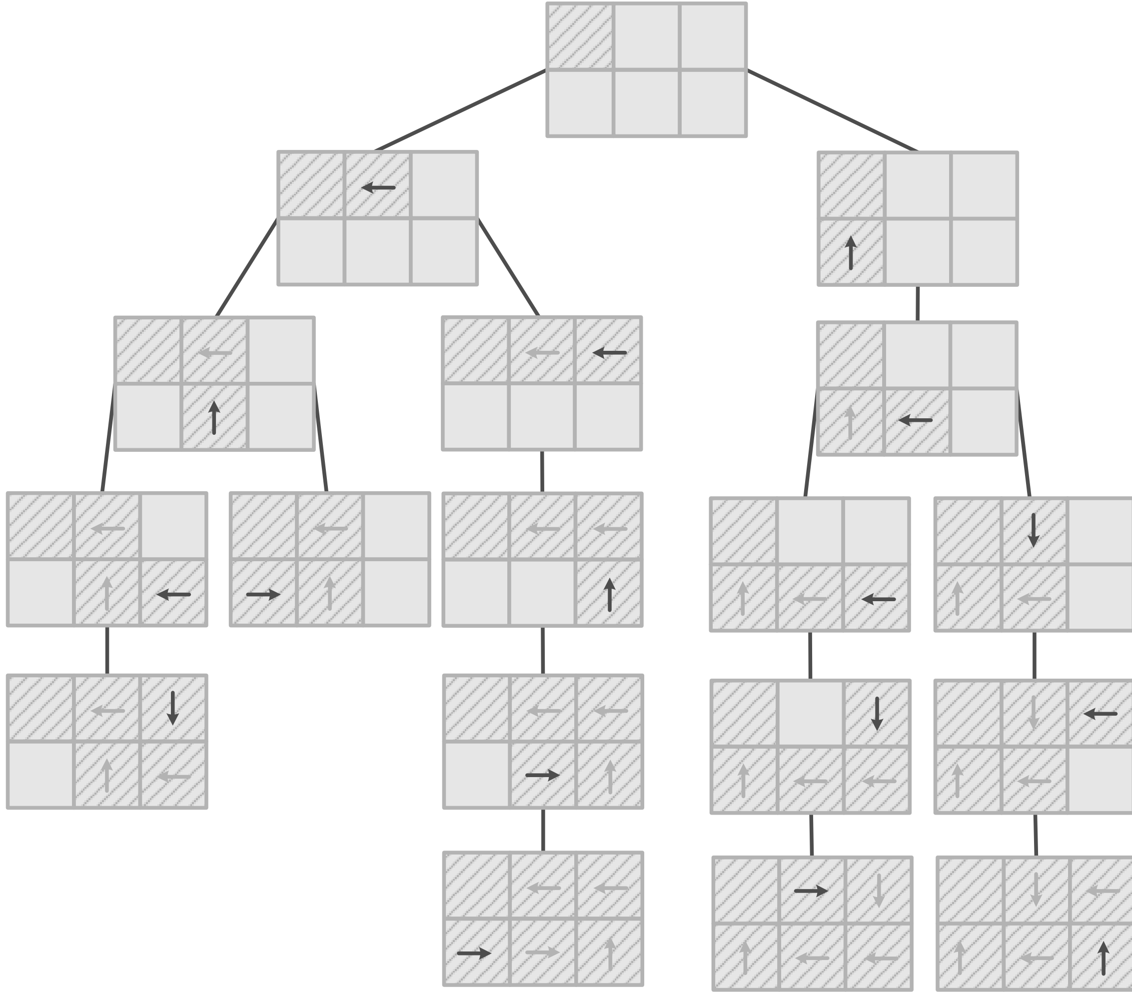


Figure 6. A tree structure demonstrates the graph traversal in this algorithm design.

3 Algorithm Design

The algorithm is designed based on the problem formulation described above. Generally speaking, in the outer loop, we go through all vertices other than the home base to examine the branch best solutions when each vertex is chosen as the starting point. Then, we take all possible steps to explore the path using recursion. The path terminates when it hit one of the tow based cases: when the front most vertices on the path is blocked. There are no further steps to take; when the battery life is exhausted and can not sustain cleaning more vertices.

The abstract data types created for this algorithm includes the Graph, the BooleaMatrix, the GraphSolution, and the Vertex node. Mostly, they are just collections of data that help us to present data more clear.

In the **Graph** data type, we store the information relevant to the input mesh graph. Namely, they are the width, height of the mesh graph, the home base vertex, the battery capacity, the battery consumption for each vertex in the mesh graph, the priority value for each vertex in the mesh graph, the positions of obstacles. This is an immutable class.

In the **BooleanMatrix** data type, we store the information indicating whether the vertices are covered or not by the particular path. It has a same size as the input mesh graph, Graph. It always starts with all false values. Once the vertex is covered, it is marked with true. By looking up this matrix, we can decide if a vertex is blocked as a dead end or not. This is a mutable class as we need to mark and unmark elements.

In the **GraphSolution** data type, we store information relevant to the solution. It contains a distinct sequence of vertices and the summation of priorities of this sequence of vertices. Of course, we can always compute the summation of priorities if we know about the sequence of the vertices. However, this summation value is used all the time through the algorithm execution, repeating the computation dramatically impair the performance. This is an immutable class.

In the **Vertex** data type, we simply store the information of the coordinate information of the vertex in the mesh graph. This is an immutable class.

In the pseudo code provided in this paper, we do not go into details of all of the data types. The accessors and mutators for each data type should be clean by referring to the paragraphs above.

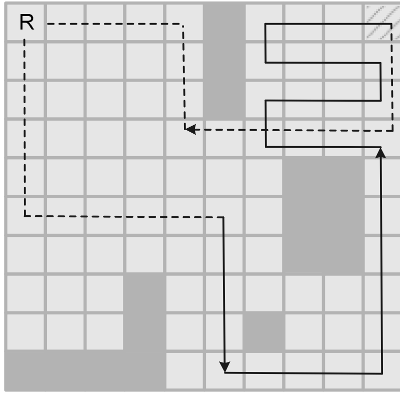
3.1 Key Idea

As we discusses, this algorithm is a brute-force algorithm that traverses every possible path under the constraint of battery allowance and compares the summation of priorities for all these paths to select the final optimization solution. The correctness of the graph traverse is proved in the property of optimization solution section. Here, we discuss more into detail of the end cases of this algorithm.

3.1.1 Base case 1: dead end

The first end case is that the current front most vertices on the paths are blocked (Figure 7). It could be surrounded by vertices that are already covered on the same path or the boundary of the map. As we discussed in the preliminary section that we impose a strict floor coverage of one, we don not allow the robot back track the vertices that are already cleaned. At the same time, the problem is also constructed based on the assumption that the robot does not have the ability of resuming the cleaning mode within one round. This means, when the dead end case is hit, there is only one option for the robot—returning the home base. The following graph shows two scenarios for this case. The shaded vertices represent the blocked vertices for each of the scenarios. Even if at the dead end the robot still has enough battery to conduct more cleaning work, it can not take that action.

Example 1



Example 2

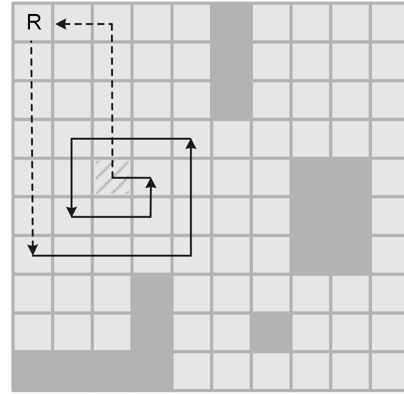


Figure 7. Two examples of end case 1: dead end (blocked end).

3.1.2 Base case 2: battery capacity can not sustain more cleaning

The second end case is when the battery life can not sustain any more cleaning. When we examine the current vertex, we calculate the battery consumption of cleaning this vertex plus the battery consumption of traveling back to the home base from this vertex. We compare this summation with the battery life that we have at that point. If we have enough, then this vertex is taken as part of the potential solution; if not, the robot needs to go back to the home base without cleaning this vertex.

3.2 Pseudo Code for the algorithm

Based on the key ideas of this algorithm design, we present here the pseudo code for this algorithm. As we decideded, the detailed information of the abstract data types is not included in this paper. The accessors and mutators method calls should be clear by referring to the previous section where each abstract data type is described.

We do not provide the pseudo code for A* algorithm implementation in this paper. It will be applied directly to the algorithm implementation of this project. According to the project requirement, we take the distance from point to point as a given. Here, we use name "distance" as the method name. It will be further created by implementing A* algorithm. The method takes the position of the home base and the position of a vertex in the mesh graph. In the pseudo code, we pass in the coordination of the vertex and the coordination of the home base as the method's parameters.

1 **Graph** *input*;
2 **BooleanMatrix** *matrix*;
3 **int** *maxGoodness*;
4 **GraphSolution** *output*;
5 **Stack<Vertex>** *sequence*;

JADE-MESH-OUTERLOOP(**Graph** *graphInput*)
6 *input* ← *graphInput*;

```
7     matrix ← initialize as the size of graphInput and populate with false;
8     maxGoodness ← the minimum integer;
9     output ← null;
10    sequence ← initialize as a new object;
11    for int i ← 0 to i ← graphInput.width; i++ {
12        for int j ← 0 to j ← graphInput.height; j++ {
13            JADE-MESH-RECURSION (i,j, 0, graphInput.capacity – capacityToBase(i,j));
14        }
15    }
16    return output;
```

JADE-MESH-RECURSION(**int** *x*, **int** *y*, **int** *goodness*, **int** *capacity*)

```
17    if isBlocked(x, y) = true or isBatteryExhausted(x, y, capacity) = true {
18        If output = null or goodness > maxGoodness {
19            mxGoodness ← goodness;
20            Output ← new GraphSolution(sequence, goodness);
21        }
22        return;
23    }
24    sequence.push(new Vertex(x, y));
25    matrix.mark(x, y);
26    int newGoodness ← goodness + input.priority(x, y);
27    int newCapacity ← capacity – input.comsumption(x, y) – 1;
28    JADE-MESH-RECURSION(x – 1, y, newGoodness, newCapacity);
29    JADE-MESH-RECURSION(x + 1, y, newGoodness, newCapacity);
30    JADE-MESH-RECURSION(x, y – 1, newGoodness, newCapacity);
31    JADE-MESH-RECURSION(x, y + 1, newGoodness, newCapacity);
32    sequence.pop();
33    matrix.unmark(x, y);
34    }
```

The helper methods used to conduct here are listed as below. As we decided, the heuristic distance method—A*algorithm is not presented. We use "distance" as its method name. It is used in the computation of battery life needed to travel from place to place.

CAPACITYTOBASE(**int** *x'*, **int** *y'*)

```
1     return distance(input.base.x , input.base.y, x', y');
```

ISBATTERYEXHAUSTED(**int** *x*, **int** *y*, **int** *capacity*)

```
2     if capacityToBase(x, y) + input.consumption(x, y) + 1> capacity {
3         return true;
4     }
5     return false;
```

ISBLOCKED(**int** *x*, **int** *y*)

```
6     if x < 0 or x <= matrix.width or y < 0 or y >= matrix.height {
7         return true;
8     }
9     if x = input.base.x and y = input.base.y {
10        return true;
11    }
12    return matrix.isMarked(x, y);
```

3.3 Conclusion of Algorithm Design

As a brief summery of the algorithm design, we use the following example to demonstrate the computation procedure. We have a mesh with four vertices in this example. We assume the top left vertex A is the home base vertex. Therefore, we have three options of the starting point shown as the second level of the tree structure. They are the other three vertices, B, C or D. For each starting point option, we examine all available neighboring vertices. A new candidate path is generated after each step. We end up having 9 candidate paths. Coincidently, every chosen starting point generates three candidate paths. It does not necessarily always true that the chosen starting points contribute evenly. The paths generated from a particular map with a chosen starting point are determined by the obstacles of the map, the position of the starting point and many other factors.

For each candidate path, there are several data fields associated with it, a distinct sequence of vertices that represent the path, a battery consumption value by taking this path, a summation of priorities that can be collected by taking this path. The sequence of vertices is of course the final output required as the project's goal. The battery consumption tells us the feasibility of the path when taking into consideration of the battery capacity constraint. The summation of the priorities helps us to eventually decide that which one is the final maximization solution among the feasible paths.

Notice that in this example we first list out all possible paths without considering the battery capacity input. This is to assist the later analysis for all possible combinations of different battery capacity input values. In the algorithm, however, once the base case of battery consumption overflow is hit, program would not compute any of the following paths along this way. For instance, in the figure below, if the path 2 is determined to have a battery consumption overflow, path 5 and path 9 would not be calculated.

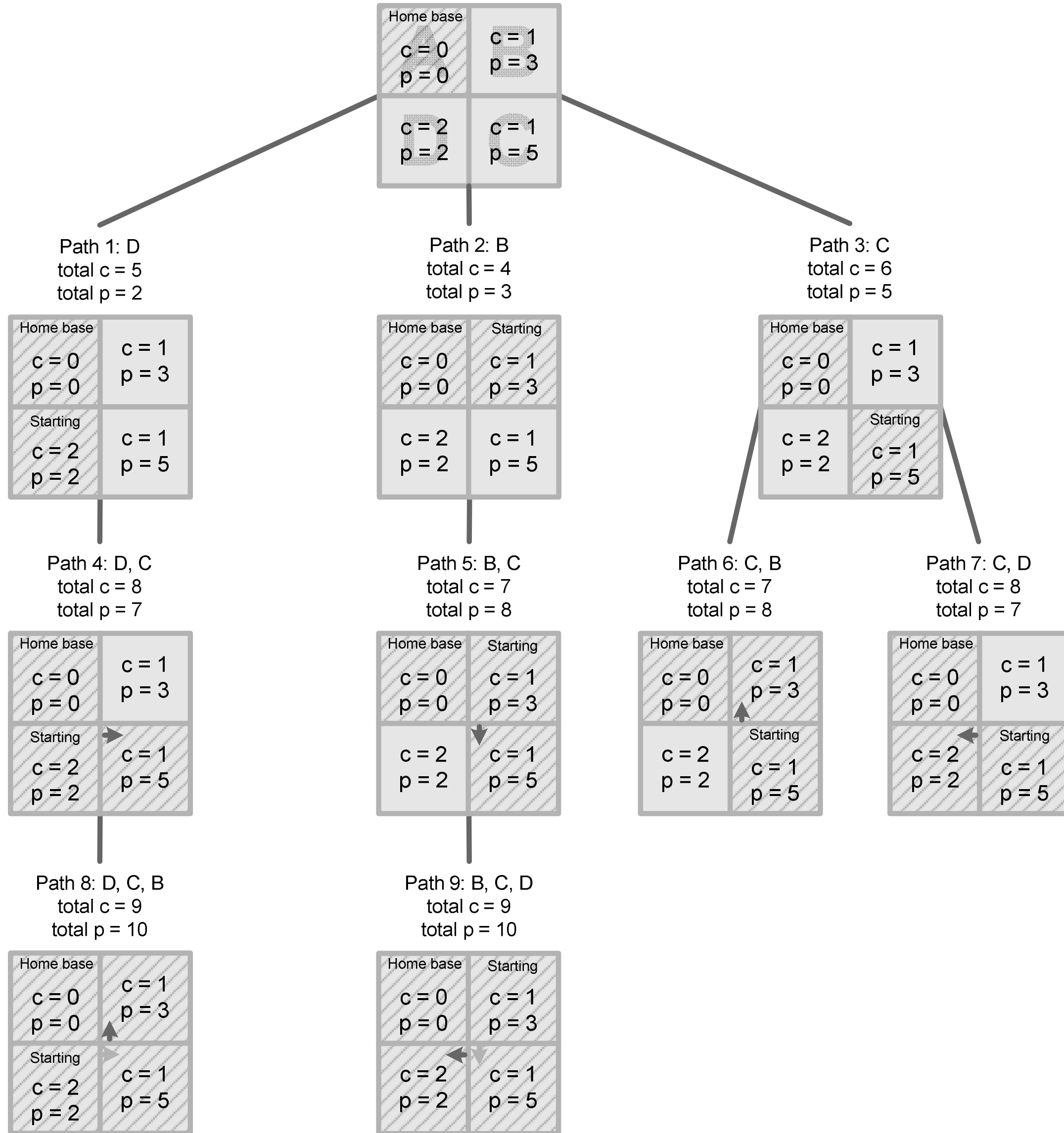


Figure 8. A simplified example of our algorithm.

After these computations, we generate the candidate paths and the field data that we care about for each of them. Now, if we take into consideration of the battery capacity constraint, we would get the final optimization solution. We use the following tables to display the relationships between different battery capacity inputs with the candidate solutions.

Path of Vertices	Battery Consumption	Summation of p Values
D, C, B	9	10
B, C, D	9	10
C, B	7	8
B, C	7	8
D, C	8	7
C, D	8	7
C	6	5
B	4	3
D	5	2

Battery Capacity Input C	Optimized Solution Path
$C \geq 9$	D, C, B or B, C, D
$7 \leq C < 9$	B, C or C, B
$C = 6$	C
$4 \leq C < 6$	B
$C < 4$	null

By observing the second table, the relationship between the battery capacity input and the final optimal solution in this mesh graph is clear. When we have more than 9 units of battery life, we can travel the longest path in this graph, which include all of the vertices of the graph. Keep in mind that the longest path may not include all of the vertices, although in this particular example, it does. If we have battery life 7, or 8, since we consider this variable is an integer, we get to clean vertex B and C. The order of these vertices does not matter. If we have a battery life of 6, we get to clean vertex C. If we have a battery input of 4, or 5, we would clean B. A battery capacity input less than 4 is not enough to clean any vertex of this graph.

Notice that for the paths that contain more than one vertex, there are always reversed paths that provide the same summation of p values and the same battery consumption. This is because the traveling behavior is symmetric. Once a particular path is chosen, the ending point of this path can be used as the starting point as well.

4 Computational Complexity Analysis

In this section, we shall analyze the complexity of the algorithm based on the pseudo code. First we discuss the time complexity. The time complexity expectation of this algorithm should be slow

because this is a brute-force implementation. The space complexity expectation of the algorithm should reasonably ideal.

4.1 Time Complexity

The time complexity of this algorithm is analyzed based on the pseudo code. The first three methods are helper methods to perform computation expressed in the Jade-Mesh-OuterLoop and Jade-Mesh-Recursion methods.

4.1.1 Time complexity of Jade-Mesh-OuterLoop

It is obvious that the Jade-Mesh-OuterLoop loops through all vertices in the graph. The graph is stored as an array, the two coordinate indexes, i and j are used to access the vertex associated information in the graph. Therefore, the Jade-Mesh-OuterLoop method runs at a time complexity of $O(V)$.

4.1.2 Time complexity of Jade-Mesh-Recursion

The Jade-Mesh-Recursion method is a simple recursive function, which checks two base cases and recursively call itself by passing in the next vertex position. We shall use recursion tree to estimate the runtime of this algorithm and prove the guess use substitution.

The base cases checking in Jade-Mesh-Recursion method take $O(1)$ time, since the helper methods isBlocked and isBatteryExhausted both take $O(1)$ time. The recursive calls can be expressed as the equation (1) below:

$$T(n) = 4 \cdot T(n - 1) + O(1) \tag{1}$$

Apparently, equation (1) can not be solved by the master's method because 1 is not a valid base number for logarithm. We draw the following recursion tree. By examine the recursion tree, we guess that the run-time of Jade-Mesh-Recursive runs at a time complexity of $O(4^n)$.

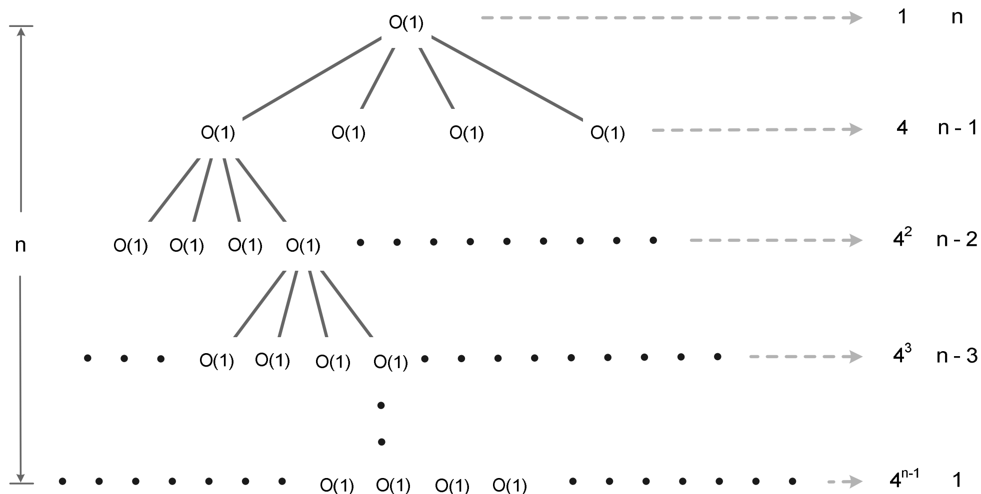


Figure 9. The Recursion Tree used to estimate the time complexity of our algorithm.

In order to prove correctness of our guess using induction, we need to first proved the bases case/cases. The base case for this algorithm is when $n = 1$. As we can see from the recursion tree, the program takes $O(1) = O(n^0)$ time to compute in this case. We can also test $n = 2$. The program takes $4 \cdot O(1) = O(n^1)$ time in this case. The base cases are therefore proved.

Then we need to further prove the common cases hold. We assume that $O(4^n)$ holds for $T(n - 1) \leq c \cdot 4^{n-1}$, of an appropriate choice of constant $c > 0$. We need to prove that it holds for $T(n) \leq c \cdot 4^n$.

$$\begin{aligned} T(n) &= 4T(n-1) + O(1) \\ &\leq 4c \cdot 4^{n-1} + O(1) \\ &= c \cdot 4^n + O(1) \\ &\leq c \cdot 4^n \end{aligned}$$

For any constant $c > 0$ the conversion above holds. Therefore, we proved of the run-time complexity is $O(4^n)$ of this algorithm by induction. The input size of Jade-Mesh-Recursion is the number of vertices, therefore the run-time analysis of Jade-Mesh-Recursion is $O(4^v)$.

4.1.3 Time complexity of the entire algorithm

As we decided, the outer loop of this algorithm, the Jade-Mesh-OuterLoop method, takes $O(V)$; the loop body, Jade-Mesh-Recursion method, takes $O(V^4)$. The overall run-time of this algorithm is therefore the multiplication of these two.

$$O(V) \times O(4^v) = O(V \times 4^v) \tag{2}$$

The recursive method, Jade-Mesh-Recursion, runs at an exponential time $O(4^v)$., the outer loop, Jade-Mesh-OuterLoop, runs at a lower term $O(V)$. Therefore the combine also runs at an exponential time, $O(V \times 4^v)$, as shown above.

4.2 Space Complexity

The space usages of this algorithm are mainly associated with the several data collector ADTs. We used two matrices to store the information regarding the mesh graph. The input matrix stores information such as the priority values and battery consumptions for the all vertices. The Boolean matrix stores information about the vertex coverage. These two data types take a space linear proportional to the number of input graph vertices, $O(V)$. We also have a Vertex data type which stores the coordinate information of the vertices in the input mesh graph. The stack of Vertex temporally stores the vertices on a certain path. It takes a space of $O(2V) = O(V)$ in the worst case. Finally, we have the graph solution stored in its own data type. It stores a stack of Vertex and a single integer. Therefore the space complexity is $O(2V + 1) = O(V)$.

Sum them together, it is clear that the space complexity of this algorithm is $O(V)$. We do no need to consider the space recycling regarding the collections described in the previous paragraph because there is no higher term space consuming components. The summation would be $O(V)$ with a constant factor, which does not change the final analysis. Therefore, we present the space complexity of our algorithm is $O(V)$.

4.3 Improvement attempts

In complexity theory, exponential time is the computation time of a problem where the time to complete the computation, $m(n)$, is bounded by an exponential function of the problem size, n. In other words, as the size of the problem increases linearly, the time to solve the problem increases exponentially. In Computer Scientists, people sometimes think of polynomial time as "fast", and anything running in greater than polynomial time as "slow"[5]. By this definition, exponential time would therefore be considered slow.

If there is any possibility, we would like to improve the algorithm's performance. On the other hand, the improvement needs to be significant. By lowering the lower term, for example the Jade-Mesh-OuterLoop method would not change the exponential property. In other words, we need to answer the question whether there is polynomial algorithm that solves our problem.

As we discussed previously, for our problem, if we have more than enough battery capacity and moderately even priority values of the vertices, the problem turns into a problem of looking of the longest path. Actually, without the second constraint regarding the priority value, we would also need to look for the longest path and compare the priority value of this path with other candidate solution paths.

Since we addressed the worst case of this problem is the longest path problem, we can now answer our question—whether there is polynomial algorithm that solves our problem. The longest path problem is known as a NP-complete problem in Computer Science[6].

Instance: Graph $G = (V, E)$.

Solution: Simple path in G, i.e., a sequence of distinct vertices $v_1, v_2, \cdots v_m$,

such that for any $1 \leq i \leq m-1$, $(v_i, v_{i+1}) \in E$.

Measure: Length of the path, i.e., the number of edges in the path.

The definition of NP-complete problem tells us that there is no known efficient way to locate a solution of the problem. The most notable characteristic of NP-complete problems is that no fast solution to them is known. That is, the time required to solve the problem using any currently known algorithm increases very quickly as the size of the problem grows. As a result, the time required to solve even moderately large versions of many of these problems easily reaches into the billions or trillions of years, using any amount of computing power available today[7].

---

[5] http://en.wikipedia.org/wiki/Cobham%27s_thesis

[6] http://www.nada.kth.se/~viggo/wwwcompendium/node114.html).

[7] http://en.wikipedia.org/wiki/NP-complete#Formal_definition_of_NP-completeness

Therefore, at least the worst case in this problem does not have a polynomial solution. We are not likely to find an algorithm runs at a speed significantly faster than what we presented here in this paper.

5 Conclusion

This paper proposed a brute-force algorithm with a exponential run-time to solve the problem of maximizing the summation of priorities through a vertex-weighted, undirected mesh graph. Potentially, the algorithm presented in this paper can be used as an implementation of a Roomba optimization feature. A simplified example was used to demonstrate the problem solving procedure. The paper provided a comprehensive problem formulation with its preliminaries, algorithm design, and the complexity analysis for the algorithm. We proved that it is unlikely for us to find a polynomial algorithm that solves this problem because the worst exists in this problem can be converted into to a well-known NP-complete—the longest path problem.