Traversing Randomly Generated Mazes

A Monte Carlo Approach

By: Adam Rayfield, Andrew Roberts, and Billy Witrock

1. Abstract

Mazes designed with specific structures can be used in theory to represent real locations, such as the rooms of a building or streets of a city. A maze fundamentally consists of a complex list of relationships between different locations, which can be mathematically represented as an adjacency matrix for mathematical analysis. In this paper, we address the problem of simulating a search party seeking to find a lost individual in a maze. Utilizing a Monte Carlo simulation, we analyze the expected number of movement steps to find a wanderer in two-dimensional mazes, which are either generated randomly using the binary maze algorithm or are fully-connected (i.e, all passages are unobstructed). We consider increasing the size of the search party as well as modifying the search algorithm itself. Our general results show that the bias associated with a specific type of maze has a significant effect on the expected number of step counts required for the search party to find the lost individual. We also conclude that the average behavior of the simulations for each maze becomes more similar as the size of the search party increases.

2. Problem

While not always obvious to the casual onlooker, many real-world geographical locations reduce to simple mazes at their most basic level. If one abstracts away from the traffic, noise, and pollution of a crowded city street and focuses solely on its geography, a maze provides an accurate approximation of this reality. Due to the vast number of settings in which a maze can serve a useful modeling purpose, we seek to better understand the behavior of a maze system through which "particles" take random walks.

Specifically, we consider two distinct maze archetypes, concentrating our analysis on mazes that share a similar set of characteristics. We then introduce randomly placed "people" into the mazes, each of which traverse the landscapes according to a random walk. This basic setup offers the flexibility to easily model numerous real-world phenomena. In this paper, we consider a variety general scenarios.

Our first scenario begins with two individuals randomly placed in a maze. Intuitively, we consider one individual as a search party, actively seeking to find the other person who is lost in the maze. Henceforth, we will refer to these individuals as the *searcher* and *roamer,* respectively. We assume the landscape if foreign to both individuals, so that they simply take random walks over the maze. This scenario resembles a "double blind" hide-and-seek, in which both individuals randomly traverse the terrain until a collision occurs (i.e., the *roamer* is found).

This paper considers additional scenarios occurring in these two distinct maze classes. We seek to understand how an increase in the number of *searchers* and changes to the *searchers'* movement algorithm influence the *searchers'* success in finding the *roamer*.

3. Literature Review

A 2015 paper by Sam Snodgrass and Santiago Ontanon at Drexel University discusses procedural generation of two-dimensional video game maps and discusses the ability to categorize low-level structures and their higher-level combinations using a set of Markov Chains [1]. This approach may be useful for the generation of large maps from a known set of elements, mainly if the "maze" being tested is intended to be made out of similar parts, and is an example of how mazes can be applied to different computational problems. For

example, this kind of map generator could produce a set of rooms more analogous to a building than a more typical series of twisted corridors.

Professor Whitt from Columbia University has produced available lecture notes that describe, in simpler terms, the means by which a stochastic matrix can describe the mapping between different regions of a maze; these could be rooms of a building or intersections of a corridor [2]. It is possible to examine the long-term behavior of these, including whether the mazes are periodic. However, a maze with exits cannot be modeled by an irreducible stochastic matrix. Professor Whitt suggests a means of analyzing such a matrix as an "absorbing Markov chain," which can be used to examine behavior in the "transient" state and the expected number of steps before exit.

A class project by Robert Ramirez from Washington University in St. Louis discusses maze generation techniques and uses the resulting mazes to simulate a fugitive attempting to escape a building while being pursued by a SWAT team [3]. Ramirez examines how increasing the number of people searching for the fugitive increases the probability of success, but focused primarily on the generation and does not apply different searching algorithms in his simulations.

A book by Jamis Buck describes various maze generation algorithms which begin from a representation of a maze which does not contain any connections [4]. The simplest maze generation algorithm described is the biased Binary Tree Algorithm (see Section 4.2.2). Significantly better maze generation algorithms have been the subject of prior research, such as work presented in 2009 from the Tokyo Institute of Technology and Japan Advanced Institute of Science and Technology, concerning the generation of mazes which have pathways spanning an input image from a designated entrance to a designated exit [5]. This involves the generation of a "Hamiltonian Path" between the two endpoints of the pathway which visits each node of the maze overlapping the input image, which may be difficult to implement.

4. Model

4.1 Maze Representation

We represent the mazes using a graph structure, which is in turn represented by an adjacency matrix. Letting M denote the symmetric adjacency matrix, the element M_{ij} takes the value one if a path exists from vertex *i* to vertex *j* and zero if a wall bars direct passage between the two. In this paper, we consider 121 node graphs corresponding to 11x11 mazes. We choose to number the nodes row-by-row from left to right, such that the top-left maze location corresponds to vertex 1 and the bottom-right to vertex 121. A maze can be represented as a graph and as an adjacency matrix (Figure 1).

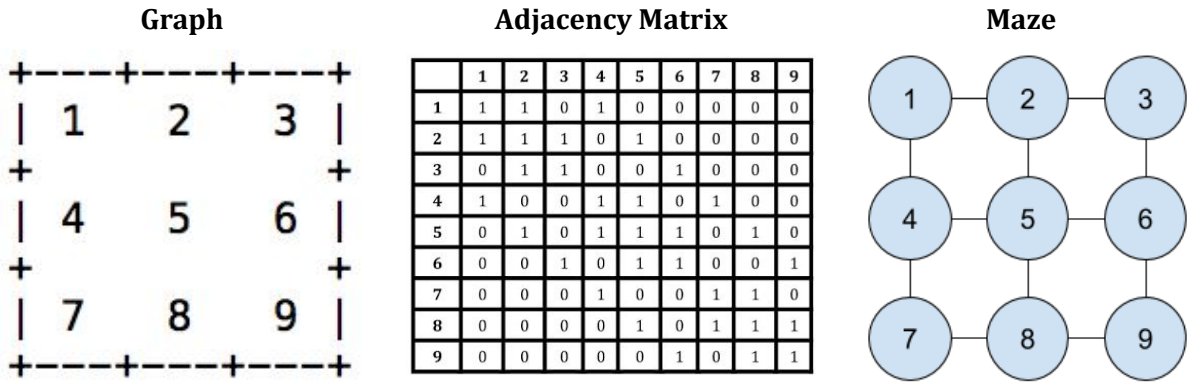


Figure 1: A simplified 3x3 fully-connected maze, with both an equivalent graph and adjacency matrix.

4.2 Maze Generation

To better understand the impact of the landscape of the *searcher's* success, we perform simulations using two distinct maze classes, each of which is described below.

4.2.1 Fully-Connected Maze

Figure 1 corresponds to a fully-connected maze, which essentially can be thought of as grid. Conditional on the number of nodes, this graph is non-stochastic as it simply allows passage between any adjacent node not blocked by the outer wall. Thus, for any interior node in the graph, the *searcher* and *roamer* are able to move in all four directions.

4.2.2 Binary Maze

The binary maze is a random maze generated by a simple binary algorithm. Beginning with an unconnected maze ($M_{ij} = 0, \ \forall \ i \neq j$), the algorithm then visits each node in the graph and with probability 0.5 generates a path either to the node directly to its right or directly downwards [4]. If a wall obstructs one of these directions, then the algorithm extends the passage to the remaining direction with probability one. Although stochastic, the binary maze's generation algorithm leads to *bias*, meaning every maze generated with this algorithm will share certain characteristics. In the context of this research, this bias is useful as it allows us to explore the effect that these characteristics have on the simulation results. Concretely, our implementation of a binary maze will always feature an open passageway along both the right and bottom wall of the maze. Furthermore, a path will always exist from node 1 to node 121 by simply moving right or down at each location (Figure 2). The binary maze is an example of a "perfect maze", in which there exists exactly one path between each pair of nodes [4].

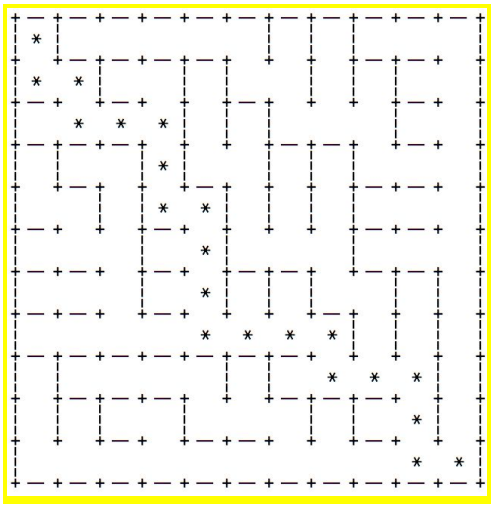


Figure 2: An 11x11 binary maze showing maze bias.

4.3 Search Algorithm

After generating one of the two types of mazes described above, we begin each individual simulation by randomly placing the *searcher* and *roamer* on one of the 121 nodes in the graph. In discrete time steps, each individual moves to one of the connected nodes according to a uniform distribution. We allow both the *searcher* and the *roamer* to remain fixed at their current locations at each time step; this can be thought of as either individual considering their next move before acting. For example, at the interior nodes of a fully-connected maze, each individual can move in any of the four directions or remained fixed in place at

each time step, each with probability one-fifth. This process repeats until the *searcher* and *roamer* find themselves on the same node.

For further analysis, several variations of the random search algorithm behavior were examined as well. Initially, attempts were made to examine the long-term performance of a *searcher* which stops at a certain location in the maze. Initially, the *searcher* would follow the usual random movement algorithm for a number of steps (set arbitrarily to be equal to the size of the mazes, 121) and maintain a variable which contains the largest number of connections from an encountered node. After this period, when a node with connections greater than or equal to the largest number seen is encountered, the *searcher* will stop moving and remain in place until the end of the simulation. Another variation simply had the *searcher* stop at the first location with greater than 3 possible paths to choose (i.e. an intersection of multiple corridors). Modified movement algorithms were also considered, where the *searcher* cannot move to the node at which it was previously located or where the *searcher* does not have the option of remaining at the node it is currently located at.

4.4 Monte Carlo Simulation

To test the average differences in the results across varied scenarios, we utilize Monte Carlo simulations. By averaging the output of many individual maze-search simulations, we are able to obtain the average results associated with a certain type of maze (or with other search factors), rather than picking up on random noise that may be associated with any single simulation. Results analyzed include the average number of steps required for a *searcher* to find the *roamer* and the locations at which the individual maze simulations are likely to end.

5. Assumptions and Weaknesses

In order to model these "hide-and-seek" dynamics we make several assumptions. Searchers are also assumed to be randomly placed, which currently includes the possibility that the *roamer* may be found at the beginning. On a very basic level, the actors in are maze are assumed to have no knowledge of the terrain, only basic tactile senses, and maintain no memory of the nodes visited except in certain tested algorithms. Practically, this signifies that searchers in the baseline algorithm simply wander through the maze randomly, choosing arbitrarily at each location where to go next. Furthermore, our model offers no possibility for more complex behaviors or strategies, such as calling out for the lost individual in the first scenario, or guarding the exit in the second scenario. The actors also move at the same speed and an individual is only classified as "found" if a *searcher* physically collides with a *roamer*. It would not be difficult to increase the complexity of this model, giving the actors different abilities and senses, or allowing them to employ various search strategies. In this paper, we only consider slight variations of our baseline analysis, but these potential expansions in complexity are interesting sources of future research.

6. Implementation

We use MATLAB to implement the functions necessary to run our simulations. We define both the *searcher* and *roamer* as instantiations of a Person class, which tracks each individual's current and past locations as well as contains functions to move and randomly place each person. This class interacts seamlessly with the maze implementation, which is stored as a binary matrix and generated by a separate function. We use another function to run a single maze simulation. This function generates a maze, randomly places the *searcher* and *roamer*, iterates through time steps until a stop condition is met (found), and returns summary information used for analysis. This single simulation function is then wrapped in a Monte Carlo function which performs a large number of these simulations and averages the results.

7. Analysis

7.1. Baseline Analysis

Using a monte carlo approach with 100,000 iterations, we find the mean number of steps required for a single *searcher* to find the *roamer* in a binary maze is 919.301. The simulation converged to this average in approximately 4,000 steps. The mean step count associated with the fully-connected maze is considerably lower; the simulation approaches the average of 230.85 after around 3,000 steps.

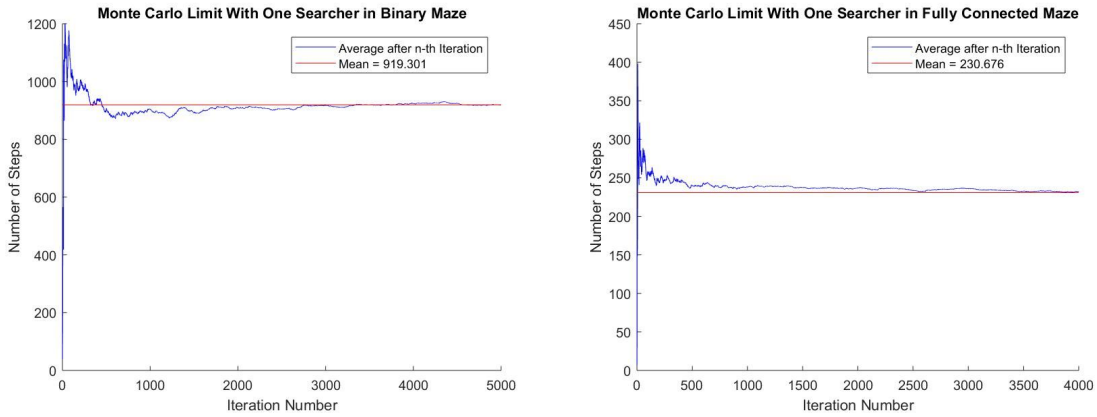


Figure 3: Monte Carlo Simulations approaching the mean

The median number of steps associated with these two simulations is 575 and 157, respectively. The drastic difference between the mean and median may imply that outliers are inflating the mean. The presence of outliers becomes clear as we observe the histogram showing the frequencies of the step counts associated with each iteration of the Monte Carlo simulation.

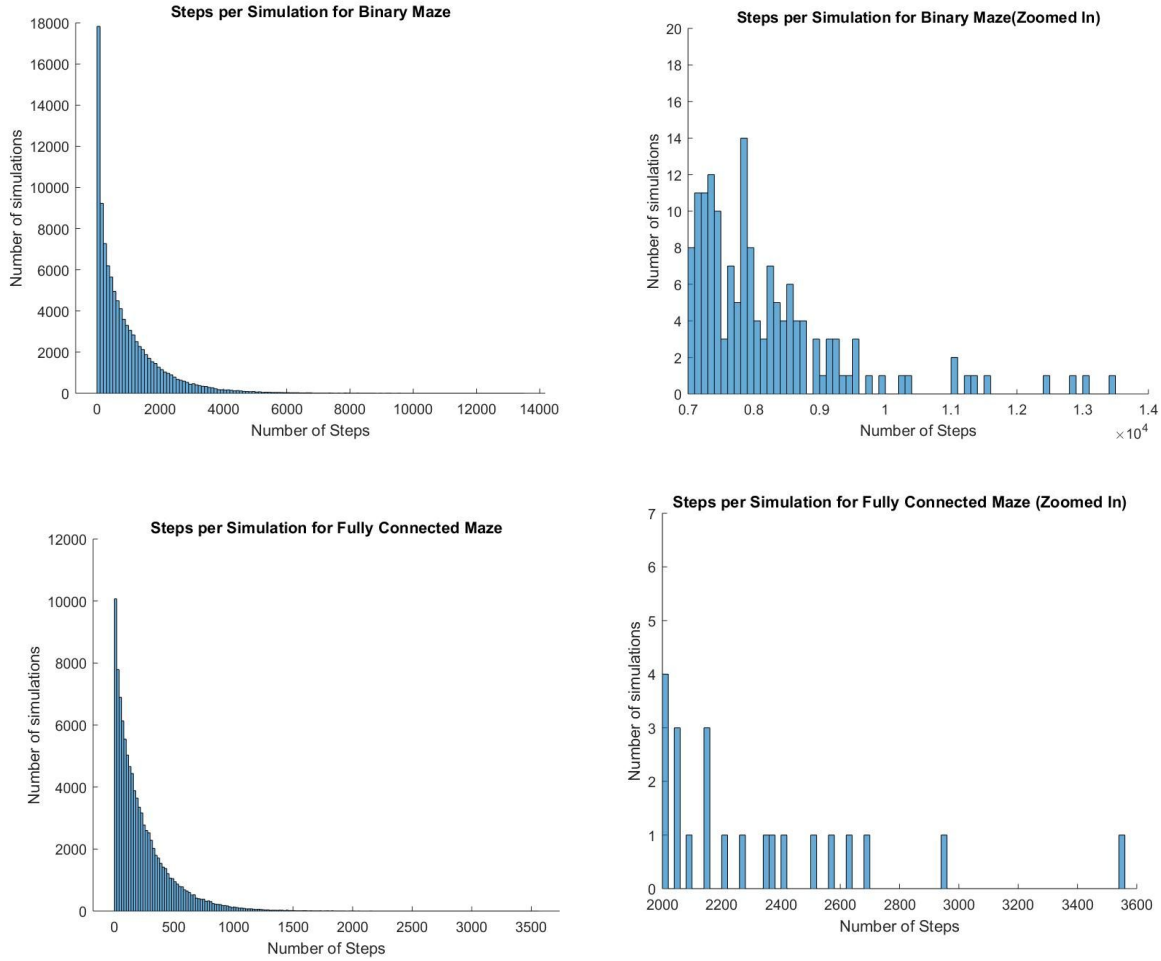


Figure 4: Results from individual performances of maze simulation in both maze types

Defining outliers as step counts at least 1.5 interquartile ranges away from the upper or lower quartile, we discover that the binary and fully-connected maze simulations have approximately 5,000 and 4,800 outliers, respectively. Outlier removal changes the associated mean step counts to 749.35 and 194.52. These averages are still considerably higher than the medians, indicating right skewness in our data; the histograms confirm this hypothesis.

These results are unsurprising as the probability of the two individuals in an 11x11 maze randomly starting two or fewer places away from each other is 13/121, or about 10.7%. If we assume that about 10% of these "close starts" require very few step counts to achieve impact and only 5% of simulations result in uncharacteristically high step counts, we can begin to gain an intuitive grasp as to why our data shows this skewness.

Another useful avenue for analysis thus far ignored in this paper is to observe on which nodes the *searcher* typically finds the *roamer* (henceforth, the "impact location").

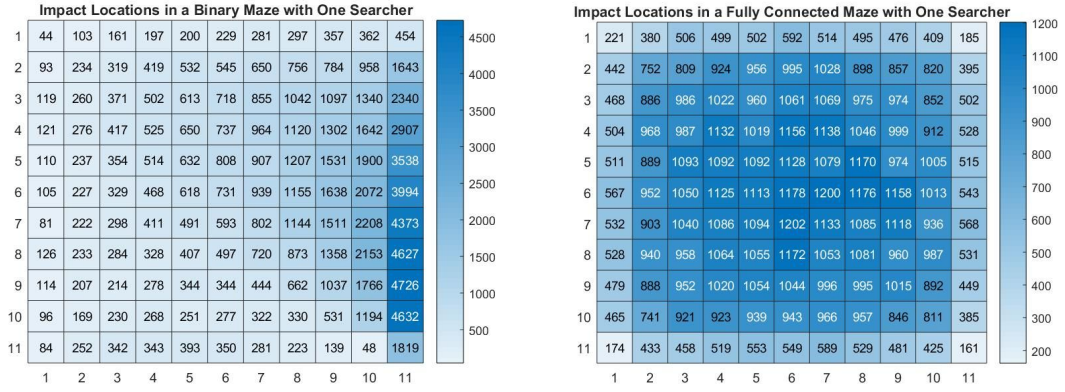


Figure 5: Impact locations for both maze types with one searcher

Looking at the distribution of impact locations for each maze type, it becomes clear that maze characteristics heavily influence the dynamics of the simulation. The impact locations for the binary maze are heavily concentrated along the eastern wall, which will always feature an unobstructed passage (see image in section 4.2.2). The binary maze features a diagonal wall stretching from the top-left node to the bottom-right node; on average we expect this wall to divide the maze in half. Therefore, we can assert that the *searcher* and *roamer* start on opposite sides of the maze with probability ½. When this is the case, one of the two individuals must move to the right and down far enough to be able to circumvent the diagonal wall. Therefore, probability dictates that they will likely meet about equidistant from each other, which corresponds to the lower-right corner on the maze. Furthermore, as the entire eastern passage is unblocked, once a person hits the rightmost wall, they will typically remain along the wall at the next time step with probability ¾ . This explains the concentration of impacts along the right wall.

The fully-connected maze impact locations are also unsurprising considering the nature of the maze. They are concentrated around the center of the maze and spread towards the outer boundaries fairly uniformly. This is a natural consequence of the fully-connected maze as an individual is able to move to any adjacent node. Therefore, the *searcher* and *roamer* will most often meet in the middle.

7.2. Multiple Searchers

We now conduct a similar analysis, but assess the impact of adding multiple *searchers*.

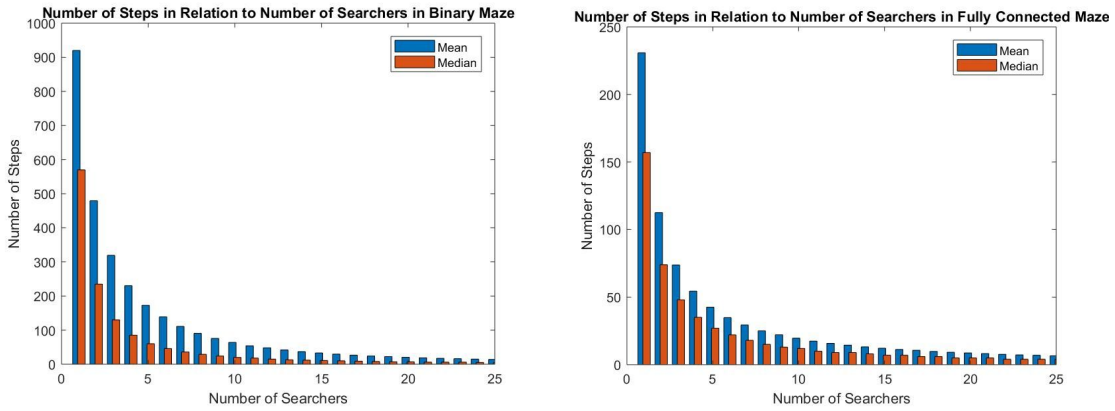


Figure 6: Mean and median number of steps for both maze types with varied number of searchers

As illustrated by these figures, adding *searchers* decrease the mean and median at similar rates in both mazes. In the binary maze, adding the first additional *searcher* cuts the amount of steps needed about in half. Once above twenty, adding an additional *searcher* only decreases the average number of steps by one. The median falls much faster than the mean, dropping below ten once seventeen *searchers* are introduced, and below five with twenty-five *searchers*.

The fully-connected maze exhibits similar behavior. The mean also drops by about 50% when adding the second searcher. However, since the mean number of steps is already lower than with the binary maze, adding a thirteenth *searcher* only decreases the number of steps by one.

An additional observation is that the ratio of the median over the mean is closer to one for the fully-connected maze irrespective of the number of *searchers*, implying the fully connected maze has less of a skewed distribution. The ratio changes in similar ways as more *searchers* are added. The figures below demonstrate that skewness increases in the number of *searchers*.

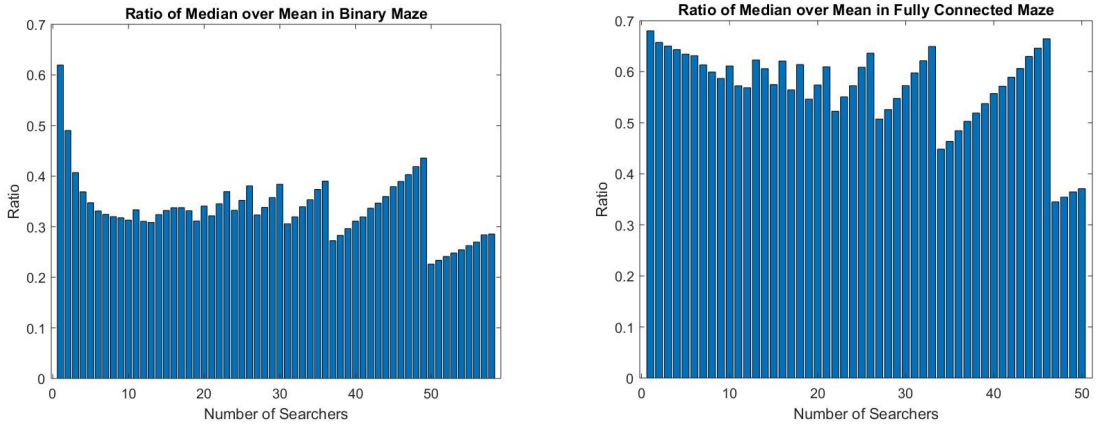


Figure 7: Ratio of Median divided by Mean for both maze types with varied number of searchers

Furthermore, the maze bias becomes less prevalent in the distribution of impact locations as the number of *searchers* is increased. The heat maps below demonstrate that the distribution becomes more uniform across the mazes. This is likely being driven by the decrease in median number of steps before impact; as the median number of steps decreases, the starting locations of the individuals become a larger factor while the subsequent paths taken become less relevant.

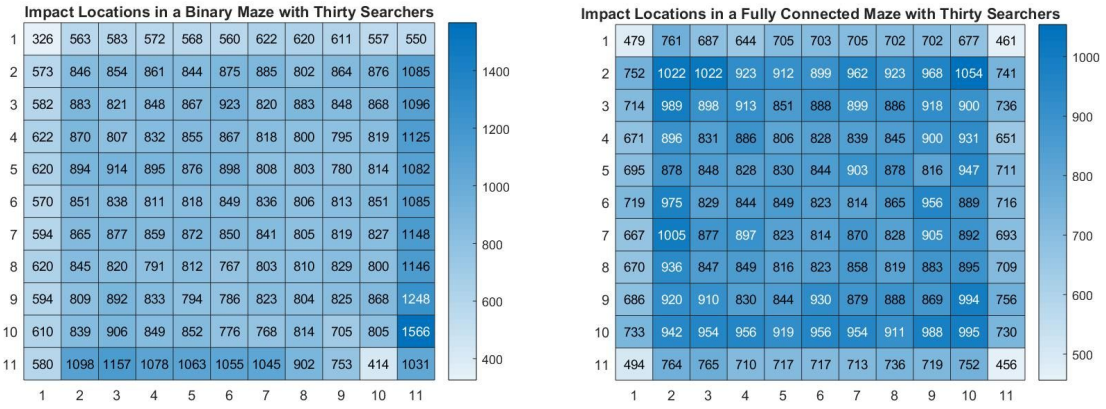
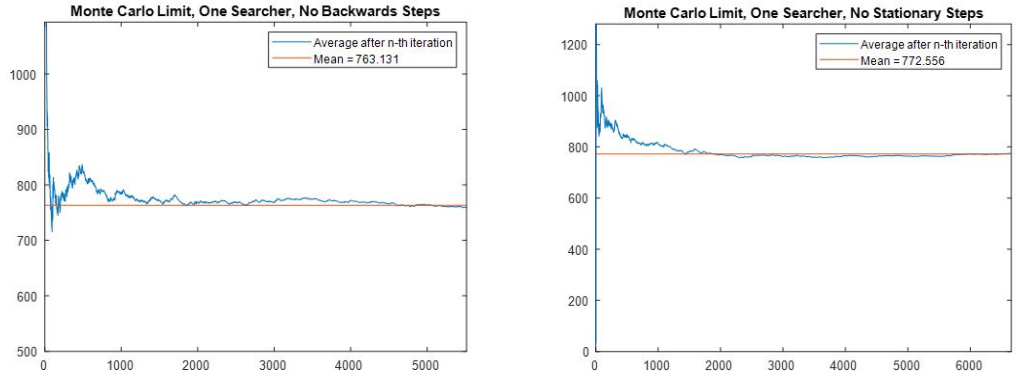


Figure 8: Impact locations for both maze types with thirty searchers

7.3 Changing the Search Algorithm

Initially, the first waiting algorithm using searcher memory for the first 121 steps was implemented and tested. However, Monte Carlo analysis of this algorithm stalled. Debugging an initial analysis attempt for the first waiting algorithm in MATLAB in a binary maze was conducted during the sixth Monte Carlo iteration. The movement simulation was halted after nearly 12 hours, having finished 1,856,290 steps with the roamer failing to locate the stopped searcher, which was located at node 114 (i.e. at row 11, column 4 on impact plots). This algorithm and the second waiting algorithm where the searcher is stopped at the first corridor intersection it reaches were noted to lead to similar problems relatively frequently and subsequently discontinued from rigorous analysis.

Two modifications to the random search algorithm were analyzed through Monte Carlo methods, one which prevented the searchers' movements from selecting "backwards" steps to the node it was located at during the previous step (Figure 9) and one which prevented the selection of "stationary" steps where the searcher remains at the same node (Figure 10).



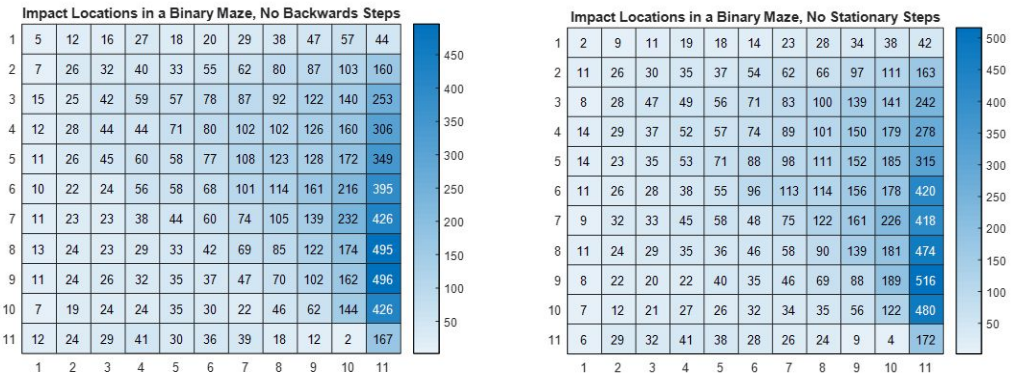


Figure 9: Results of 10,000 Monte Carlo simulations using one searcher in a binary maze with two modified search algorithms: prevention of "backwards" steps (Left) and prevention of "stationary" steps (Right). Relative to the random search mean value of 919.301 steps, these algorithms reduced the mean steps until impact to 763.131 and 772.556 steps respectively. Monte Carlo simulations converged visibly after 5,000-6,000 iterations. As before, bias in impact locations towards the right wall of the maze is noticeable.

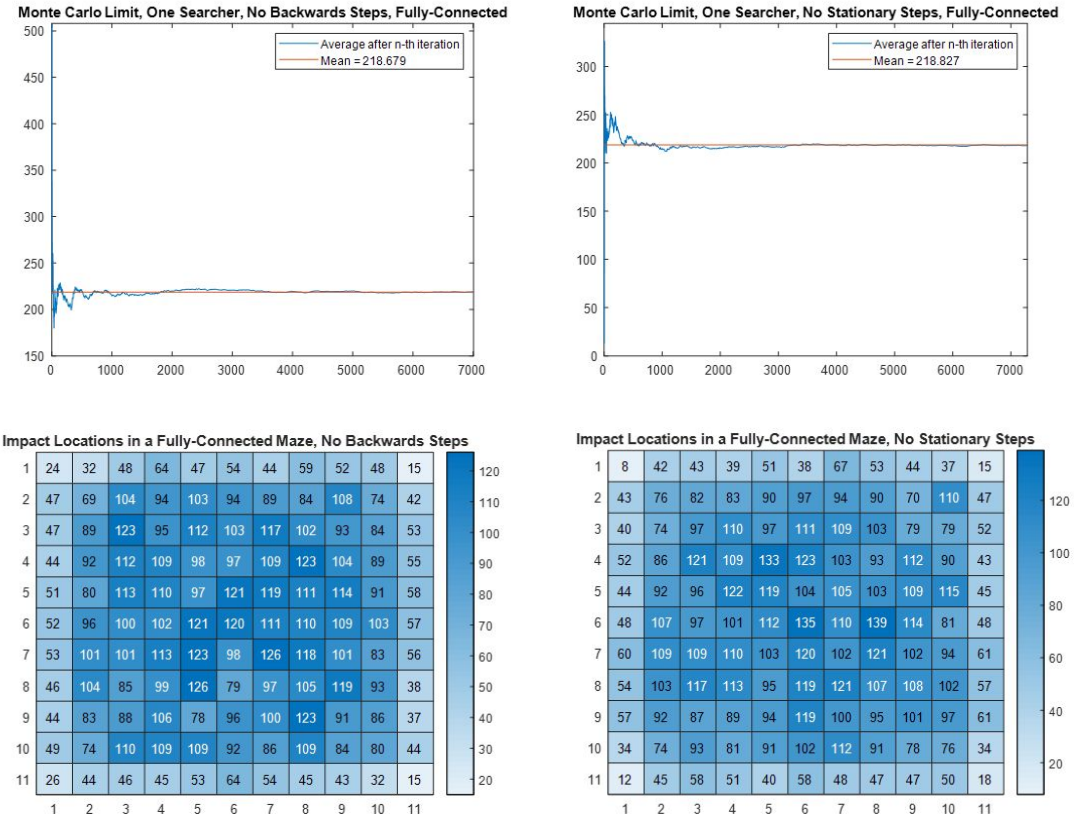


Figure 10: Results of 10,000 Monte Carlo simulations using one searcher in fully-connected mazes with two modified search algorithms: prevention of "backwards" steps (Left) and prevention of "stationary" steps (Right). Relative to the random search mean value of 230.676 steps, these algorithms reduced the mean steps until impact to similar values of 218.679 and 218.827 steps respectively. Monte Carlo simulations converged visibly after 5,000-6,000 iterations. As before, the impact locations have become distributed more uniformly throughout the maze, albeit lesser on the edges.

8. Conclusions

In this paper, we simulate a "cat and mouse" maze search, tweaking basic parameters to determine their effect on a *searcher's* success in finding a lost individual. Although our models make strong assumptions in the interest of simplicity, we are able to arrive at a handful of interesting conclusions. Our primary finding is that maze bias plays a significant role in determining the number of steps required before an impacts occurs. The nature of the binary maze algorithm is such that impact locations are concentrated at the bottom-right wall of the maze, while the fully-connected maze results in a more uniform distribution of impacts, with a high concentration in the middle. We also find that the most frequent step count in the single simulations was zero, reflecting the fact that the probability of the two individuals starting in the same location is greater than the probability of collision on any other non-zero time step. Although the majority of step counts fell on the lower end of the distribution, both mazes featured a fairly significant presence of outliers, reflecting a right skewness in the step count distribution. In the second phase of our analysis, we find the rate at which the step count decreases as the number of *searchers* increases is fairly similar across both maze types. Moreover, the impact distributions become significantly more uniform for simulations with more *searchers,* reflecting the fact that the initial random placement tends to drive results more in these scenarios.

The failure of algorithms which implement waiting behavior to work properly may be because the waiting searcher is not guaranteed to be anywhere close to the likely long-term behavior of a random wanderer, or the bottom right wall of the maze for binary mazes. By comparison, the sampled waiting algorithm which was unable to complete had the searcher stopped near the bottom left of the maze. Simulations such as conducted in this project may inform better waiting strategies in the future, as could using Markov Chains to produce a stationary probability vector of the maze for a random wanderer [2]. Having searchers begin and wait in a nonrandom location where a random wanderer is expected to gravitate towards due to maze bias may be a viable strategy; simulations of such a strategy should be pursued in the future. Both movement-related changes to the algorithms were confirmed to produce better results than purely random searching in binary mazes. This effect was somewhat stronger for the prevention of "backwards" steps, though the prevention of "stationary" steps still performs well, and both perform similarly in fully-connected mazes. This may indicate that generally moving in a manner which has the potential to observe more regions quickly is preferable to slower random wandering. More sophisticated algorithms in this vein could include a decision tree which prevents exploration of previously-visited regions when able and should be studied in any future research.

References

1. S. Snodgrass and S. Ontañón, “A Hierarchical MdMC Approach to 2D Video Game Map Generation,” Association for the Advancement of Artificial Intelligence, 2015, 205–11.
2. W. Whitt, “Lecture Notes: Introduction to Markov Chains,” n.d., available at http://www.columbia.edu/~ww2040/4106S11/lec0127.pdf
3. Robert Ramirez, “Applications of Random Mazes and Graphs Generated via Markov Chain Monte Carlo Methods,” Washington University of St. Louis, 2012, available at https://www.math.wustl.edu/~feres/Math350Fall2012/Projects/mathproj11.pdf
4. J. Buck, Mazes for Programmers: Code Your Own Twisty Little Passages, Pragmatic Bookshelf, 2015.
5. Y. Okamoto and R. Uehara, “How to make a picturesque maze,” CCCG 2009. August 17-19, 2009.

Appendix 1 - Group Participation

Adam Rayfield

- Code: Modifications to maze simulation and Person class code to implement movement algorithms, minor adjustments to Monte Carlo inputs to use new functions when desired
- Paper: Abstract, Literature Review, Assumptions and Weaknesses, Analysis of changing the search algorithm, Conclusions

Andrew Roberts:

- Code: Binary maze generation, fully-connected maze generation, Monte Carlo simulation
- Paper: Problem, Model, Assumptions and Weaknesses, Implementation, Conclusion

Billy Witrock:

- Code: Base Person Class and Base Maze Simulation. Some wrapper functions to the monte carlo function to gather data for different number of searchers
- Paper: Basic Analysis and Analysis of different number of searchers

Appendix 2 - Code

Person Class:

```
classdef Person < handle
  % person: This is to represent a single person in a maze
  % By Billy Witrock other modified versions by Adam Rayfield
  % and Andrew Roberts (see comments)
  %
  % For modifications to movement algorithm, was saved as a “Person_Memory” class

  properties
    location % current location of person in maze
    visited  % list of places node visited
    %searching %for modified algorithm that searches before stopping, 1 if randomly moving, 0 if
                %preparing to stop
    %stopped %for modified algorithms, 0 if moving, 1 if stopped moving
    %sMem %for modified algorithms, vector representing previous location(s), may track
              %numbers of connections at locations visited
  end

  methods

    % constructor
    % start could be 'r' for random, 's' for start, or 'c' for center
    %   defaults to random
    % size is size of maze
    function obj = Person(start, size)
      switch start
        case 'r'
          obj.location = floor(rand() * size) + 1;
        case 's'
          obj.location = 1;
        case 'c'
          obj.location = round(size/ 2);
        otherwise
          fprintf("Error!, start invalid\n");
          obj.location = floor(rand() * size) + 1;
      end

      obj.visited = [obj.location];
      %obj.sMem = 0; %(2, size) for wait/memory algorithm
      %obj.searching = 1; %For algorithm which searches before stopping/waiting
```

```
        %obj.stopped = 0; %For algorithms which involve stopping/waiting
    end

    % moves to any connected spot with equal probability
    % returns the location of the person after the move
    function moved = move(obj, maze)

        trans_vector = maze(obj.location,:);
        %if obj.sMem ~= 0
                %trans_vector(obj.sMem) = 0;
        %end
        %For prevention of backwards steps algorithm

        %trans_vector(obj.location) = 0;
        %For prevention of stationary steps algorithm

        % random number 1 - number of places could move
        random = floor(rand() * sum(trans_vector,2) + 1);

        % finds the place where it is going to move
        for i = 1:size(trans_vector,2)
            if trans_vector(i) ~= 0
                random = random - 1;
                if random == 0
                    obj.location = i;
                    moved = i;
                    obj.visited = [obj.visited,i];
                        %obj.sMem = obj.location; %For prevention of backwards steps algorithm
                    return
                end
            end
        end

    end

  end
end
```

Maze Simulation:

```
function [num_steps, location, paths] = maze_simulation(maze_creation_type, ...
                                num_searchers, maze_size)
% Maze_simulation: simulates finding a single person lost in a maze
```

```
%      By Billy Witrock other modified versions by
%       Adam Rayfield and Andrew Roberts (see comments)
%
% Note: To be found a searcher must be in the same spot as the lost person
%
%      Also, the searchers and lost person move at same time:
%          So, if the searcher moves from 2 to 3 and the lost person
%          3 to 4, the person would not be found
%
%      Also, multiple searchers can be in the same place
%
% inputs:
%    maze_creation_type:
%        'b' for binary algorithm
%        'f' for fully_connected algorithm
%        defaults to binary
%    num_searchers:
%        the number of people searching for the lost person in the maze
%    maze_size:
%        the number of nodes in the maze
%        Must be a perfect square
%  returns:
%    num_steps: number of steps it took to find the lost person
%    location: the location of where the lost person was found
%    paths: the paths each person took. The top row is the lost person
%          then the following rows are the people looking

%blank returns just in case there is an error
num_steps = [];
location = [];
paths = [];

if num_searchers <= 0
   fprintf("Error: need at least one searchers\n");
   return;
end

% creates all the searchers
searchers = [];
for i = 1:num_searchers
   searchers = [searchers, Person('r',maze_size)];
   %searchers = [searchers, Person_Memory('r',maze_size)];
   %For modifications to movement algorithms, as the wanderer is still assumed to move randomly
```

```
end

lost = Person('r',maze_size);

% creates the maze based on the given algorithm
maze_dim = maze_size .^ (1/2);
maze = [];
switch maze_creation_type
   case 'f'
      maze = gen_fully_connected_maze(maze_dim,maze_dim);
   otherwise
      maze = binary_maze(maze_dim,maze_dim);
end

found = false;

%starts at -1 so the first loop makes the number of steps zero
num_steps = -1;
while ~found
   [found, location] = is_found(searchers,lost);
   num_steps = num_steps + 1;
   for i = 1:num_searchers
      %if searchers(i).stopped == 0 %For algorithms which implement stopping
      move(searchers(i),maze);

      %Below code for the initial-random-search-followed-by-stopping algorithm
      %if searchers(i).searching == 1
          %searchers(i).sMem(1,(num_steps+1)) = searchers(i).location;
          %searchers(i).sMem(2,(num_steps+1)) = sum(maze(searchers(i).location,:));
          %if searchers(i).sMem(1,maze_size) ~= 0
          %        searchers(i).searching = 0;
          %end
          %else
          %Stop searchers after they arrive at a "well-connected" area.
          %if sum(maze(searchers(i).location,:)) == max(searchers(i).sMem(2,:))
          %        searchers(i).stopped = 1;
          %end
          %end
      %else
          %Need to update visited nodes vector outside of 'move' method
          %if stopped == 1
          %searchers(i).visited = [searchers(i).visited,searchers(i).location];
          %end
```

```
    %end %For algorithms which do not implement stopping
  end
  move(lost,maze);
end

% sets path return variable
paths = zeros(1 + size(searchers,2),num_steps + 2);
paths(1,:) = lost.visited;
for i = 1:num_searchers
   paths(1 + i,:) = searchers(i).visited;
end

% elimanates the last column because it is the move after the person is
% found
paths(:,num_steps + 2) = [];

end
```

Monte Carlo Simulation:

```
function [step_list, impact_locations] = monte_carlo_num_searchers...
                    (n_itr, maze_size,num_searchers,maze_type)
  %
  % Monte Carlo simulation: Binary maze, 1 searcher, random starting
  %                 positions, no exits
  % By: Andrew Roberts, slightly modified by Billy Witrock/Adam Rayfield
  %     other modified versions not posted
  %
  % Inputs:
  %     n_itr: Number of iterations in Monte Carlo simulation
  %     maze_size: Number of nodes in maze
  %
  % Outputs:
  %     step_list: Vector containing number of steps taken until impact
  %           for each iteration; dimension: 1 x n_itr
  %     impact_locations: Vector containing counts of impact locations
  %                 for each node; dimension: 1 x maze_size;
  %                 E.g. impact_locations(1) is the count of
  %                 impacts that occurred on the first node
  %

  impact_locations = zeros(1, maze_size);
  step_list = zeros(1, n_itr);
```

```
  for i = 1:n_itr
    [n_steps, loc_impact, path_hist] = maze_simulation(maze_type,...
                                num_searchers, maze_size);

    impact_locations(loc_impact) = impact_locations(loc_impact) + 1;
    step_list(i) = n_steps;
  end

end
```

Binary Maze Generator:

```
%
% binary_maze.m
% Creates random maze via Binary Maze Algorithm
%
% Last Modified: 4/19/2018
% Modified By: Andrew Roberts
%

function [maze] = binary_maze(dim1, dim2)
  n = dim1 * dim2;
  maze = eye(n);

  for i = 1:n
    % Right wall
    if (mod(i, dim2) == 0)
      if i ~= n
        maze(i, i+dim2) = 1;
      end

      if i ~= dim2
        maze(i, i-dim2) = 1;
      end
    elseif i > n - dim2 % Bottom wall
      if i ~= n
        maze(i, i+1) = 1;
      end

      if i ~= n-dim2+1
        maze(i, i-1) = 1;
      end
```

```
    else % Internal Walls
      if rand() > .5
        maze(i, i+1) = 1;
        maze(i+1, i) = 1;
      else
        maze(i, i+dim2) = 1;
        maze(i+dim2, i) = 1;
      end
    end
  end

end
```

Fully Connected Maze Generator:

```
%
% gen_fully_connected_maze.m
% Generates fully connected maze represented as a graph/adjacency matrix
%
% Last Modified: 4/19/2018
% Modified By: Andrew Roberts
%

function [maze] = gen_fully_connected_maze(dim1, dim2)
  n = dim1 * dim2;
  maze = zeros(n, n);

   for i = 1:n
     % Self Connections
     maze(i, i) = 1;

     % Vertical Connections
     if i > dim2
       maze(i, i-dim2) = 1;
     end

     if i <= n-dim2
       maze(i, i+dim2) = 1;
     end

     % Horizontal Connections
```

```
        if mod(i-1, dim2) ~= 0
           maze(i, i-1) = 1;
           maze(i-1, i) = 1;
        end

        if mod(i, dim2) ~= 0
          maze(i, i+1) = 1;
          maze(i+1, i) = 1;
        end
    end

end
```