Topography and Behavior Based Movement Modeling for Missing Hikers in Land-Wilderness Settings

J. Alanis[1,a], M.M. Brown[2,a], J. Kitchens[3,a], J. Magaña[4,a], C. Velastegui[5,a], M. Thakur[a], L. Arriola[6,a], B. Espinoza[a], A. Murillo[a], M. Rodriguez-Messan[a], R. Koester[7], and C. Castillo-Garsow[8,a]

[1]San Joaquin Delta College, Stockton CA, United States

[2]Scripps College, Claremont CA, United States

[3]Warren Wilson College, Swannanoa NC, United States

[4]University of North Georgia, Dahlonega GA, United States

[5]Yachay Tech University, San Miguel de Urcuquí Imbabura, Ecuador

[6]University of Wisconsin-Whitewater, Whitewater WI, United States,

[7]Kingston University London, London, United Kingdom

[8]Eastern Washington University, Cheney WA, United States

[a]Simon A. Levin Mathematical, Computational, and Modeling Sciences Center: Mathematical & Theoretical Biology Institute, Arizona State University, Tempe AZ, United States

July 2019

Abstract

Search and Rescue (SAR) operations are critical to the safety and well-being of individuals who visit state and national parks. In a missing hiker case, time is crucial as survival rates decrease each hour. It is estimated that approximately 4080 individuals become lost every year in the US. [21] The average cost of a SAR operation is $1,375 per person, and the cost of each mission is increasing yearly. Most SAR operations are based on rings of probabilistic distances. There is a need to incorporate a mechanistic, mathematical model that takes human behavior into consideration for determining SAR operations. Data from resources such as the International Search and Rescue Incident Database are analyzed to identify patterns in human behaviors and key geographic influences to develop a mechanistic model of missing persons. We use a discrete-time Markov Decision Process, where the lost individual's state is used to determine a personal strategy for being found. The individual then interacts with the environment, where a utility function for that strategy over the geographic environment determines direction of travel. We take incident reports in various national parks as a case study to test our model. Implications are discussed for SAR, hiker survival training, and other aspects. The proposed model may be extended to the prediction of path-tracing of specific groups of people including experienced hikers or individuals who suffer from mental illnesses.

Introduction

Background

Every year, it is approximated that 4080 individuals become lost in the U.S [21]. In Yosemite National Park alone, 393 search incidents were reported from 2000 to 2010 [5]. There is a direct, negative correlation between survival probability and the duration of the search. After the first 24 hours, the probability of survival drops to 75 percent. By the third day, the probability of a hiker surviving decreases to 50 percent. For lost children[1], this probability is even lower, often below 25 percent [16]. Every hour lost reduces the probability of being found [23].

Missing person cases are handled by various federal, state, and local authorities, which are grouped together under the general label of "Search and Rescue", also known as SAR. Search missions are conducted differently depending on the time, location, and condition of the lost individual. Most often, the search begins by defining a probability of area (POA), which is a bullseye surrounding the hiker's position last seen (PLS). If the hiker is not found inside the POA, the search then extends to the rest of the world (ROW). SAR leaders are responsible for deciding when a search mission needs to switch from the POA to the ROW. This decision is often guided primarily by intuition[2], and frequently either too much or too little time is spent searching in POA versus ROW [11]. After determining the search area, SAR teams begin searching

[1]The lost children from the data are between the ages of 4-6 and 10-12 years [16].

[2]Hill describes this as subjective probability and explains the reasons why this is not a reliable method of decision-making in detail [11]

on foot. Commonly, teams will begin with interviewing those who last saw the person, determining if the person's estimated arrival time has passed, looking for traces of the missing individual, etc. [5].

Search missions, including relatively simple operations, are expensive to conduct. Moreover, some search operations can require air support, which can cost upwards of $1600 per hour of operation [2]. The average cost of a SAR operation is $1,375 per person [21]. In 2012, search missions cost local and federal governments 5.3 million dollars, and the cost continues to rise [21]. Additionally, a large amount of volunteer time and risk is involved with nearly every rescue mission [9].

Lost Person Behavior

The path that a missing individual selects is heavily influenced by their mental state, making behavioral analysis critical to the prediction of a missing person's direction of travel [23]. The scope of our model is limited to lost persons in a wilderness environment, so it is important to clearly understand the conditions under which a person becomes lost. The state of being "lost" in a human context is formally defined by Hill [10] as a condition in which an individual cannot determine their geospatial location and has no means to orientate themselves relative to their surroundings. Jones-Christensen and Hammond add that "lostness" is commonly understood as the disparity between where someone currently is and where they wish to be [13]. Dudchenko describes two ways in which a person may become lost, namely misorientation and disorientation. Misorientation happens when a person is placed in an entirely new environment or exposed to new elements in a familiar environment. Weather phenomena, such as fog or rain, can block the visibility of landmarks in the terrain, which are necessary for orientation. If recognizable landmarks cannot be seen, the individual can become lost. Another form of misorientation occurs when the individual does not sense slight changes in direction; they may believe they are following a straight route, when in reality they are not. In misorientation, the individual maintains a spatial awareness of the landmarks beyond their field of vision, however their direction is misperceived. Disorientation occurs wherever there is a disconnect between direction and distance. This means that the location of an object is not well understood in the mind of the traveler, and the person is unable to self-localize. In other words, disorientation is a breakdown in the map itself; the individual loses both their sense of direction and spatial awareness of the landmarks beyond their field of vision. This is different from misorientation, because the person cannot identify any landmarks around them [7]. In disorientation, they are entirely lost.

Jones-Christensen and Hammond [13] identify the psychology of being lost in 5 ordered stages, namely *Separation*, *Isolation*, *Deviation*, *Deprivation* and *Realization*.

- *Separation* occurs when a person is physically and psychologically lost in their environment. The missing person is physically lost because they do not know where they are. When a person becomes psychologically separated, they feel locally lost, realizing that other people are not around. They feel capable of saving themselves if need be, but still have hope in finding other people.

- *Isolation* occurs when the individual feels lost in a global sense. They lose hope in finding other people, causing them to feel as though they have to save themselves.

- *Deviation* is when the person acts unrealistically, with an incomplete view of the situation. For example, the lost individual may deny that they are lost in an attempt to rescue themselves.

- *Deprivation* happens when the person becomes very emotional, feeling hopeless or panicked. They feel they have to save themselves because no one else cares.

- *Realization* occurs when the person gains acceptance and comes to terms with the situation, which can result in either productive or counterproductive behavior [13]. In productive behavior, the individual feels capable of being located. In counterproductive behavior, the individual loses hope.

In the fist two mental stages, the individual is actively trying to find their way, but in the last stage, they are likely to wait for rescue. Therefore, the mental condition of a person determines their confidence [3] with the landscape and, by extension, the strategies they are most likely to adopt.

There are different strategies an individual can adopt for the purpose of increasing their chances of being found, or reorienting themselves. Hill [10] defines 9 strategies, that are commonly observed, namely: Random Traveling, Route Traveling, Direction Traveling, Route Sampling, Direction Sampling, View enhancing, Backtracking, Using Folk Wisdom and Staying Put. These strategies are described further under the "Methods" section. Traveling cases are the most challenging for SAR teams. Understanding the mechanisms that describe the movement patterns of lost persons, even outside of the highest-probability area, can be effective in improving current SAR protocol.

[3] Confidence is defined as how likely the individual is to rescue themselves, or wait for rescue. The correlation between lost person behavior and confidence is further defined in the subsection of "Methods" entitled, "Decision Tree"

Effects of Topography

The location of a lost person impacts their decision making. For example, if the hiker is at the top of a peak, both their vision and chance of rescue improves [22]. On the contrary, steep terrain can limit a hiker's field of vision, thereby affecting their decision-making process. Decisions are often driven by a desire to increase the amount of available information. Studies show that a third of lost hikers travel uphill in hopes of improving their field of view [16]; however, most individuals prefer to take a clear path rather than a steep route [4, 5]. This preference for easier paths, coupled with a limited knowledge of their surroundings, limits the number of destinations the person may consider. Hikers will attempt to avoid traveling across certain terrains, including unlevel areas, sloping terrain, cliffs, and so forth. This is especially true if the hiker has low energy.

Previous Models for SAR

While other models only look at the last known physical location to track a missing person, or only consider behavior from a statistical point of view, our model takes both lost person behavior and geographical characteristics into account. For example, Castle's [3] model uses the steepness of terrain as a limiting factor on the individual and a discrete-time Markov chain to create a probability map; however, this only accounts for geography, and neglects human decision making entirely. As another example, Feo *et al.* [8] and Johnson's [12] independent models also only account for the difficulty of terrain. Their models use data from digital elevation models (DEMs) to measure the difficulty of the surface, ranging from clear paths to impassable rivers. While both models account for the average speed of the hiker, they provide no mechanism for modeling the decision process. Some models attempt to be more accurate by using GIS data, coupled with demographic data on human behavior, to assign general probabilities to areas on a map. This can be seen in models independently proposed by Doherty *et al.* [4] Doke [5], and Sava [21] *et al.* , all of whom use this information to delineate a *probability of area* (POA). While this is useful in certain aspects, it still does not account for the way individuals make decisions. We axiomatically accept that individual behavior will always deviate in some way from the population. Taking this as a modeling consideration affords us the opportunity to provide a more accurate framework for a missing person's travel patterns, since the missing person is an individual, and not a population.

As previously stated, individual factors such as geographical elevation, energy, and mental state play a fundamental role in the decision making process. Furthermore, since individuals do not act in a deterministic fashion, locations of tiles are treated as probabilities that change over time, which introduces an element of stochasticity to our model. This is significantly different to current models, the majority of which are statistical in nature and ill-suited at the individual scale. In short, our model improves upon prior ones by using a mathematical decision-making framework, which considers the interaction between the individual and the environment. This, in turn, provides both more focused information for SAR teams, removing many presumptions and narrowing the scope to high-probability areas. Our model will help improve the search for a missing person by better predicting where they are most likely to be.

Purpose of Study

How can the influence of *terrain* in combination with *lost person behavior* be modeled *mechanistically* to predict a *lost hiker's movement* in a land-wilderness context? To answer this question, we describe and analyze a novel, mechanistic model using a Markov Decision Process (MDP), specifically a Decision Tree Algorithm (DTA) coupled with a heuristic path finding algorithm, to predict the movements of a missing hiker in a land-wilderness context. The DTA is constructed so as to mimic human decision making in a "lost" cognitive state. The model then considers how the environment (elevation changes and notable landmarks) influences the individual's decision making process. By taking these geographic features into account, the model outputs a probability distribution of the hiker's location for each time step. The results are analyzed and discussed for possible improvements and modifications to the model are considered for future work.

Methods

Overview

In order to model how the changing state of the individual drives decisions, we implement a Markov Decision Process (MDP) using a Decision Tree Algorithm (DTA), and Dynamic Programming techniques. A visual representation of the MDP is shown in Figure 1. The model tracks an individual's general state over time by maintaining four specific agent-states, namely location, energy, mental, and current strategy. The geographic area is divided into tiles, and the missing hiker's location is bound to one or more tiles. We define energy as a state variable in terms of kilo-calories. As energy decreases, actions become limited, with the person becoming incapacitated when energy equals 1. A quantity for mental state, denoted M, is also introduced in the model, where mental state decreases at a pre-defined rate μ, which is defined as change in mental state per time. Current strategy is defined as a method which the individual chooses to use in an attempt to reach their goal/destination. The mental, energy, location states, and current strategy are inputted into the DTA, which then outputs a destination and strategy for the hiker at a given time-step. This information is fed into the heuristic algorithm, which is driven by either a Bellman equation[4] or a novel Elevation-Distance Equation[5]. Most commonly, the Bellman equation is used to optimize the most energy-efficient path for a given situation. We have modified this to return a ranked order of paths, with probabilities associated with expected energy efficiency, explained in more detail later in this paper. In both equations, probabilities are then determined for the next time-step $t + 1$ of each tile with non-zero probability at time t. In other words, each of the neighboring tiles (relative to each tile with non-zero probability) is given a new probability of being occupied in the next time-step. In this context, non-zero probability means there is a chance the hiker is on the tile. Elevation data and other factors are maintained separately and are used to determine the energy cost necessary to transition from one location state to another. In the Bellman equation, reward is measured in terms of expected future energy cost, which drives the path ranking decision process shown in Figure 1.

Markov Decision Process

For the Markov Decision Process, we begin with terrain generation, which generates a topographic map based on the hiker's place last seen (PLS). This determines the hiker's location state, which dictates their field of vision. This impacts the objects the hiker can view. This information is fed into the DTA along with the mental state, which is based on the 5 stages of Lost Person Behavior. The energy and strategy states are also fed into the DTA. At the initial time step, there is no strategy, but hereafter, a strategy is formed and flows back into the strategy state. After each state is fed into the DTA, a destination is given, which is then fed into the Heuristic Algorithm of either the Kitchens (Elevation-Distance) or Bellman equation. Because the hiker moves another tile, the entire model is updated at every time step. The cost of moving from tile to tile decreases an individual's energy, but allows an individual to increase their reward[6]. This forces a balance between energy intensive movements, where the hiker may take less time steps to reach the goal than low energy movements.

[4]Named after Richard Bellman (1920-1984), who laid the foundation for Dynamic Programming in classical AI
[5]Affectionately called the "Kitchens" equation, after James Kitchens, who developed it based on our understanding of the path finding process while the Bellman equation was being defined.
[6]Reward is defined as a closer distance to the destination/goal.

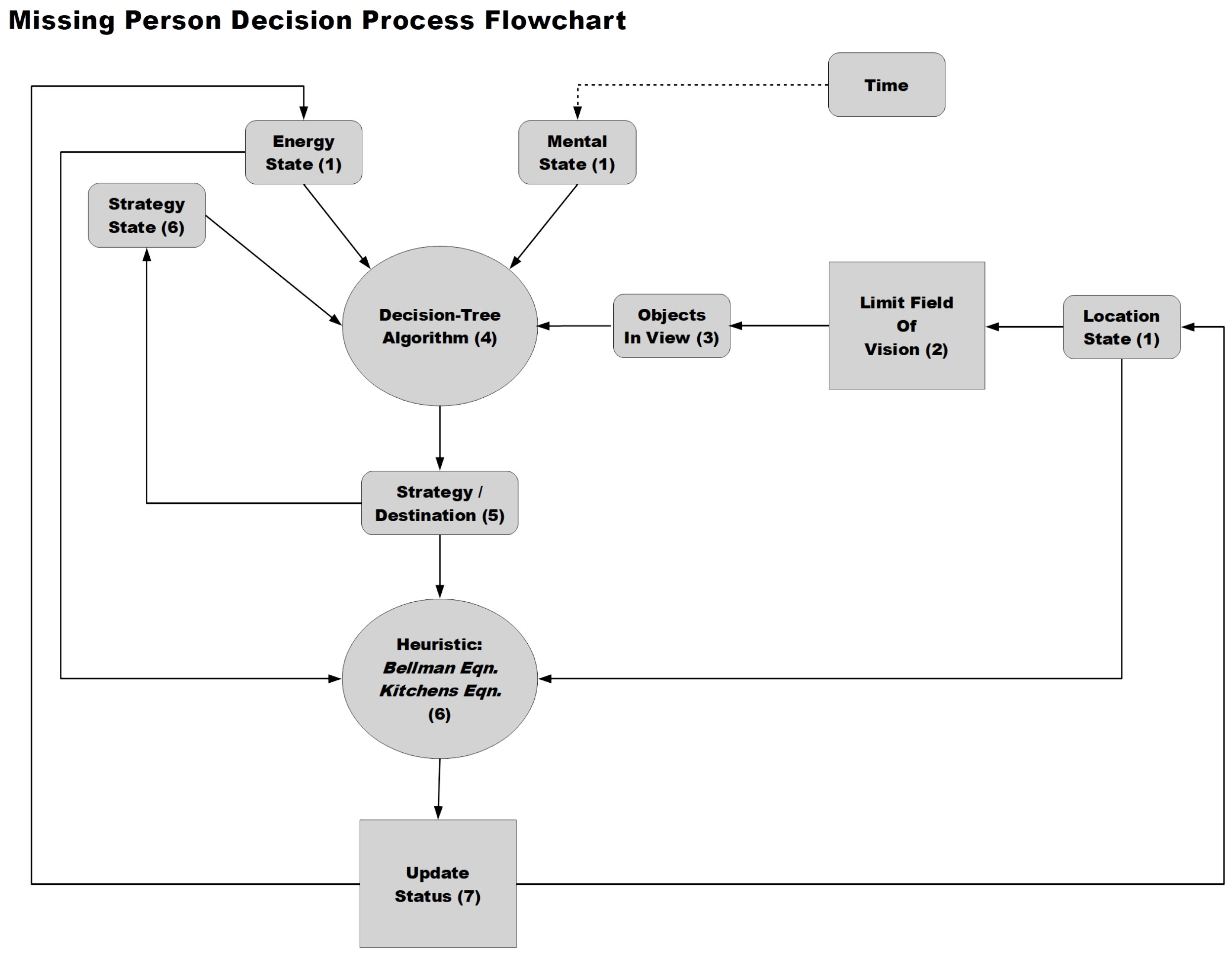

Missing Person Decision Process Flowchart
Time
Energy State (1)
Mental State (1)
Strategy State (6)
Decision-Tree Algorithm (4)
Objects In View (3)
Limit Field Of Vision (2)
Location State (1)
Strategy / Destination (5)
Heuristic: Bellman Eqn. Kitchens Eqn. (6)
Update Status (7)

Figure 1: Figure 1: Complete MDP Flow Chart. *(1) At the beginning of each time step, the hiker has a certain energy, mental, and location state. If* $t \neq 0$*, the hiker also has a current strategy. (2) The location state is used to determine what landmarks are within view of the hiker, and the available potential destinations are listed (3). These are all fed into the DTA (4), which outputs a strategy and destination (5). The strategy state is updated and the destination is fed into the heuristic algorithm (6). This results in a selected "next step". Once the hiker moves to this new tile, the location and energy states are updated (7). The mental state decreases at a constant rate after a fixed number of time steps.*

Assumptions

In the context of our model, certain assumptions are made about the hiker. These may not be true in all cases, but are meant to serve as a baseline for a typical lost hiker. In order to model the probability of finding a lost hiker, we must first assume the hiker is lost in a land-wilderness environment. An exact PLS is not needed, but we assume that the hiker became lost somewhere on the map upon which the model is applied[7]

[7]For example, if the model is applied to a map of a national park, and the hiker wanders outside the bounds of that map at a certain point, clearly the model is no longer valid after that point.

For the hiker to be entirely lost, we assume they do not have any navigational aids such as a compass, map, or GPS. We suppose that the lost hiker may have some prior experience, but is not a trained survival expert. We also preclude any mental illness that would affect their decision making, and we assume that the hiker has a will to live. We presume the hiker is able-bodied; not suffering from any debilitating injury, as injuries may effect the types of decisions they make. We also assume that the terrain, specifically elevation and the presence of notable features, are the only geographic features that influence decision-making[8]. Additionally, for the time being, we exclude changes in weather and day/night cycles from consideration in the context of our model. We shall revisit this limitation in the "Discussion" section of the present manuscript. Finally, we assume that a hiker is not a forager. That is to say, energy is always decreasing, and that the hiker's decisions are not influenced by the availability of food or water.

Simulation

Code is implemented using Python (Version 3.5). Below we describe various sections of the code which drive the simulation of a lost person moving across a terrain. The source code is available in its entirety in the appendix.

Terrain Generation

As previously stated, terrain generation creates a topographic map based on the hiker's last known location. As a hiker's decisions are influenced by elevation, it is a key factor in determining the path of a missing person. First, a high resolution terrain map is randomly generated using the OpenSimplex noise module [9]. Then, a topographic map is created by generating the contour lines from the elevation data. Lastly, a low resolution version of the map is created by pixelating the original, high resolution version. For the low-resolution map, each pixel represents one tile. A visual representation of the topographic map can be observed in 2. The majority of the simulation utilizes the low resolution map as opposed to the high resolution map to increase the speed of analysis. Hereafter, it can be assumed that the phrase "terrain map" refers to the low resolution map unless stated otherwise. The elevation changes between pixels are calculated and used to identify landmarks, such as peaks and saddles. For a pixel to be identified as a peak, all other pixels in its von Neumann neighborhood must have a negative elevation change [10]. For a saddle, an opposite pair of pixels in its von Neumann neighborhood must have the same sign, but opposite from the other pair.

[8]This kind of terrain is very common in northern Arizona/southern Utah. Capitol Reef National Park provides a ready example

[9]Python Library to generate random numbers with smooth noise

[10]von Neumann neighborhood includes all pixels directly adjacent to a pixel in a single direction (up, down, left, and right

For simplicity, it is assumed that the lost individual can locate the nearest peaks and saddles (landmarks) that appear on the map. This information is then passed as input into the DTA, which determines the hiker's strategy.

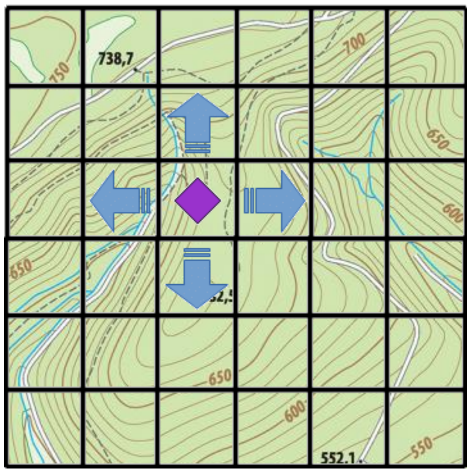


Figure 2: Visual Representation of square-grid rasterization over a topographic map. Each tile is reduced to one pixel during calculations. The purple diamond represents the position of the hiker and the blue arrows refer to the possible movements. Credit (Topo. Map): (Lesniewski, 2018)

A high resolution map (700 pixels x 700 pixels) containing peaks and saddles (SEED: 44259 (Windows)) is used as a setup for all simulations. For the simulations themselves, a low resolution version of the map (70 pixels x 70 pixels) is created from the high resolution map.

Decision Tree

We have generated a decision tree that combines the inputs of an individual's mental state, energy, and objects in view to determine the individual's next strategy. We use the 5 stages of being lost (previously stated in the background section) to define confidence. Confidence refers to the likelihood the hiker will rescue themselves or wait for rescue. Using the 5 stages of lost person behavior, we divide confidence into 3 stages, consisting of high, medium, and low. During high confidence, the hiker has hope of realizing where they are. This stage would refer to separation and isolation. Medium confidence is a bridge state, where the hiker tends to move unpredictably. This state takes place during the deviation stage. Low confidence represents the state when the hiker has lost their decision making ability and, overwhelmed by the situation, tends to inaction. This occurs during deprivation and realization.

The energy of the individual is reduced to 3 categorical states: high, medium, and low. These energy states are used to limit the decisions available to the individuals as their energy decreases. For example, a hiker who is low in energy should not attempt to seek higher elevation goals, as this would deplete their energy entirely.

The path-finding algorithm outputs integers for the confidence category and energy state. These integers are divided into high, medium, and low and inputted into the decision tree.
After the initial time step, the individual has a strategy which is updated in the current strategy state. This is then fed into the DTA. Field of vision is defined as what the individual is able to see from their current location. This includes any notable landmarks (peaks and saddles, specifically) that may be used as directional goals by the hiker. Lastly, the decision tree considers the current strategy employed by the hiker when deciding on the next strategy. We found 9 strategies in literature [10], stated as follows:

1. *Random Traveling*: The individual impulsively chooses a path, instead of choosing a direction or goal to move towards.

2. *Route Traveling*: The person decides to follow a path, road, watershed, or other kind of guide.

3. *Direction traveling*: The lost person selects one direction and attempts to maintain that trajectory.

4. *Route sampling*: The person chooses an intersection on a clear path, for example, a forked trail, and travels some distance before backtracking and trying the other intersections.

5. *Direction sampling*: This is very similar to route sampling, however, the person does not have access to clear paths. To account for this, they choose a direction relative to a landmark and sample it.

6. *View enhancing*: The person searches for a position of height to view landmarks in the distance.

7. *Backtracking*: The person reverses their direction of travel, in an attempt to find something familiar.

8. *Using Folk Wisdom*: The person will apply strategies based on common knowledge they were previously told, for example, "Following a river leads to civilization". [16]

9. *Staying Put*: Mental state and energy level influence the decision to stay in place and wait for rescue (either from SAR or family) [10]. Oftentimes, staying in place is more beneficial, as traveling causes the hiker to lose energy and can complicate the search itself [10].

In our model, we simplify this list to four strategies: Directional Traveling, View Enhancing (Landmark), Backtracking and Staying Put. Currently, rivers and paths are not included in the terrain maps. Because of this, strategies pertaining to the these features are not included. Other strategies have been enclosed within these four selected strategies. Random Traveling and Using Folk Wisdom are included within Directional Traveling. Directional traveling and backtracking are included in directional sampling, where the individual samples a path and then goes back to their starting point. The process of decision making was based on literature that discussed how lost hikers evaluate their surroundings and decide on a strategy [17]. Figure 3 shows the decision tree, the flowchart of the algorithm on deciding the next strategy. The potential combinations of inputs and their output strategies can be found in Appendix [A.1]. The decision tree includes a table with all possible outcomes based in conditionals of mental, energy, location, and current strategy states. The conditional, confidence depends on the hiker's mental state. We denote the quantitative value of "confidence" as M, which ranges from zero to one hundred. For example, if M is high (above 66) the hikers can decide to orientate themselves, so that they will continue traveling. Another condition is energy, which ranges from 0 to 3000 Cal. Low energy limits the possibility of the hiker traveling to a high point. The condition of field of vision allows the hiker to travel forward, backward, and stay put The last condition is the current strategy which determines if the hiker will change strategies. The output of all these conditions are the 5 strategies previously defined.

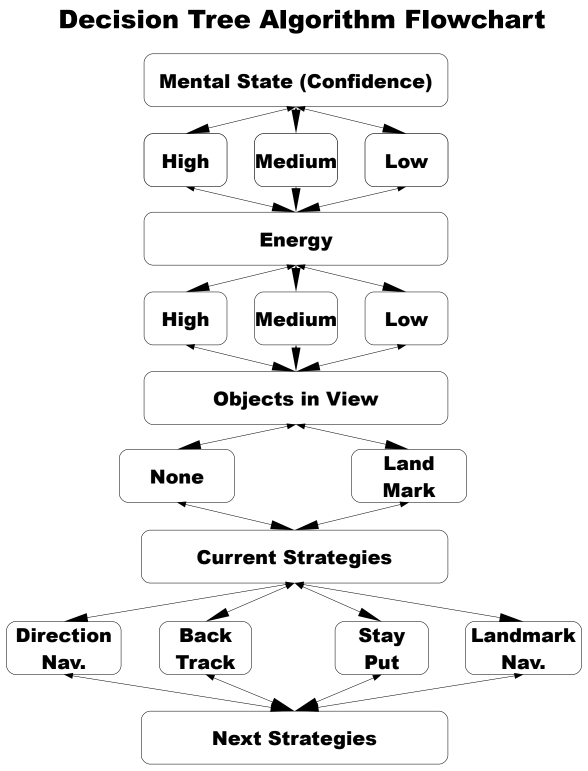


Figure 3: Decision Tree Diagram: *The DTA is used to determine the destination, based on a selected strategy. Influencing factors are: Confidence (mental state), Energy state, Objects in View, and Current Strategy. The DTA uses these conditions to determine the next strategy at each time step. If confidence, energy, and objects in view remain the same, then the list of strategies also remains the same.*

Goal Selection

Once a strategy is selected in the DTA, all of the potential goals that correspond to that strategy are evaluated. For example, an individual employing the landmark strategy will identify the closest peak or saddle as their goal destination. Unique to the landmark strategy is that if the individual has a high or medium level of confidence, they will continue to move after arriving at a selected landmark. This is implemented in the program by removing the landmark as a potential goal after fifty percent of the probability cloud reaches the landmark, forcing the hiker to select the next closest strategy. In terms of the hiker's psychology, this represents different behavior of a hiker who stays in place and a hiker who chooses to keep moving. During backtracking, the hiker will attempt to walk to the location where their last strategy changed. When utilizing the Stay Put strategy, the individual does not move. Lastly, in employing directional traveling, the hiker selects a direction to walk (with goal set at the horizon line). Determining the goal for hikers using directional traveling is difficult, as we do not know which direction the hiker may select. Unlike other strategies, which attempt to move towards a location or goal, the goal of directional traveling is to move away from the start location.

Heuristic Path Finding Algorithm #1: Elevation-Distance Equation

Path finding is performed using a forward, heuristic algorithm that mimics the decision making process of a lost individual with minimum view and knowledge of their environment. In the present framework, two factors affect the individual's path to their goal destination: elevation change and distance to their goal. Distance to the hiker's goal was measured using Manhattan distance[11]. It is assumed that a lost individual will prefer to take the path with smallest elevation change over the shortest distance. Varying energy required to traverse different terrain is taken into account. These two factors are combined to describe the probability of moving from one pixel to another (Figure 4).

The Elevation-Distance (Kitchens) equation is a heuristic path finding algorithm that takes into consideration the changes in elevation and the distance to the goal. For each transition (adjacent tile with respect to one neighbour) we define:

$$g = e^{\frac{-|\Delta h|}{k_e}} \tag{1}$$

where k_e is a scalar value that describes the influence elevation on the hiker's decision-making. Then we normalize each elevation weight.

$$f(g) = \frac{g}{\sum g} \tag{2}$$

We do the same here, but with Manhattan distances to the goal. So for each transition we have

$$q = e^{\frac{-|\mathscr{D}|}{k_d}} \tag{3}$$

where k_d is a scalar that describes the influence of the new pixels distance to the goal on the hiker's decision-making. Normalization is given as follows:

$$f(q) = \frac{q}{\sum_{i=1}^{5} q_i} \tag{4}$$

[11]The Manhattan distance is a metric of distance that assumes that an agent navigating a square grid can only move in four directions: forward, backward, left, or right. The name is derived from grid-block structure of New York City

Where each i represents one of the four possible directions, plus the option of staying put. With the changes are in normalized elevation weights and normalized distance weights, we define the following expression:

$$c = f(g) * f(q) \tag{5}$$

$$f(c) = \frac{c}{\sum c} \tag{6}$$

where $f(c)$ is the probability that a hiker moves to the tile in question. This information is saved in a matrix that, which contains the probability of the hiker inhabiting each tile at each time step, which yields the probability cloud.

Heuristic Path Finding Algorithm #2: Bellman Equation

Path selection is the result of a decision-process driven by a heuristic algorithm of some description. The alternate method used in this paper is the Bellman Equation for a finite time-horizon problem [1]. Normally, dynamic programming is applied in the context of path optimization. However, we apply the bellman equation here in a different way. Specifically, we use the equation as a means to rank the probability of a hiker transitioning to a specific tile i based on the energy cost of each route. This idea was inspired by the forager model presented by Mangel and Clark [20]. Table 1 contains the definitions of the terms used in this section. The expected energy cost of the hiker at time t who takes path i when they are at an energy state

Symbol	Definition	Units
ϕ_t	The energy state of the agent at time t.	Cal
J	Basic energy requirement per tile, with no change in elevation.	$\frac{Cal}{\tau_l}$
ξ	Index of fatigue. Mathematically, a function of state and time.	No Units
Δh	Change in elevation.	τ_l
b	Weighted energy cost for inclines versus declines.	No Units
$z(\Delta h)$	Distance conversion to account for elevation.	Cal
$\tau_{j,k}$	Tile located at coordinate (j, k)	No Units
τ_l	Tile length.	Mi
d_i	Distance to destination from tile i without elevation change.	τ_l
$\mathscr{C}$	Energy cost to move in the direction of i	Cal
$\mathscr{C}'$	Energy cost of existing, even when standing still.	Cal
$\mathscr{D}_t$	Selected destination at time t.	(x, y)
$\mathscr{L}_{\phi,0}$	Initial location of agent in state ϕ.	(x, y)
$\phi_F(\phi, d_{i,0}, \mathscr{D}_t)$	Maximum remaining energy after travel from $d_{i,0}$ to $\mathscr{D}_t$.	Cal

Table 1: Index of variables and parameters for the Bellman Equation

ϕ is quantified as $\mathbb{E}(\phi, i)$, given as follows:

$$\mathbb{E}(\phi, i) = \phi - (Jd_i + \xi\mathscr{C}_i + \mathscr{C}') \tag{7}$$

Where Jd_i represents the total expected future cost of the journey from i without regard to elevation, ξ represents an index of fatigue, and $\mathscr{C}'$ represents the energy cost to exist. The term $\mathscr{C}_i$ represents the energy cost to move from one tile to another, dependent on elevation and distance, and is given by:

$$\mathscr{C}_i = z(\Delta h) \tag{8}$$

The index of fatigue ξ in general, is a function of energy state and time, but is here left as a constant, i.e.:

$$\xi(t, \phi) = 1 \tag{9}$$

The cost due to elevation, $z(\Delta h)$, is a function of the change in elevation that has units of kilo-calories. We define the function as follows:

$$z(\Delta h) = \begin{cases} J + 0.3845(mg)(\tau_l)(\Delta h); \Delta h > 0 \\ J - 0.3845(mgb)(\tau_l)(\Delta h); \Delta h \leq 0 \end{cases} \tag{10}$$

Where m is mass (in kilograms), g is the gravity constant (in meters per seconds squared), τ_l is the tile length, and J is the energy per tile ratio. The constant 0.3845 is a ratio of kilo-calories per newton-mile. Δh is measured in tile lengths. The parameter b is the "downhill discount", which represents a lost individuals willingness to speed up when traveling downhill.

Define $\phi_F(\phi, \mathscr{L}_{\phi,0}, \mathscr{D})$ as the maximum amount of energy remaining after reaching the destination D_t, starting at $d_{\mathscr{L}_{\phi},0}$. Define the standard Bellman Equation for the Missing Person Model, as follows:

$$\phi_F(\phi, \mathscr{L}_{\phi,0}, \mathscr{D}) = \underset{i}{max}\{\phi_F(\phi_i, d_{\phi_i,0}, \mathscr{D})\} \tag{11}$$

Normally, the Bellman Equation is used to find the optimal strategy [1]. In our case, however, we use the equations to determine the energy remaining for each possible path taken. This information is then used to assign a probability that the individual any one path, i. Stated differently, the expected energy remaining is the set of each remaining energy quantities for each possible path i. We express this mathematically as follows:

$$\phi_F(\phi, \mathscr{L}_{\phi,0}, \mathscr{D}) = \{\phi_{F,i}(\phi_i, d_{\phi_i,0}, \mathscr{D});\}_{i\in\{1,\ldots,4\}} \tag{12}$$

Since our metric (Manhattan Distance) limits us to 4 different directions, we restrict i so that $i \in \{1,2,3,4\}$. Using the coupled DTA and Bellman decision-making processes, we can generate a "probability cloud" based on the individual probabilities of agent-occupancy for each tile. We now define the term fomally, as follows: Let X be the set of all tiles that make up the terrain map, and let τ denote an arbitrary tile. Then, $\forall \tau \in X$ and $\forall t \in (t_0, T)$, there is a probability $P_t(\tau)$ of the hiker occupying that tile at time t. The probability cloud is defined as follows: Let $P_t(\tau_{j,k})$ be the probability of a hiker occupying tile (j,k) at time t. Then the probability cloud $\mathscr{P}_t(X)$ at time t is defined as the set of all tiles and their associated probabilities, such that $P_t(\tau_{j,k}) > 0$. Expressed mathematically:

$$\mathscr{P}_t(X) = \{(\tau_{j,k}, P_t(\tau_{j,k}))\}_{(j,k)\in I\times I}; P_t(\tau_{j,k}) > 0 \tag{13}$$

Graphics

The simulation produces a time series of probability clouds images that are then combined with the topographic map to produce an animation of possible lost individual's location at each time. Example images can be found in Figure 4 and Figure 5. One observation is that, at any given time $t > 0$, there are five possible tiles for the hiker to go. The Bellman equation or Kitchens equation governs movement at every time step in accordance with the current strategy. Furthermore, specific probabilities for each time step can be inferred from this equation. We relate a more optimal (that is, lower expected cost) path with a higher probability, and rank the possible paths in this way.

Analysis

Parameter Estimates

Parameter estimates were made based off of the available literature from various sources. The cost of living $\mathscr{C}'$ and unit energy per tile J were calculated using established formulas from the health sciences [14, 15].

Specifically, we used the following formula to determine the energy cost of living, as follows:

$$\mathscr{C}' = \frac{B}{24} * 0.85 \tag{14}$$

Furthermore, the formula for determining the basic energy cost per tile is as follows:

$$J = \frac{Bqt}{24} \tag{15}$$

Both of these equations contain a parameter called the Basic Metabolic Rate (BMR), denoted here as B, which depends on the individual's weight, height, age, and sex. To calculate BMR, we took the average of a male hiker and a female hiker. We assumed the male hiker to be 35 years old, 6 feet tall, and 150 pounds without load. The female hiker was assumed to be the same age, 140 pounds without load, and 5 feet, 4 inches tall. Their independent BMR was calculated to be 1680.29 and 1433.2 respectively. Using the formula, the average cost of living between the male and female archetypes was calculated to be 55.1347 Cal. per hour, which was rounded down to 55 Cal. per hour for convenience. Another parameter required to calculate J was the Metabolic Equivalents (MET's), denoted here as q. This depends on time and the amount of work done. For a hiker moving at a rate of 2 miles an hour, the MET was calculated to be 1.625 Cal, which was used for both sexes [23]. Finally, we set $t = 0.1$ hours, to match our time-step. From this calculation, we derived the approximate basic rate $J = 11.5$ kcals.

Parameter	Description	Point estimate
J	Basic Energy Transfer Rate Per Tile	11.50 $\frac{C}{\tau_i}$
τ_l	Tile length	0.200 mi
t	Length of time-step	0.100 hr
ξ	Fatigue Index	1.000
m	Mass	80.00 kg
g	Gravity Acceleration	9.810 $\frac{m}{s^2}$
b	Downhill Discount	0.950
$\mathscr{C}'$	Energy Cost of Living	55.00 Cal
μ	Confidence Rate of Change	-0.340 $\frac{M}{t}$
k_e	Scaling constant (Kitchens Eqn.)	100.0
k_d	Scaling constant (Kitchens Eqn.)	1.000
Boundary Condition	**Description**	**Initial Value**
ϕ_0	Energy	3000 Cal
$\mathscr{L}_0$	Location	$x \in [0, 1000], y \in [0, 1000]$
M	Mental state	100.0

Table 2: Table of Parameter Estimates and Initial Conditions

Numerical Analysis

In this section, we performed a scenario analysis which consists of varying conditions that might occur in a real-world situation based on reported case studies and available data [16, 18]. We also compared the two heuristic path finding algorithms. Ultimately, we identify certain areas as bottlenecks where a lost hiker is likely to be.

Output of Algorithm

The model is designed to output a probability cloud for each time step by utilizing either the Kitchens Equation the Bellman Equation. As time approaches the finite horizon T, which has been set to 100 time steps, the behavior of the probability depends on the location of landmarks, and the strategy chosen. If a landmark exists within the map, and the strategy selected accommodates landmark navigation, the probability cloud will converge around the landmark as time increases, until the occupancy threshold is met (if confidence category is high or medium). Figure 4 offers an example of how the probability cloud may be superimposed on a terrain map, and how the availability of landmarks results in convergence around a known point. Figure 5 shows a simulation in which you can see the probability cloud split between two different possible paths.

Moreover, we calculated the probabilities of occupying each tile, and noted the tiles with relatively high probabilities. We used this information to identify "bottlenecks" on the map. These tiles will the greatest likelihood of a wandering hiker passing through at time some time t. All of this information may have relevance to SAR operations, including prioritizing these regions of search as well as setting up barricades to prevent the hiker's travel through these regions.

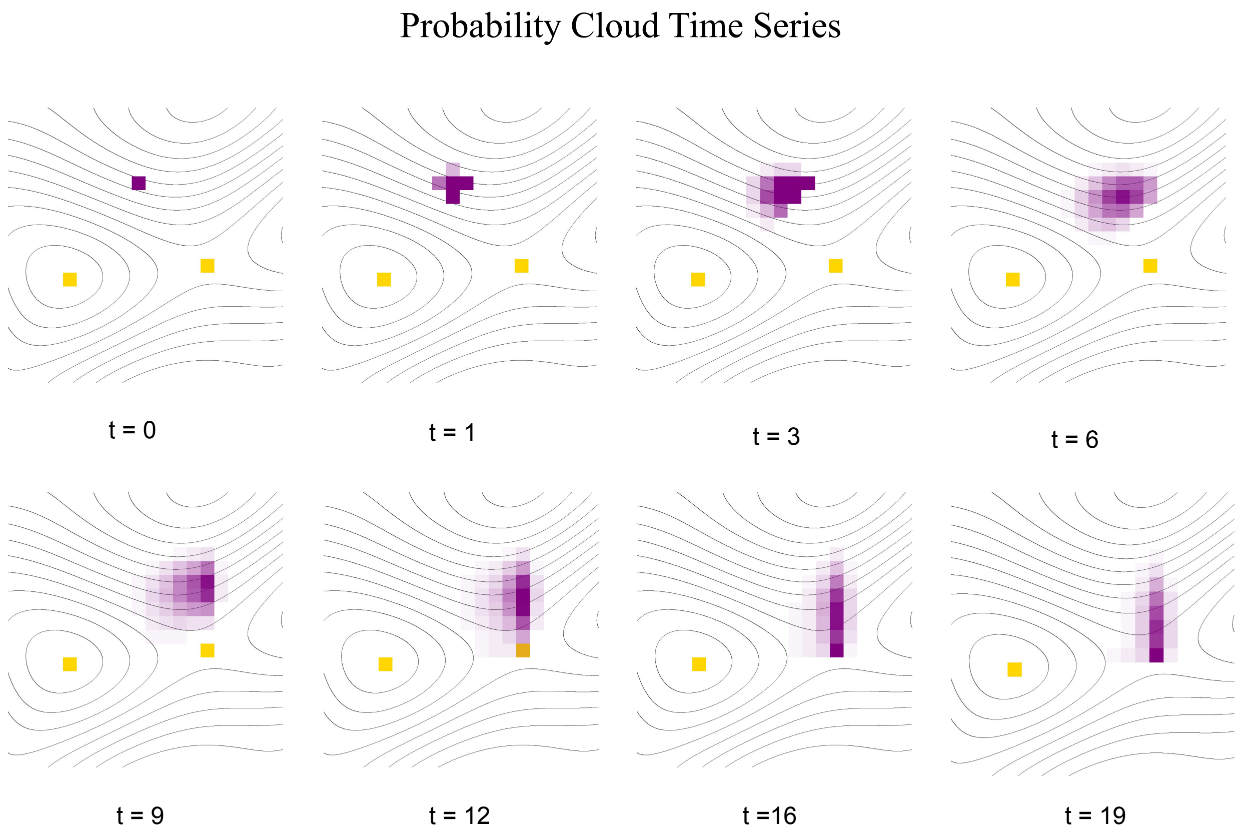


Figure 4: Probability cloud of an individual reaching their destination point in time t using the Kitchens equation model. The two yellow squares represent possible goals of a peak and a saddle. The purple square at $t = 0$ represents where the hiker was last seen. As time progresses, the number of purple squares increase, as there are many possible locations for the hiker to be. More opaque squares demonstrate a higher probability that the hiker is located there, whereas more transparent squares represent a lower probability of the hiker being located there. As time increases, the highest probability path becomes clearer. There is a chance that the hiker will begin heading toward one goal (as seen at $t = 6$ and $t = 9$), but then heads to the closer goal, (as seen at the start of $t = 12$).

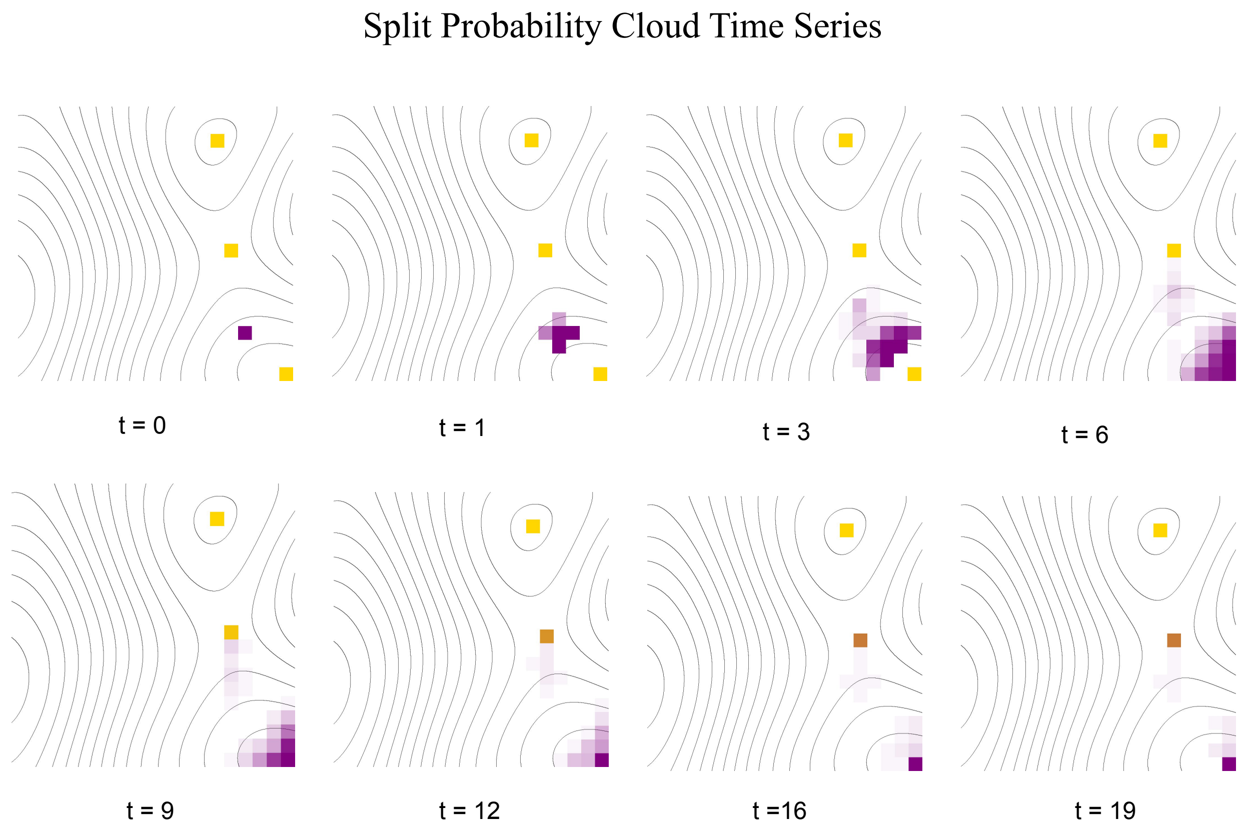


Figure 5: Probability cloud splits destination point at time $t = 3$ under the Kitchens equation model. The three yellow squares represent possible goals, i.e. a peak or a saddle. The purple square at $t = 0$ represents the probability cloud of the hiker. A darker shade of purple corresponds to a higher probable location of the hiker. As time progresses, the number of purple squares split into two probabilistic paths for two different goals. This is because the hiker is equidistant to both goals.

Scenario Analysis

We test our model using three trial simulation. These cases have been selected because the human response to the initial conditions and geographic context is well understood. This provides us a way to validate our model.

1. An individual who is unfamiliar (low confidence) with their surroundings and low in energy becomes lost in a valley between mountain peaks. We expect that this individual will remain at their current position waiting for rescue.

2. A physically fit and highly confident individual becomes lost. We expect that the individual will move

from landmark to landmark until they either run out of energy or exhaust all available landmarks.

3. An individual with high energy and confidence in an environment with low visibility (no landmarks), will most likely use the strategy "directional traveling". We expect the probability cloud to generally expand in all directions, slightly favoring constant elevation.

We discuss these in the results section of this paper.

Parameter Variation

The following parameters and initial conditions were given special attention, as these were expected to have the strongest impact on the resulting probability cloud. For example, the initial location directly impacts the destination that the DTA will select, and thus our model could have vastly different outcomes depending on the initial location. Likewise, the degree of roughness of the terrain will change the amount of energy needed to move between tiles, and is thus also expected to change the outcome of our simulation (see Scenario 3 in Results section). It is important to understand the effects that these parameters and initial conditions have on the overall output of the model in order to analyze the influence of initial position on the resulting path that the hiker takes.

There are many other factors including initial energy, initial confidence, and terrain roughness which may also have a large influence on the paths produce. Due to time constraints, these parameters were not tested in this study. Brief insight into how these parameters might affect the paths can be seen in the scenario analysis, though more formal testing is needed. Excluded from this list are the cost of movement, and energy cost per tile. Neither of these are parameters, because they are based on physical values, and do not change once the map scale is set. Since energy cost per tile is not included, the fatigue index can also be excluded. Even though changes to the fatigue index, ξ would indicate a higher or lower energy cost per tile, (since it is a factor of energy cost), ξ is left as a constant, so it is also omitted.

The energy state ϕ can range between 0 and 3000 Cal. For the purpose of our model, once a lost hiker's energy amount reaches zero they are unable to move. Higher energy correlates to a greater potential travel distance, and a lower energy correlates to a more limited range of travel. Elevation Δh is a simple linear distance that must be non-negative. all else being equal. Both outcomes are discussed in detail under the results section.

Comparing Heuristics

In this section, we simulate model with Kitchens and Bellman equations independently, in which both simulations start from the same initial value ans use the same map. Kitchens equation is based in altitude and distance that change the probability cloud while the hiker is walking. Bellman equation takes in consideration other parameters that affect the energy and try to select the best path based in less cost for the individual.

Bottleneck Analysis

We performed a bottleneck analysis, overlaying many possible paths, to generate the most likely locations that an individual may be at a given time step. After running the simulation for many initial states other, we identified the bottlenecks, or regions that many paths share. These regions represent areas that may be of particular importance to SAR operations, as they may provide locations for more efficient search.

Results

Scenario-Specific Simulation

We described three test-cases in the analysis section of our present work. These scenarios were chosen as they represent an isolated time period in a larger lost person case. This allowed us to predict the hikers movements to confirm that the model was acting in the expected way. The results of those are laid out in detail below.

First Scenario

A simulation was conducted for the scenario in which an individual who has low confidence (as a result of unfamiliarity with their surroundings) and low energy. Figure 6 summarize the model output with confidence at 30 units (confidence category: low) and energy at 900 Cals (energy category: low). As expected, the individual did not move, and the resulting probability cloud was focused around the initial position for the entire time horizon. Though a predictably uninteresting scenario, this shows that as an hiker's energy decreases, the number of possible strategies that can be utilized by the hiker also decreases. Eventually, the only available strategy is to stay in place, as shown in the figures below.

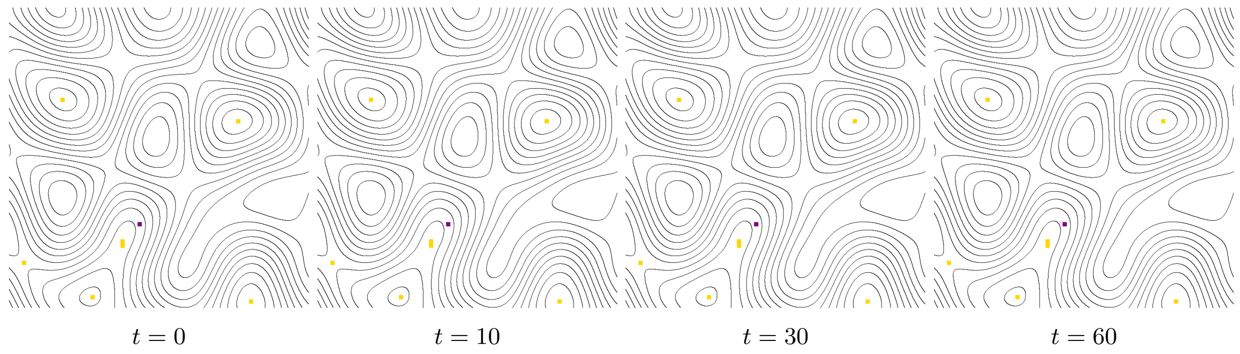


Figure 6: An individual with low energy and low confidence will adopt Stay Put strategy. Their probability cloud remain constant in the same position

Second Scenario

A simulation scenario was conducted for a person with high energy and high confidence from an arbitrary location. Figure 7 summarizes the model output with confidence at 100 and energy at 3000 Cals. The individual is moving from landmark to landmark in the different time steps.

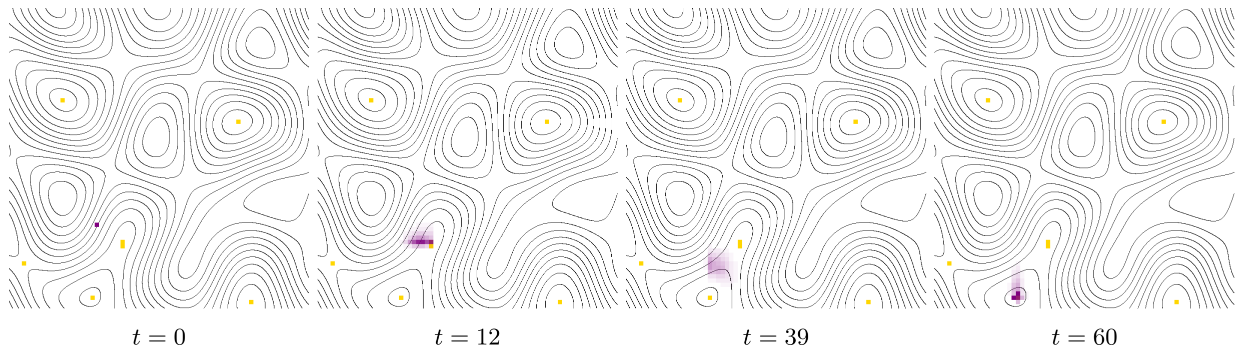


Figure 7: An individual with high energy and high confidence will go from landmark to landmark

Third Scenario

A simulation was conducted for the scenario in which a hiker has high energy, high confidence, and no variation in terrain. As a result, there are also no landmarks. Figure 8 summarize the model output with confidence at 1000 and energy at 3000 Cals. Being that the terrain bears no influence on the individual, there was no destination for the DTA to select. Since the hiker is in a high-energy and high-confidence state, they adopted directional traveling, and the probability cloud spreads out in all directions evenly, as a result. We noticed also that the direction of the cloud tended to favor the diagonal axis. This is a result of using the Manhattan distance. Individuals that adopt the directional traveling are trying to maximize their distance from the current location, and a slight advantage is afforded to diagonal travel, compared to linear travel. This should be corrected in future iterations.

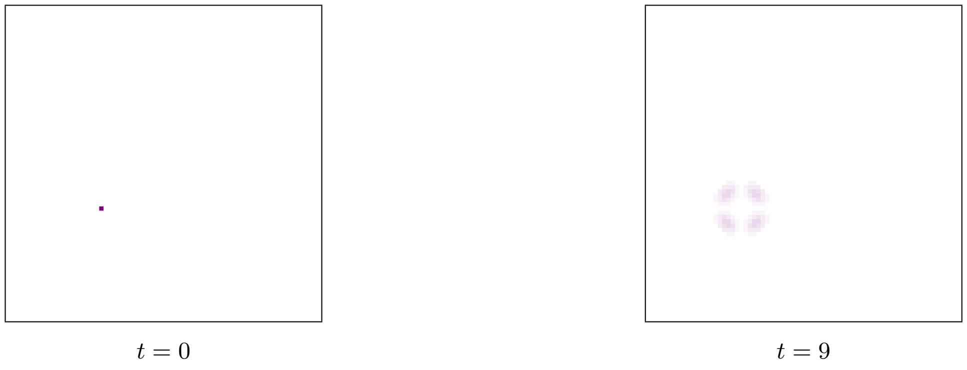
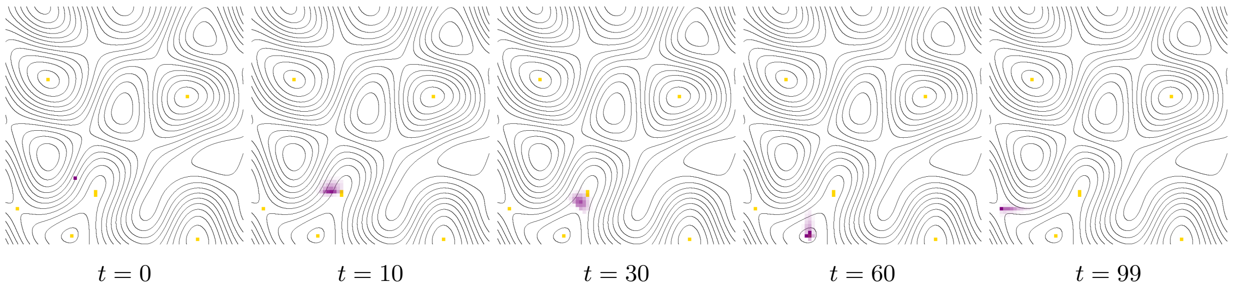


Figure 8: An individual with high energy and high confidence will adopt directional travel when there are no visible landmarks

Comparing Heuristic Algorithms

We ran the same simulation with the same initial values and terrain map, changing only the heuristic applied. The results are summarized below. Figure 9 show a side-by-side comparison of a single path. The single hiker path shows how the algorithms begin to diverge after the hikers reach their initial destination. The Bellman equation simulation reached its destination several time steps faster than the Kitchens equation simulation. This is most likely due to the Bellman equation punishing inactivity more severely, forcing movement of the hiker. Figures 11 and 12 shows the Kitchens and Bellman path overlays for comparison.

(a) Kitchens



(b) Bellman

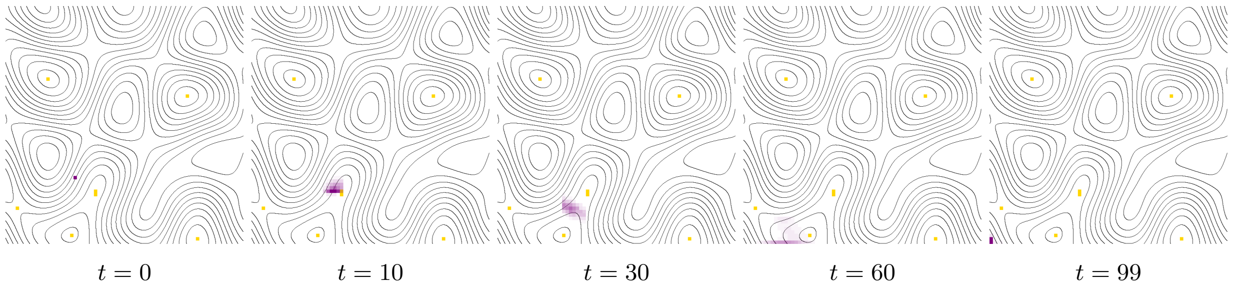


Figure 9: Comparison Heuristic Algorithms in different times with the same initial position

Hiker Movement From Unknown Initial Location

In many SAR operations, the exact location where an individual becomes lost. Many simulations were run across the map row-by-row. Each simulation in the "sweep" was given an identical set of initial conditions, with the exception of the start location. To determine the location of any high-probability paths or tiles common across different starting locations, we overlaid the outputs of these simulations generating a single map of all the paths, with darker pixels representing areas with the most number of path crossovers. These crossovers, or bottlenecks, may be regions of high importance to SAR personnel, as they represent places of travel that are consistent across many different starting locations, and so can be search more effectively than searching each path individually. At these locations, personnel may look for clues of recent disturbance and/or set up watch to prevent further movement through this region (similarly to methods use to block movement at bridges).

To reduce artificial effects that occur around the edges of the map, we selected start positions within a 35 pixel x 35 pixel region surrounding the center of the map, with buffer of at least 17 pixels on either side. Start locations were separated by five pixels along both axes. Figure 10 provides a visualization of the initial setup.

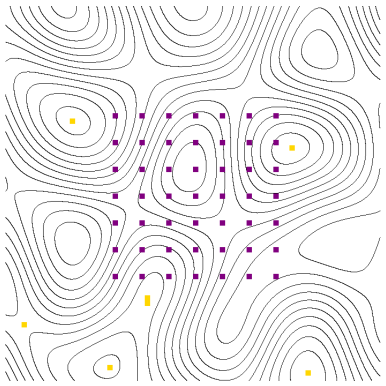


Figure 10: Start locations of simulated hikers. Purple pixels represent the hikers at time step 0. Gold pixels represent landmarks (peaks and saddles). Contour lines occur every 100ft.

The simulations progressed over 100 time steps. We recorded the distribution of hiker location probability clouds for each time step. The opacity of each pixel corresponds to the probability of hiker being located at that pixel at that time. Darker pixels correspond with higher sums of probabilities. All hikers exhibit view-enhancing behavior (landmark), increasing elevation while seeking both peaks and/or saddles. Though hikers all start with the the same energy levels, simulated hikers whose path takes them over more energy intensive land will lose energy faster than their counterparts. This can lead to selected strategies differing between simulated hikers. After the simulation completed, we overlaid all of the simulated movements.

We found that under the Kitchens Equation hikers were more willing to spend energy than under the Bellman. Further analysis showed that the energy difference resulted in a change in strategies chosen after a certain time. The end result was that, under the Kitchens equation, hikers tended to move from landmark to landmark, eventually congregating around one of the peaks or saddle points. In contrast, under the Bellman equation, hikers tended to the nearest landmark in a similar fashion to the Kitchens heuristic, but thereafter switched to directional traveling and spread out in all directions.

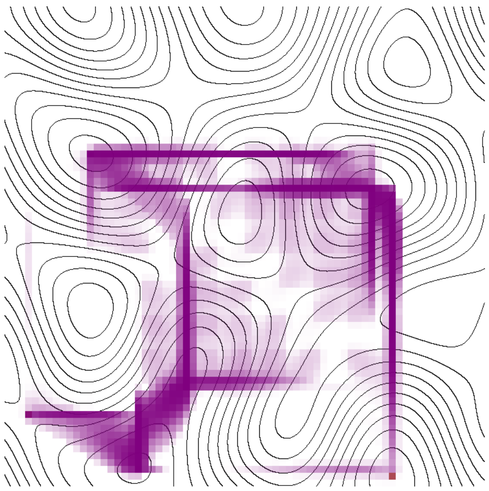


Figure 11: Overlay with Kitchens Algorithm: Purple pixels represent the location probability cloud of the hiker. Contour lines occur every 100ft.

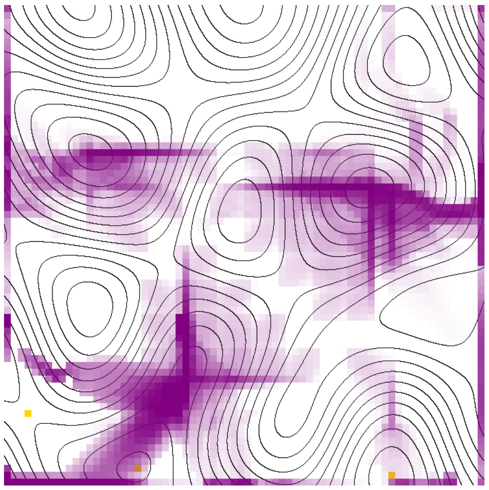


Figure 12: Overlay with Bellman Algorithm: Hiker's movements. Purple pixels represent the location probability cloud of the hiker. Contour lines occur every 100ft.

From Figures 11 and 12, we can see that simulated paths did not cover the terrain evenly. Areas with the greatest number of crossovers represent bottleneck locations where SAR individuals should be sent to look for any signs of recent disturbance and block future lost hiker travel. Bottlenecks are areas with a high probability of finding the hiker. Areas surrounding landmarks, such as peaks and saddles received the greatest number of path crossovers. As these landmarks are topographically unique locations, it is unsurprising that many individuals congregated in these regions. Without knowing the start position of the lost hiker, there are also regions of the map that can be ruled out as potential search locations based on having little to no probability of hiker presence at any time step throughout the simulation. These regions correspond with low elevation areas that are not located between landmarks of interest.

Discussion

In order to predict the movement of a missing hiker as determined by human behavior and the geography in the wilderness, we have analyzed our model, considered several test cases, validated our test results, compared heuristic algorithms, and tested various initial conditions to determine the viability of our model. Herein, we discuss the results, outline the limitations of our framework, and discuss future improvements.

Doherty, *et al.* described the need for behavioral analysis to be included in future missing persons models [4].

This was achieved herein through the inclusion of a DTA that mimics human decision-making. We found our test scenarios along with the bellman's matched our expectations within reason. Figure 6 demonstrate a hiker with low confidence and low energy will stay put. Figure 7 showcases clear landmark navigation for a hiker with high energy and high confidence. Comparable models analyzed in [6] demonstrated that between three dissimilar models, all four identified peaks and high-points as high-probability areas for locating a missing hiker. This closely matches our results. Finally, Figure 8 display directional traveling for a high energy/high confidence hiker with no landmarks in sight. This confirms the basic functionality of our model matches documented behavioral patterns for well understood scenarios, which implies that our model can be applied to learn more about how movement patterns change in less obvious scenarios.

The travel patterns are elucidated in Figures 11 and 12. While the complete set of paths covers a wide area of the map, peaks and saddle points did indeed attract the most traffic. We expected little variability between starting locations, which was confirmed by the simulation. This information provides the highest probability locations of where a hiker may be at a given time. After comparing heuristic algorithms, we found that both the Bellman and the Kitchens path finding equations took similar paths early on, and began to diverge after reaching the first destination. As noted previously, under the Kitchen's equation, hikers maintain their landmark-navigation strategy, which results in the probability clouds centering around one of the peaks. For a certain class of hiker, this is not an unreasonable outcome, and can be rather favorable for SAR teams. The Kitchens equation does not conserve energy to the degree that the Bellman equation does, which might represent how a young hiker with a lot of energy and physical conditioning might wander. On the other hand, the Bellman equation predicted a much worse scenario for the missing individual, in which, after reaching the first peak, the hiker adopted a directional traveling strategy, and the probability cloud spread out to the edges of the map. The Bellman equation is likely the more realistic process of the two.

From this comparison, we conclude that SAR teams have the best chance of finding the missing hiker if they intercept them before they reach their first destination or before they travel to their first view location, when the probability cloud is most densely centered. After that point, the probability cloud is evenly distributed over a large part of the map, and the POA begins to mirror the bullseye shapes that are well represented in literature. The results from [12] corroborated our results, in that less mobile hikers were more accurately described by both models, and that both worked best within a short time horizon. However, we found additionally that this divergence occurs after the hiker reaches a first destination. At that point, they continue along a linear path away from the original destination, which suggests a possible search trajectory can be predicted if additional information about the hiker's previous locations can be obtained.

Limitations of the Model

We take the opportunity here to outline the limitations and future work of the study. Many of these limitations are the result of time constraints, and offer opportunities for improvement in the future. The following are limitations with regard to terrain factors:

- Currently, the terrain generation does not account for bodies of water, such as lakes, rivers, or streams. In reality, this plays an important role in both path finding and decision making for the individual. Castle and Sava, *et al.* both illuminated the need for additional geographic features to be taken into consideration, most notably water features [3, 21]. Many individuals are found at or near water sources [16].

- As it stands, we are only considering "ideal wilderness," such as national wilderness reserves. In the future, the terrain generation should also include paths/trails that are likely to occur in national parks and hunting land.

- As of now, the only geographic information stored with each tile is elevation. In reality, other factors play an important role, such as forest density, canopy cover, and directional markings (including signs or blazes[12]). Future iterations should encode this information within the tile to be considered by the decision-making process.

- We assume that weather is constant throughout out model. Clearly, this is not the case in many SAR missions. Eventually, the model should be able to factor in dynamic environmental factors, such as weather or natural disasters.

- Our model does not take land cover or restricted vision into consideration, which is known to effect navigation skills.

Regarding the individual, we list the following limitations and future work:

- During the simulation, energy decreases relative to the sum of the movement cost and the living cost. Living costs are consistent across all regions of the map. Future studies may attempt to quantify how exposure to weather may effect energy costs for being at various locations.

[12]A strike of paint on trees that detail trails in national parks.

- Two improvements can be made to the movement costs of the simulated hikers decisions: 1) Movement speeds are constant, irrespective of terrain and 2) Movement costs are currently based solely on the work put in to change elevation. Land cover has been shown to have an effect on movement speeds and energy required to travel through a region [4].
- Lost individuals may prefer sheltered terrain, such as valleys, rather than peaks and high elevation areas to avoid deadly exposure [24]. Currently, our model does not consider this preference. Going forward, each tile should have an attribute that describes land coverage, and the DTA should seek to minimize exposure. Alternatively, the heuristic path finding algorithm could be weighted in favor of land cover.
- Our current model assumes a typical hiker. Future iterations should divide hikers into classes with different attributes and initial conditions, and analysis on the different classes should be conducted.

Finally, a we list possible avenues of future investigation, as follows:

- Compare model output to existing GIS maps and missing persons data [19].
- Consider different heuristic algorithms that more accurately reflect lost human behavior.
- Analyze our framework as applied to other contexts, such as in urban environments, or after natural disasters.

Conclusions

Our research evaluates the influence of terrain in combination with lost person behavior. This is used to create a mechanistic model that predicts a lost hiker's movement in a land-wilderness context. Our model is unique from others because it incorporates both geographic information and behavioral analysis. We constructed a finite time horizon Markov Decision Process with two decision algorithms, one for selecting a destination on the map, and the other for selecting a path to that destination. The framework was designed so that different decision engines could be interchanged. Thereafter, two different heuristics were constructed, and simulations were run with both algorithms.

We found that the hikers require more time as compared with the Bellman equation to reach the goal when the initial energy is high. The heuristic chosen did change the outcome of the model significantly, because the disparity between energy states causes a shift in strategy. We found that conservative heuristics such as the Bellman equation resulted in a wider probability cloud than the less energy efficient Kitchens algorithm. We found that after reaching an initial destination, usually a nearby peak or saddle, the probability cloud begins to increase drastically in area. However, there was some artifacting inherent to our model as a result of using the von Neumman neighborhood and Manhattan distance metric. If additional signs that the hiker can be found in a particular direction are available, it may be possible to predict their trajectory based off of their initial location and first destination.

We encourage interested researchers to investigate better heuristics that can be incorporated to the present framework, compatibility with additional terrain features, most importantly bodies of water and existing trails, as well as adding terrain factors that inhibit the field of vision of the hiker into the model.

As GIS and computation methods become more available to SAR teams, further research into the interactions between human behavior and the environmental context will be crucial to improving SAR operations as a whole. We suggest a deeper investigation into applications of Artificial Intelligence as it can be applied to predicting missing persons behavior, as a way to improve SAR protocols, and save the lives of future hikers.

Acknowledgements

We would like to thank Dr. Carlos Castillo-Chavez, Founding and Co-Director of the Mathematical and Theoretical Biology Institute (MTBI), for giving us the opportunity to participate in this research program. We would also like to thank Co-Director Dr. Anuj Mubayi as well as Coordinator Ms. Rebecca Perlin and Management Intern Ms. Sabrina Avila for their efforts in planning and executing the day to day activities of MTBI. We also want to give special thanks to Dr. Paul Ustach, Matthew Toro, Jill Sherwood, and Dr. Leon Arriola. This research was conducted as part of 2019 MTBI at the Simon A. Levin Mathematical, Computational and Modeling Sciences Center (MCMSC) at Arizona State University (ASU). This project has been partially supported by grants from the National Science Foundation (NSF Grant MPS-DMS-1263374 and NSF Grant DMS1757968), the National Security Agency (NSA Grant H98230-J8-1-0005), the Office of the President of ASU, and the Office of the Provost of ASU.

References

[1] Richard Bellman, *Dynamic programming*, Dover Publications, Mineola, N.Y, 2003.

[2] CHARLES W. BRYANT, *Who pays for search and rescue operations?*, 2010.

[3] Timothy S Castle, *Coordinated inland area search and rescue (sar) planning and execution tool.*, NAVAL POSTGRADUATE SCHOOL MONTEREY CA, 1998.

[4] Paul J Doherty, Quinghua Guo, Jared Doke, and Don Ferguson, *An analysis of probability of area techniques for missing persons in yosemite national park*, Applied Geography **47** (2014), 99–110.

[5] Jared Doke, *Analysis of search incidents and lost person behavior in yosemite national park*, Ph.D. Thesis, 2012.

[6] Sebastian Drexel, Susanne Zimmermann-Janschitz, and Robert J Koester, *Network analysis for search areas in wisar operations*, International Journal of Emergency Services **7** (2018), no. 3, 192–202.

[7] Paul A Dudchenko, *Why people get lost*, Oxford University Press, 2010.

[8] Eduardo Feo, Luca Gambardella, and Gianni A Di Caro, *Search and rescue using mixed swarms of heterogeneous agents: Modeling, simulation, and planning*, Technical Report IDSIA-05, 2012.

[9] Michael Patrick Ghiglieri and Charles R Farabee, *Off the wall: death in yosemite: gripping accounts of all known fatal mishaps in america's first protected land of scenic wonders*, Puma Pr, 2007.

[10] KA Hill, *The psychology of lost*, Lost person behavior (1998), 116.

[11] Kenneth Anthony Hill, *Cognition in the woods: Biases in probability judgments by search and rescue planners* (2012).

[12] Mark Powell Johnson, *Evaluating the utility of a geographic information systems-based mobility model in search and rescue operations*, Ph.D. Thesis, 2016.

[13] Lisa Jones Christensen and Scott C Hammond, *Lost (but not missing) at work: Organizational lostness as an employee response to change*, Journal of Management Inquiry **24** (2015), no. 4, 405–418.

[14] Aaron Kandola, *How many calories do you burn by walking?* (2019).

[15] ______, *How to calculate the calories a person burns while sleeping* (2019).

[16] Robert J Koester, *Lost person behavior: A search and rescue*, dbS Productions LLC, 2008.

[17] ______, *Lost person behavior: Instructor activity guide*, dbS Productions LLC, 2010.

[18] ______, *International search & rescue incident database (isrid)* (2011).

[19] ______, *International search & rescue incident database (isrid)* (2019).

[20] Marc Mangel and Colin Whitcomb Clark, *Dynamic modeling in behavioral ecology*, Princeton University Press, 1988.

[21] Elena Sava, Charles Twardy, Robert Koester, and Mukul Sonwalkar, *Evaluating lost person behavior models*, Transactions in GIS **20** (2016), no. 1, 38–53.

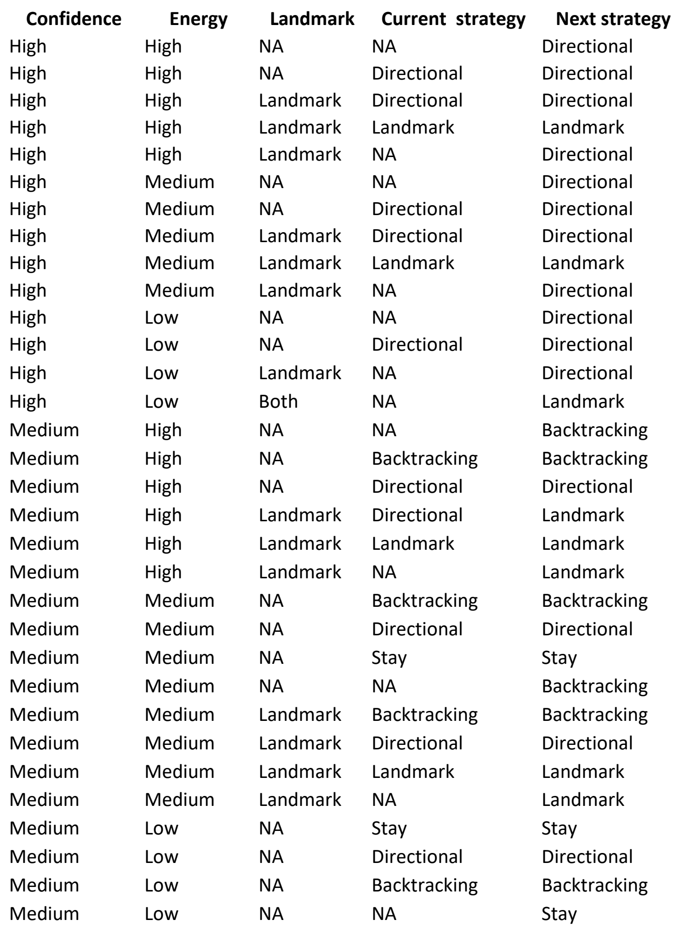
[22] National Search and Rescue Council, *Lost person behavior* (2017).

[23] Northeast Wilderness Search and Rescue, *Lost person behavior a look at the generalities of lost person behavior* (2017).

[24] Nate Seltenrich, *Between extremes: Health effects of heat and cold*, National Institute of Environmental Health Sciences, 2015.

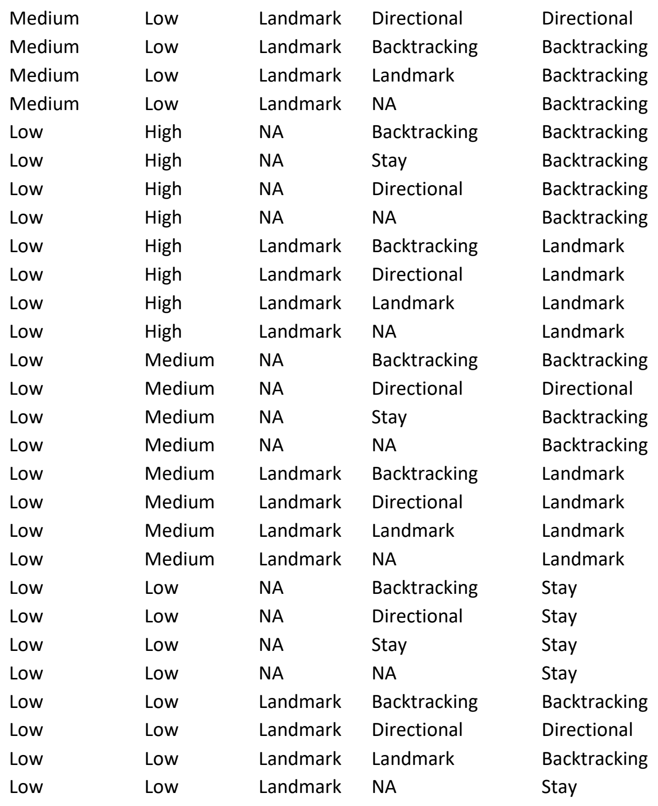
A Appendix

A.1 Table of Decision Tree



Confidence	Energy	Landmark	Current strategy	Next strategy
High	High	NA	NA	Directional
High	High	NA	Directional	Directional
High	High	Landmark	Directional	Directional
High	High	Landmark	Landmark	Landmark
High	High	Landmark	NA	Directional
High	Medium	NA	NA	Directional
High	Medium	NA	Directional	Directional
High	Medium	Landmark	Directional	Directional
High	Medium	Landmark	Landmark	Landmark
High	Medium	Landmark	NA	Directional
High	Low	NA	NA	Directional
High	Low	NA	Directional	Directional
High	Low	Landmark	NA	Directional
High	Low	Both	NA	Landmark
Medium	High	NA	NA	Backtracking
Medium	High	NA	Backtracking	Backtracking
Medium	High	NA	Directional	Directional
Medium	High	Landmark	Directional	Landmark
Medium	High	Landmark	Landmark	Landmark
Medium	High	Landmark	NA	Landmark
Medium	Medium	NA	Backtracking	Backtracking
Medium	Medium	NA	Directional	Directional
Medium	Medium	NA	Stay	Stay
Medium	Medium	NA	NA	Backtracking
Medium	Medium	Landmark	Backtracking	Backtracking
Medium	Medium	Landmark	Directional	Directional
Medium	Medium	Landmark	Landmark	Landmark
Medium	Medium	Landmark	NA	Landmark
Medium	Low	NA	Stay	Stay
Medium	Low	NA	Directional	Directional
Medium	Low	NA	Backtracking	Backtracking
Medium	Low	NA	NA	Stay

Figure 13: Decision tree table with the possible combination of states and outputs that a hiker could experience.[Part 1]



Medium	Low	Landmark	Directional	Directional
Medium	Low	Landmark	Backtracking	Backtracking
Medium	Low	Landmark	Landmark	Backtracking
Medium	Low	Landmark	NA	Backtracking
Low	High	NA	Backtracking	Backtracking
Low	High	NA	Stay	Backtracking
Low	High	NA	Directional	Backtracking
Low	High	NA	NA	Backtracking
Low	High	Landmark	Backtracking	Landmark
Low	High	Landmark	Directional	Landmark
Low	High	Landmark	Landmark	Landmark
Low	High	Landmark	NA	Landmark
Low	Medium	NA	Backtracking	Backtracking
Low	Medium	NA	Directional	Directional
Low	Medium	NA	Stay	Backtracking
Low	Medium	NA	NA	Backtracking
Low	Medium	Landmark	Backtracking	Landmark
Low	Medium	Landmark	Directional	Landmark
Low	Medium	Landmark	Landmark	Landmark
Low	Medium	Landmark	NA	Landmark
Low	Low	NA	Backtracking	Stay
Low	Low	NA	Directional	Stay
Low	Low	NA	Stay	Stay
Low	Low	NA	NA	Stay
Low	Low	Landmark	Backtracking	Backtracking
Low	Low	Landmark	Directional	Directional
Low	Low	Landmark	Landmark	Backtracking
Low	Low	Landmark	NA	Stay

Figure 14: Continuation of Decision tree table with the possible combination of states and outputs that a hiker could experience [Part 2]

A.2 Python Code

```
import opensimplex, random
from PIL import Image
import math
from datetime import datetime
import imageio
from colour import Color
import pandas as pd
import os, shutil


def rescaleValuesToNewRange(n, start1, stop1, start2, stop2):
# This line defines a function "p5map" (replicating the map function from p5.js)
    return ((n-start1)/(stop1-start1))*(stop2-start2)+start2
    # values that range from value x1->x2 now range from x3->x4

def printMapToPNG(map, colorRange, valueRange, path, sizeScale=1):
# This function displays the a Matrix Object as a .png
    if colorRange[0] == "clear":
    # Creates a transparency gradient from clear to color
        colors = [Color(colorRange[1]) for i in range(256)]
        alpha = [i*10 for i in range(256)]
    elif colorRange[1] == "clear":
    # Creates a transparency gradient from color to clear
        colors = [Color(colorRange[0]) for i in range(256)]
        alpha = [i*10 for i in range(256)]
    else:
        startColor = Color(colorRange[0])
        endColor = Color(colorRange[1])
        colors = list(startColor.range_to(endColor,256))
        # Creates a color gradient between two colors
        alpha = [255 for i in range(256)]
    size = len(map)
    out = Image.new(mode='RGBA', size=(size * sizeScale, size * sizeScale))
    # Opens a new image (using PIL module)
    for row in range(size):
        for column in range(size):
            for rowBuffer in range(sizeScale):
                for columnBuffer in range(sizeScale):
                    index = int(rescaleValuesToNewRange(map[row][column], valueRange[0],
                    valueRange[1], 0, 255))
                    color = []
                    for value in colors[index].rgb:
                        color.append(int(value*255))
                    out.putpixel(xy=(column*sizeScale+rowBuffer,row*sizeScale+columnBuffer),
                    value=(color[0],color[1],color[2],alpha[index]))
    out.save(path)
```

```
def distanceCalculator(start, stop):  # Function which calls a general distance calculator
    return manhattanDistanceCalculator(start=start, stop=stop)
    # Calls manhattanDistanceCalculator

def manhattanDistanceCalculator(start, stop): # General manhattan distance calculator function
    dx = abs(start[0] - stop[0])
    # Somewhat weirdly defined because of the way locations in the matrix are defined,
    dx is how far left or right the goal is
    dy = abs(start[1] - stop[1])
    # Somewhat weirdly defined because of the way locations in the matrix are defined,
    dy is how far up or down the goal is
    return dx+dy    # Returns the sum

def mean(numbers):  # Takes the mean of a list of values
    return float(sum(numbers)) / len(numbers)

def activationFunction(x, scale):   # Function to return the importance of this feature
    return math.exp(-abs(x) / scale)

def normalizeBySum(dict):     # Normalizes values in a dictionary to be between 0 and 1
    denominator = sum(dict.values())     # Sum all values
    normalizedValues = {}   # Empty dictionary that will be contained the normalized values
    if denominator == 0:     # If all of the values are 0, then all have equal probability
        for t in dict.keys():
            normalizedValues[t] = 1 / len(dict.keys())
    else:   # Else normalize
        for t in dict.keys():
            normalizedValues[t] = dict[t] / denominator
    return normalizedValues # Return the normalized values dictionary


class Region:

    def __init__(self,fullSize,workingSize,elevations=[],terrainSmoothness=10,terrainSeed=-1):
    # Initialize the Region Object
        if fullSize <= 1 or workingSize <= 1: # Doesn't let it run if sizes won't work
            print("ERROR: Map Matrix sizes must be greater than 1.")
            exit()
        self.fullSize = fullSize
        # All maps will be square, size refers to the number of pixels in a row of
        high resolution map size
        self.workingSize = workingSize    # Low resolution map size
        self.terrainSeed = terrainSeed
        if len(elevations) == 0:    # If previously defined elevations are not present
            if self.terrainSeed == 'NA':
                self.terrainSeed = random.randint(0,100000)
                # Generates a random number which will be used as the seed of
                the OpenSimplex random terrain generation
            self.highResElevationMap = self.generateElevationMap(terrainSmoothness=
            terrainSmoothness,seed=self.terrainSeed)
            # Generate the elevations using Simplex Noise
        else:
            self.highResElevationMap = elevations
            # Elevations are stored as a sympy.Matrix object to allow for matrix math
        self.lowResElevationMap = self.pixelateMap(highResMap=
        self.highResElevationMap,newSize=self.workingSize)
        # Takes high res map and creates low res version
        self.topoMap = self.generateTopoMap(highResMap=self.highResElevationMap)
        # Takes high res map and creates a topographic map
        if self.terrainSeed != -1:
            print('Generated␣Random␣Terrain␣Map␣(SEED␣' + str(self.terrainSeed) + '):
␣␣␣␣␣␣␣␣␣␣␣␣' + str(datetime.now() - startTime))
        else:
            print('Generated␣Terrain␣Map:␣' + str(datetime.now() - startTime))
        self.highResSlopeList= self.generateSlopeList(map=self.highResElevationMap)
        self.lowResSlopeList = self.generateSlopeList(map=self.lowResElevationMap)
        print('Generated␣Slopes␣Map:␣' + str(datetime.now() - startTime))
        self.highResLandmarks = self.identifyLandmarks(slopeList=self.highResSlopeList,
        size=self.fullSize)
        self.highResLandmarkMap = self.generateLandmarkMap(landmarks=self.highResLandmarks,
        size=len(self.highResElevationMap))
        self.lowResLandmarks, self.lowResLandmarkMap =
        self.generateLowResLandmarkMap(highResMap=self.highResLandmarkMap,newSize=self.workingSize)
        if len(self.lowResLandmarks) > 0:
            print('Generated␣Landmarks␣Map:␣' + str(datetime.now() - startTime))
        else:
            print('No␣Landmarks:␣' + str(datetime.now() - startTime))
```

```
def generateElevationMap(self,terrainSmoothness,seed):
# If elevations not present, generate the elevations (smoothness: how quickly
the elevations change, larger numbers are smoother)
    blackBox = opensimplex.OpenSimplex(seed=seed) # Determines the simplex seed
    noiseMap = []   # Stores elevation data, which is the turned into the sympy.Matrix
    for y in range(self.fullSize): # Loops over the rows
        line = []   # Stores the data for a single row
        for x in range(self.fullSize): # Loops over each state in a row
            line.append(rescaleValuesToNewRange(blackBox.noise2d(x=(x/terrainSmoothness)+50,
            y=(y/terrainSmoothness)+100), -1, 1, 0, 2000))   # Generates the values and adds them to l
        noiseMap.append(line)   # Adds the line to the map
    return noiseMap  # Stores noiseMap as a sympy.Matrix

def pixelateMap(self,highResMap,newSize):   # Function which creates a low res version of an image
    oldSize = len(highResMap)
    allPixelData = [[[] for i in range(newSize)] for j in range(newSize)] # Will hold all pixel info
    heightData = [[] for i in range(newSize)]  # Will hold the values for the pixelated map
    for workingRow in range(newSize): # Loops over each row in low res map
        for workingColumn in range(newSize):   # Loops over each column in low res map
            for highResRow in range(workingRow * (oldSize//newSize), (workingRow+1) * (oldSize//newSiz
            # Loops over each row in high res map
                for highResColumn in range(workingColumn * (oldSize//newSize), (workingColumn+1)
                * (oldSize//newSize)):
                # Loops over each column in high res map
                    allPixelData[workingRow][workingColumn].append(highResMap[highResRow][highResColum
                    # Groups pixels
            heightData[workingRow].append(mean(allPixelData[workingRow][workingColumn]))
            # Takes mean of the pixels from high res correspond to a single low res pixel
    return heightData   # returns heighData to be stored as sympy.Matrix

def generateTopoMap(self,highResMap):
# Function which generates a topographic map based on high resolution map
    size = len(highResMap)
    contourData = [[1 for i in range(size)] for j in range(size)] # Stores 1 for all pixels
    rightNeighboringPixels = [(0, 1), (1, 0)]   # Checkes the neighbors to the right and down
    for row in range(size):    # Loops through every row
        for column in range(size): # Loops through every column
            for contour in range(100,2000,100): # Contour lines will occur at 100 unit intervals
```

```
                if contourData[row][column] == 1:   # Skips this loop if the pixel has already been co
                    for neighbor in rightNeighboringPixels: # Loops over right neighbors
                        if row+neighbor[1] >= 0 and row+neighbor[1] < size and column+neighbor[0] >=
                        0 and column+neighbor[0] < size: # Checks to see if neighbor actually exists
                            if highResMap[row][column] >=
                            contour and highResMap[row+neighbor[1]][column+neighbor[0]] < contour:
                            # Checks whether the neighbor occurs on the other side of a contour line
                                contourData[row][column] = 0    # If so then color the contour line
                                break
                            elif highResMap[row][column] <=
                            contour and highResMap[row+neighbor[1]][column+neighbor[0]] > contour:
                            # Checks whether the neighbor occurs on the other side of a contour line
                                contourData[row][column] = 0    # If so then color the contour line
                                break
    return contourData  # returns contourData to be stored as sympy.Matrix

def generateSlopeList(self,map):
# Generates a dictionary containing the elevation change between pixel and all its neighbors
    size = len(map)
    neighboringPixels = [(0,0),(0,1),(1,0),(0,-1),(-1,0)]
    allTransitions = []
    for row in range(size):
        rowTransitions = []
        for column in range(size):
            transitionsDict = {}
            for neighbor in neighboringPixels:
                if row + neighbor[1] >= 0 and row + neighbor[1] <
                size and column + neighbor[0] >= 0 and column + neighbor[0] < size:
                    transitionsDict[(column + neighbor[0], row + neighbor[1])] =
                    map[row + neighbor[1]][column + neighbor[0]] - map[row][column]
            rowTransitions.append(transitionsDict)
        allTransitions.append(rowTransitions)
    return allTransitions

def identifyLandmarks(self, slopeList, size):
# Function which identifies peaks and saddles on the map
    peaks = self.identifyPeaks(slopeList=slopeList, size=size)
    saddles = self.identifySaddles(slopeList=slopeList, size=size)
    return peaks + saddles
```

```
def identifyPeaks(self,slopeList,size):     # Identifies peaks on the map
    listPeaks = []
    for row in range(size):
        for column in range(size):
            elevationChanges = list(slopeList[row][column].values())
            #make the dictionary a list so that the 0s at the beginning can be removed
            if len(elevationChanges) >= 5:
                elevationChanges.remove(0)
                if all(coordinate < 0 for coordinate in elevationChanges):
                #check for if all coordinates in list are negative (means surrounding neighbors
                are lower elevation, so it is a peak)
                    listPeaks.append((column,row)) #add to list of overall peaks
    return listPeaks

def identifySaddles(self,slopeList,size):  # Identifies saddles on the map
    listSaddles = []
    for row in range(size):
        for column in range(size):
            elevationChanges = slopeList[row][column]
            if len(elevationChanges) >= 5:
                if elevationChanges[(column+1,row)] < 0 and elevationChanges[(column-1,row)] < 0:
                # Saddles are identified as have opposite neighboring pixels having
                both negative or both positive slopes
                    if elevationChanges[(column,row+1)] > 0
                    and elevationChanges[(column,row-1)] > 0:
                        listSaddles.append((column,row))
                if elevationChanges[(column+1,row)] > 0 and elevationChanges[(column-1,row)] > 0:
                    if elevationChanges[(column,row+1)] < 0 and elevationChanges[(column,row-1)] < 0:
                        listSaddles.append((column,row))
    return listSaddles

def generateLandmarkMap(self,landmarks,size):
# Function which generates Goals Matrix (currently goals are peaks/saddles)
    landmarkMap = [[0 for j in range(size)] for i in range(size)]
    for landmark in landmarks:
        landmarkMap[landmark[1]][landmark[0]] = 1
    return landmarkMap
    def generateLowResLandmarkMap(self,highResMap,newSize):
    # Low res goals map is made by pixelating high res map and setting all non zeroes to 1
        lowResLandmarkMap = self.pixelateMap(highResMap=highResMap,newSize=newSize)
        landmarks = []  # Lists all landmarks in low res
        for row in range(len(lowResLandmarkMap)):
            for column in range(len(lowResLandmarkMap[row])):
                if lowResLandmarkMap[row][column] != 0:
                    lowResLandmarkMap[row][column] = 1
                    landmarks.append((column, row))
        return landmarks, lowResLandmarkMap


class Walker:   # Walker (hiker) class which moves on a region

    def __init__(self, region, startPosition, confidence, energy):
    # Initializes the walker with a number of attributes, most of which will change as it is updated
        self.region = region
        positionMap = self.generatePositionMap(position={startPosition:1}, default=0)
        confidenceMap = self.generatePositionMap(position={startPosition:confidence}, default='NA')
        energyMap = self.generatePositionMap(position={startPosition:energy}, default='NA')
        confidenceCategoryMap = self.categorizeTraitValues(valueMap=confidenceMap, scale=100)
        energyCategoryMap = self.categorizeTraitValues(valueMap=energyMap, scale=3000)
        if len(self.region.lowResLandmarks) > 0:
            visionMap = self.generatePositionMap(position={}, default='Landmark')
        else:
            visionMap = self.generatePositionMap(position={}, default='NA')
        self.strategy = self.generatePositionMap(position={}, default='NA')
        self.goal = self.generatePositionMap(position={}, default='NA')
        self.position = positionMap
        self.confidence = confidenceMap
        self.energy = energyMap
        self.confidenceCategory = confidenceCategoryMap
        self.energyCategory = energyCategoryMap
        self.vision = visionMap
        self.availableLandmarks = self.region.lowResLandmarks[:]
        if len(self.region.lowResLandmarks) > 0:
            if self.confidenceCheck():
                self.availableLandmarks = self.removeOccupiedLandmarks(landmarks=self.availableLandmarks)
            else:
                self.availableLandmarks = self.region.lowResLandmarks[:]
            if len(self.availableLandmarks) > 0:
                self.vision = self.generatePositionMap(position={}, default='Landmark')
                self.closestLandmarks = self.determineClosestLandmarks(landmarks=self.availableLandmarks)
            else:
                self.vision = self.generatePositionMap(position={}, default='NA')
        self.previousStrategyAndGoal = self.generatePositionMap(position={}, default=[])
        strategyMap = self.updateStrategy()
        goalMap = self.updateGoal(strategyMap=strategyMap)
        self.timeSeries = [[positionMap, confidenceCategoryMap, energyCategoryMap, strategyMap, goalMap]]
        self.strategy = strategyMap
        self.goal = goalMap

    def categorizeTraitValues(self, valueMap, scale=100):
        categorized = [[0 for i in range(self.region.workingSize)] for j in
        range(self.region.workingSize)]
        for row in range(len(valueMap)):
            for column in range(len(valueMap[row])):
                if valueMap[row][column] != 'NA':
                    normalizedValue =  valueMap[row][column] / scale
                    if normalizedValue >= 0.66:
                        categorized[row][column] = 'High'
                    elif normalizedValue < 0.33:
                        categorized[row][column] = 'Low'
                    else:
                        categorized[row][column] = 'Medium'
                else:
                    categorized[row][column] = 'NA'
        return categorized

    def updateOneTimeStep(self):     # Updates hiker as time goes on
        self.distanceMap = self.generateDistanceMap(goalMap=self.goal)
        # Creates distance map from goal
        self.possibleMovements = self.bellman(strategyMap=self.strategy, goalMap=self.goal)
        #self.possibleMovements = self.generatePossibleMovementsDictionary
        (strategyMap=self.strategy, goalMap=self.goal)
        # All possible movements and their likelihoods
        positionMap, energyMap, confidenceMap, strategyMap, goalMap, previousMap =
        self.updateStates(strategyMap=self.strategy,
        goalMap=self.goal, previousMap=self.previousStrategyAndGoal)
        confidenceCategoryMap = self.categorizeTraitValues(valueMap=confidenceMap, scale=100)
        energyCategoryMap = self.categorizeTraitValues(valueMap=energyMap, scale=3000)
        self.timeSeries.append([positionMap, confidenceCategoryMap, energyCategoryMap, strategyMap, goalMap
        self.position = positionMap
        self.confidence = confidenceMap
        self.energy = energyMap
        self.confidenceCategory = confidenceCategoryMap
        self.energyCategory = energyCategoryMap
        self.previousStrategyAndGoal = self.generatePreviousStratAndGoal(strategyMap=strategyMap,
        goalMap=goalMap, previous=previousMap)
        self.strategy = self.updateStrategy()
        self.goal = self.updateGoal(strategyMap=self.strategy, previous=self.previousStrategyAndGoal)
        if len(self.region.lowResLandmarks) > 0:
            if self.confidenceCheck():
                self.availableLandmarks = self.removeOccupiedLandmarks(landmarks=self.availableLandmarks)
            else:
                self.availableLandmarks = self.region.lowResLandmarks[:]
            if len(self.availableLandmarks) > 0:
                self.vision = self.generatePositionMap(position={}, default='Landmark')
                self.closestLandmarks = self.determineClosestLandmarks(landmarks=self.availableLandmarks)
            else:
                self.vision = self.generatePositionMap(position={}, default='NA')

    def confidenceCheck(self):
        for row in range(len(self.position)):
            for column in range(len(self.position[row])):
                if self.confidenceCategory[row][column] ==
                'High' or self.confidenceCategory[row][column] == 'Medium':
                    return True
        return False

    def removeOccupiedLandmarks(self, landmarks):
        allLandmarks = landmarks
        for landmark in landmarks:
            if self.position[landmark[1]][landmark[0]] > 0.5:
                allLandmarks.remove(landmark)
        return allLandmarks
```

```
def determineClosestLandmarks(self,landmarks):
# Measures the manhattan distance to every landmark from each pixel, marks which is the closest
    closestLandmarks = {}
    for row in range(len(self.position)):
        for column in range(len(self.position[row])):
            landmarkDist = {}
            for landmark in landmarks:
                distance = distanceCalculator(start=(column,row), stop=landmark)
                landmarkDist[distance] = landmark
            closestLandmarks[(column,row)] = landmarkDist.get(min(landmarkDist.keys()))
    return closestLandmarks

def generatePositionMap(self, position, default):
# Generates position map based on start position and movement matrix
    positionMap = [[default for i in range(self.region.workingSize)]
    for j in range(self.region.workingSize)]
    for coordinate in position.keys():
        positionMap[coordinate[1]][coordinate[0]] = position[coordinate]
        # If probabilityCloud is empty, timeStep=0 so 100% at initial position
    return positionMap

def updateStrategy(self):   # Decision Tree
    strategyData = pd.read_excel(r'decisionTree.xlsx',keep_default_na=False)
    ## read the decision tree excel file
    strategyDataFrame = pd.DataFrame(strategyData)  ##convert to data frame
    combinations = strategyDataFrame.values.tolist()  ## transform to list the excel file
    strategyMap = self.generatePositionMap(position={}, default='NA')
    for row in range(len(self.position)):
        for column in range(len(self.position)):
            if self.position[row][column] != 0:
                check = True
                for i in range(0, len(combinations)):
                    if combinations[i][0] == self.confidenceCategory[row][column]:
                        if combinations[i][1] == self.energyCategory[row][column]:
                            if combinations[i][2] == self.vision[row][column]:
                                if combinations[i][3] == self.strategy[row][column]:
                                    strategyMap[row][column] = combinations[i][4]
                                    check = False
                if check:
                    #print(self.confidenceCategory[row][column],
                    self.energyCategory[row][column],
                    self.vision[row][column],self.strategy[row][column])
                    strategyMap[row][column] = 'Directional'
    return strategyMap

def generatePreviousStratAndGoal(self, strategyMap, goalMap, previous):
    for row in range(len(strategyMap)):
        for column in range(len(strategyMap[row])):
            if self.position[row][column] > 0:
                previous[row][column] = previous[row][column] + [((column,row),
                strategyMap[row][column], goalMap[row][column])]
    return previous

def updateGoal(self, strategyMap, previous=[]):   # Selects goal based on strategy
    goalMap = self.generatePositionMap(position={}, default='NA')
    for row in range(len(strategyMap)):
        for column in range(len(strategyMap)):
            if strategyMap[row][column] != 'NA':
                if strategyMap[row][column] == 'Landmark':
                    goalMap[row][column] = self.closestLandmarks[(column,row)]
                elif strategyMap[row][column] == 'Directional':
                    if len(previous) != 0 and previous[row][column][-1][1] == 'Directional':
                            goalMap[row][column] = previous[row][column][-1][2]
                    else:
                        goalMap[row][column] = (column, row)
                elif strategyMap[row][column] == 'Backtracking':
                    if len(previous) != 0:
                        lastStrat = previous[row][column][-1][1]
                        check = True
                        for timeStep in range(len(previous[row][column])-1,-1,-1):
                            if previous[row][column][timeStep][1] != lastStrat:
                                check = False
                                goalMap[row][column] = previous[row][column][timeStep][0]
                        if check == True:
                            goalMap[row][column] = previous[row][column][0][0]
                    else:
                        goalMap[row][column] = (column, row)
                elif strategyMap[row][column] == 'Stay':
                    goalMap[row][column] = (column, row)
                else:
                    print('ERROR: Failed to identify strategy')
                    exit()
    return goalMap

def generateDistanceMap(self, goalMap):
# Function that calculates manhattan distance from every pixel to goal pixel
    allGoals = self.generatePositionMap(position={}, default='NA')
    for row in range(len(goalMap)):
        for column in range(len(goalMap[row])):
            if goalMap[row][column] != 'NA':
                allDistances = self.generatePositionMap(position={}, default=0)
                for r in range(len(goalMap)):
                    for c in range(len(goalMap[r])):
                        allDistances[r][c] = distanceCalculator(start=(c,r),
                        stop=goalMap[row][column])
                allGoals[row][column] = allDistances
    return allGoals

def generatePossibleMovementsDictionary(self,strategyMap,goalMap,slopeScale=100,distanceScale=1):
    possibleMovements = self.generatePositionMap(position={}, default='NA')
    for row in range(len(self.position)):
        for column in range(len(self.position[row])):
            if self.position[row][column] != 0:
                transitionSlopeFromPosition = self.region.lowResSlopeList[row][column]
                if strategyMap[row][column] != 'Directional':
                    if (column,row) == goalMap[row][column]:
                        normalizedCombined = {}
                        for neighboringPixel in transitionSlopeFromPosition.keys():
                            if neighboringPixel == goalMap[row][column]:
                                normalizedCombined[neighboringPixel] = 1
                            else:
                                normalizedCombined[neighboringPixel] = 0
                    else:
                        slopes = {}
                        distances = {}
                        for neighboringPixel in transitionSlopeFromPosition.keys():
                            slopes[neighboringPixel] =
                            activationFunction(x=transitionSlopeFromPosition[neighboringPixel],
                            scale=slopeScale)
                            distances[neighboringPixel] = activationFunction(x=self.distanceMap
                            [row][column][neighboringPixel[1]][neighboringPixel[0]],
                            scale=distanceScale)
                        normalizedSlopes = normalizeBySum(dict=slopes)
                        normalizedDistances = normalizeBySum(dict=distances)
                        combined = {}
                        for transition in transitionSlopeFromPosition.keys():
                            combined[transition] = normalizedSlopes[transition]
                            * normalizedDistances[transition]
                        normalizedCombined = normalizeBySum(combined)
                else:
                    slopes = {}
                    distances = {}
                    for neighboringPixel in transitionSlopeFromPosition.keys():
                        slopes[neighboringPixel] =
                        activationFunction(x=transitionSlopeFromPosition[neighboringPixel],
                        scale=slopeScale)
                        distances[neighboringPixel] = 1/activationFunction
                        (x=self.distanceMap[row][column]
                        [neighboringPixel[1]][neighboringPixel[0]],scale=distanceScale)
                    normalizedSlopes = normalizeBySum(dict=slopes)
                    normalizedDistances = normalizeBySum(dict=distances)
                    combined = {}
                    for transition in transitionSlopeFromPosition.keys():
                        combined[transition] = normalizedSlopes[transition]
                        * normalizedDistances[transition]
                    normalizedCombined = normalizeBySum(combined)
                final = {}
                for coordinate in normalizedCombined.keys():
                    final[coordinate] = [normalizedCombined[coordinate], self.energy[row][column]
                    - abs(transitionSlopeFromPosition[coordinate])*(1/100), self.confidence[row][colum
                possibleMovements[row][column] = final
    return possibleMovements

def calculateElevationCost(self, elevation, energyPerTile=11.5, tilesPerMile=5,
massOfHiker=80, declineDiscount=0.95):
    """Calculates the incline or decline between two points, using the piecewise
```

```
        function defined in the Bellman Equations
        """
        b = declineDiscount

        if elevation >= 0:
            return energyPerTile + 0.3845*massOfHiker*9.81*(1/tilesPerMile)*elevation
        else:
            return energyPerTile - 0.3845*massOfHiker*9.81*b*(1/tilesPerMile)*elevation

    def bellman(self, strategyMap, goalMap):
        """Calculates the next step of the hiker using the Bellman Equations
        """
        energyPerTile = 11.5
        costToExist = 60
        indexOfFatigue = 1
        feetPerMile = 5280
        tilesPerMile = 5

        possibleMovements = self.generatePositionMap(position={}, default='NA')
        for row in range(len(self.position)):
            for column in range(len(self.position[row])):
                if self.position[row][column] != 0:
                    slopeDict = self.region.lowResSlopeList[row][column]
                    if strategyMap[row][column] != 'Directional'
                    and (column, row) == goalMap[row][column]:
                        normalizedCombined = {}
                        for neighboringPixel in slopeDict.keys():
                            if neighboringPixel == goalMap[row][column]:
                                normalizedCombined[neighboringPixel] = 1
                            else:
                                normalizedCombined[neighboringPixel] = 0
                    else:
                        slopes = {}
                        distances = {}
                        #print(slopeDict)
                        for neighboringPixel in slopeDict.keys():
                            slopes[neighboringPixel] =
                            (self.calculateElevationCost(
                            (slopeDict[neighboringPixel]*tilesPerMile)/feetPerMile))
                    #Feet to miles then miles to tiles
                    distances[neighboringPixel] = self.distanceMap[row][column]
                    [neighboringPixel[1]][neighboringPixel[0]]
                #print(slopes)
                #print(distances)
                costToMoveDict = {}
                for transition in slopes.keys():
                    costToMoveDict[transition] = slopes[transition]
                #print(costToMoveDict)
                expectedCostDict = {}
                for transition in costToMoveDict.keys():
                    if strategyMap[row][column] != 'Directional':
                        expectedCostDict[transition] = self.energy[row][column]
                        - (energyPerTile*(distances[transition]))
                        - (indexOfFatigue*costToMoveDict[transition]) - costToExist
                    else:
                        expectedCostDict[transition] = self.energy[row][column]
                        + (energyPerTile*(distances[transition]))
                        - (indexOfFatigue * costToMoveDict[transition]) - costToExist
                #print(expectedCostDict)
                translationValue = min(list(expectedCostDict.values()))

                #denominator = max(list(expectedCostDict.values()))
                for transition in costToMoveDict.keys():
                    expectedCostDict[transition] = expectedCostDict[transition]
                    - translationValue  #expectedCostDict[transition] =
                    math.exp(expectedCostDict[transition]/denominator)
                #print(expectedCostDict)
                normalizedCombined = normalizeBySum(dict=expectedCostDict)
                #print(normalizedCombined)
            final = {}
            for coordinate in normalizedCombined.keys():
                final[coordinate] = [normalizedCombined[coordinate], self.energy[row][column]
                - self.calculateElevationCost((slopeDict[coordinate]*tilesPerMile)/feetPerMile)
                + energyPerTile
                - costToExist, self.confidence[row][column] - 0.34]
            possibleMovements[row][column] = final
return possibleMovements
```

```
    def updateStates(self, strategyMap, goalMap, previousMap):
        newPositionMap = self.generatePositionMap(position={}, default=0)
        newEnergyMap = self.generatePositionMap(position={}, default='NA')
        newConfidenceMap = self.generatePositionMap(position={}, default='NA')
        newStrategyMap = self.generatePositionMap(position={}, default='NA')
        newGoalMap = self.generatePositionMap(position={}, default='NA')
        newPreviousMap = self.generatePositionMap(position={}, default=[])
        for row in range(len(self.possibleMovements)):
            for column in range(len(self.possibleMovements[row])):
                if self.possibleMovements[row][column] != 'NA':
                    dict = self.possibleMovements[row][column]
                    for transition in dict.keys():
                        newPositionMap[transition[1]][transition[0]] +=
                        self.position[row][column]*dict[transition][0]
                        if newEnergyMap[transition[1]][transition[0]] != 'NA':
                            if newEnergyMap[transition[1]][transition[0]] < dict[transition][1]:
                                newEnergyMap[transition[1]][transition[0]] = dict[transition][1]
                                newStrategyMap[transition[1]][transition[0]] = strategyMap[row][column]
                                newGoalMap[transition[1]][transition[0]] = goalMap[row][column]
                                newPreviousMap[transition[1]][transition[0]] = previousMap[row][column]
                        else:
                            newEnergyMap[transition[1]][transition[0]] = dict[transition][1]
                            newStrategyMap[transition[1]][transition[0]] = strategyMap[row][column]
                            newGoalMap[transition[1]][transition[0]] = goalMap[row][column]
                            newPreviousMap[transition[1]][transition[0]] = previousMap[row][column]
                        if newConfidenceMap[transition[1]][transition[0]] != 'NA':
                            if newConfidenceMap[transition[1]][transition[0]] < dict[transition][2]:
                                newConfidenceMap[transition[1]][transition[0]] = dict[transition][2]
                        else:
                            newConfidenceMap[transition[1]][transition[0]] = dict[transition][2]
        return newPositionMap, newEnergyMap, newConfidenceMap, newStrategyMap, newGoalMap, newPreviousMap


def generateGIF(files): # Generates gif at the end
    topoMap = Image.open(directory + '/' + 'topoMap.png')
    goalMap = Image.open(directory + '/' + 'goalsMap.png')
    Image.alpha_composite(topoMap, goalMap).save(directory + '/' + 'background.png')
    background = Image.open(directory + '/' + 'background.png')
    images = []
    for filename in files:
        probabilityCloud = Image.open(filename)
        minusPNG = filename.split('.')[0]
        Image.alpha_composite(background, probabilityCloud).save(minusPNG + 'withBackground.png')
        images.append(imageio.imread(minusPNG + 'withBackground.png'))
    imageio.mimsave(directory + '/' + 'heatmap.gif', images, duration=1)


fullSize = 700
# Sets resolution of high resolution map
workingSize = 70
# Sets resolution of low resolution map, for simplicity, please keep the value some
factor of the fullSize
startTime = datetime.now()
terrainSeed = 44259 #random.randint(0,100000)
region = Region(fullSize=fullSize, workingSize=workingSize,

terrainSmoothness=fullSize*(5/16),terrainSeed=terrainSeed)
#start = (random.randrange(workingSize), random.randrange(workingSize))

#  UNCOMMENT OUT YOUR LINE #
#yAxis = 20  ########## Camila
#yAxis = 25  ########## Jose
#yAxis = 30  ########## James 1
#yAxis = 35  ########## James 2
#yAxis = 40  ########## James 3
#yAxis = 45  ########## Madeline
#yAxis = 50  ########## Jerry

for start in range(20, 55, 5):
    directory = 'LostHiker_'+ str(terrainSeed) + '_' + str(start) + '_' + str(yAxis)
    if os.path.isdir(directory):
        shutil.rmtree(directory)
    os.mkdir(directory)
    filenames = []
    timeSteps = 100
    hiker = Walker(region=region, startPosition=(start,yAxis), confidence=50, energy=1500)

    print('\nPathfinding:')
```

```
for timeStep in range(timeSteps):
    printMapToPNG(map=hiker.position ,colorRange=('clear', 'purple'),valueRange=(0, 1),
    path=directory + '/' + str(timeStep) + '.png',sizeScale=int(fullSize/workingSize))
    filenames.append(directory + '/' + str(timeStep)+'.png')
    print('--Time Step ' + str(timeStep) + ': ' + str(datetime.now() - startTime))
    hiker.updateOneTimeStep()
print()

printMapToPNG(region.topoMap, colorRange=('black','white'), valueRange=(0,1),
path=directory + '/' + 'topoMap.png')
printMapToPNG(region.lowResLandmarkMap, colorRange=('clear','gold'), valueRange=(0,1),
path=directory + '/'
+ 'goalsMap.png', sizeScale=int(fullSize/workingSize))
print('Printed Background Maps to PNG: ' + str(datetime.now() - startTime))

generateGIF(files=filenames)
print('Combined Maps and Generated Pathfinding GIF: ' + str(datetime.now() - startTime))
```