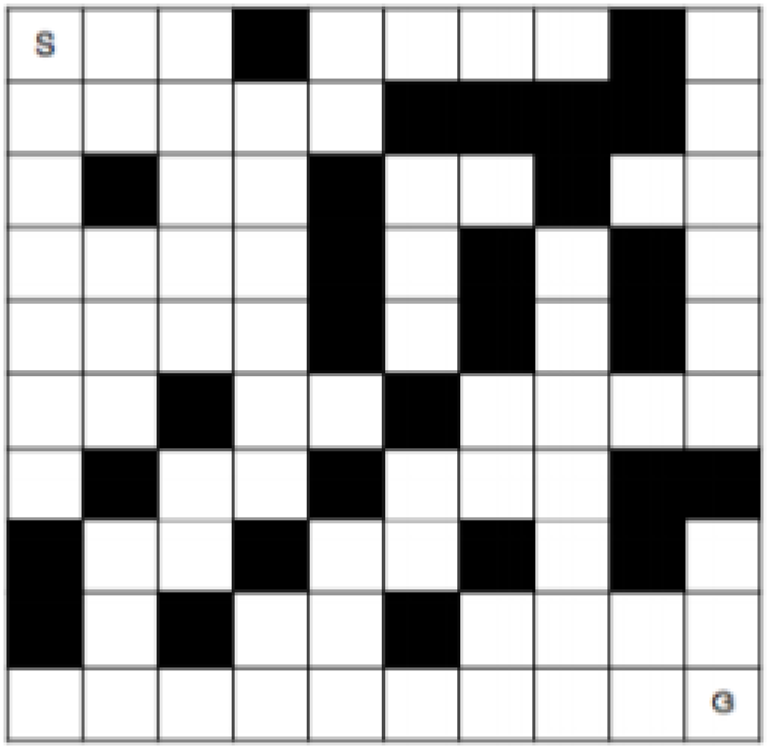
RUTGERS UNIVERSITY

Exploration of Burning Mazes

Prathamesh Kulkarni
October 1, 2020

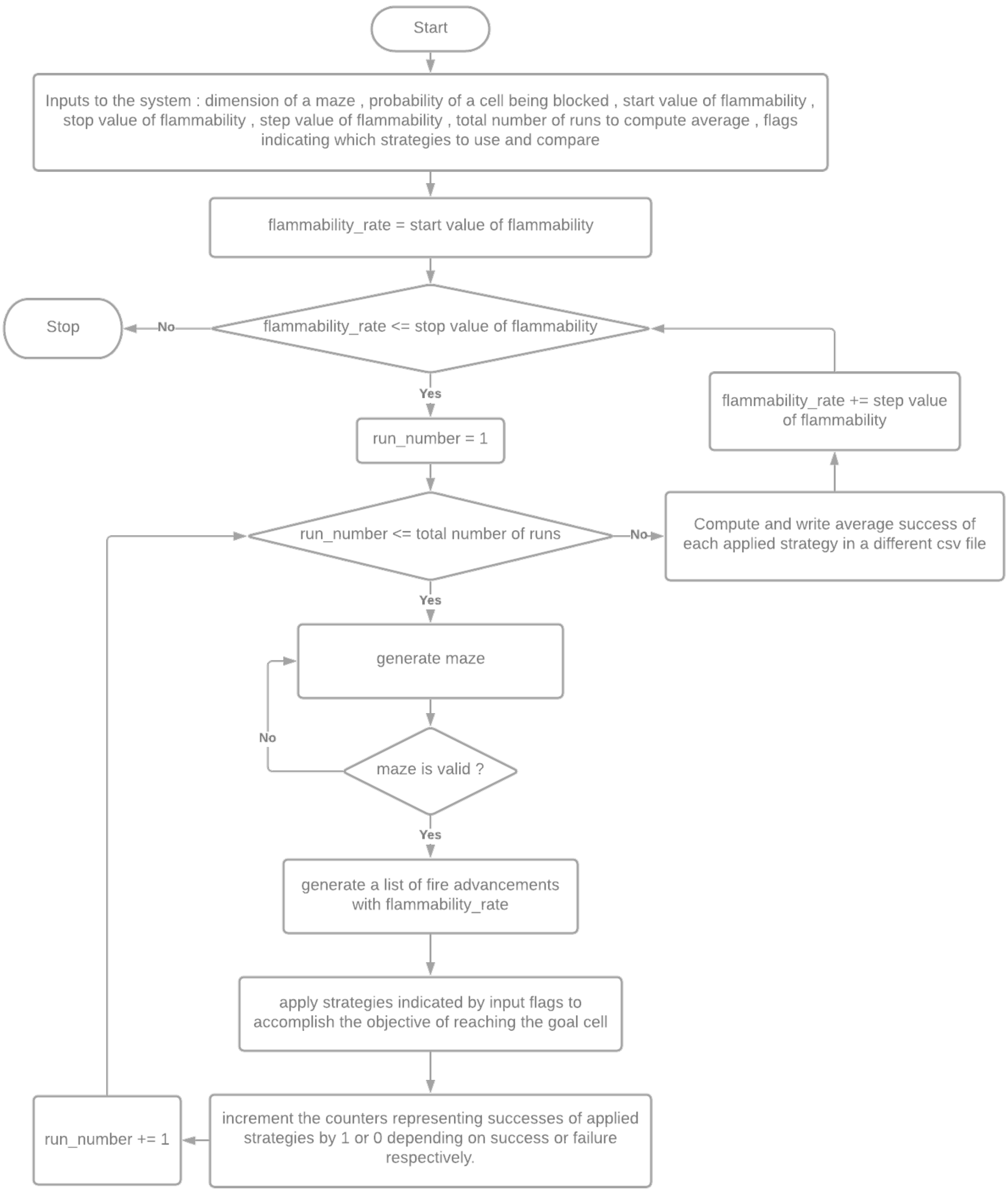
1 INTRODUCTION

Maze is a square grid of cells where each cell is either open, blocked or on fire. Maze has a start cell in the upper left corner and a goal cell in the lower right corner which are always open. Initially, a randomly selected open cell (excluding start cell and goal cell) is set on fire. Agent begins in a start cell and their movement alternates with fire advancement. Objective of the agent is reach the goal cell before getting burnt by fire. Agent has three strategies to accomplish this objective.



2 System Design

2.1 System Architecture


Start
Inputs to the system : dimension of a maze , probability of a cell being blocked , start value of flammability , stop value of flammability , step value of flammability , total number of runs to compute average , flags indicating which strategies to use and compare
flammability_rate = start value of flammability
flammability_rate <= stop value of flammability
No
Stop
Yes
run_number = 1
run_number <= total number of runs
No
Compute and write average success of each applied strategy in a different csv file
flammability_rate += step value of flammability
Yes
generate maze
maze is valid ?
No
Yes
generate a list of fire advancements with flammability_rate
apply strategies indicated by input flags to accomplish the objective of reaching the goal cell
increment the counters representing successes of applied strategies by 1 or 0 depending on success or failure respectively.
run_number += 1

2.2 Important Modules

1. Maze Generation
   It generates the maze of a given dimension and sets the status of every cell to either open or blocked based on a given probability of cell being blocked. It also sets a randomly selected open cell (excluding start and goal cells) on fire and returns the generated maze.

2. Maze Validation
   It ensures the following two aspects of a maze - A) There has to be a path from start cell to goal cell. B) There has to be a path from start cell to a cell initially set on fire. If either or both aspects fail, then the maze is not valid and a new maze is generated. If maze is valid, then system proceeds to Fire Advancement.

3. Fire Advancement
   It generates and saves a complete list of fire advancement steps ahead of time to present identical maze structure and fire behavior to different strategies so as to yield fairer and better comparisons between strategies over shorter runs.

4. Search Algorithms
   A) Uniform Cost Search algorithm is used by Maze Validation to ensure that there is a path from start cell to a cell initially set on fire.
   B) Breadth First Search algorithm is used by Fire Advancement to count the number of reachable open cells in a maze (count is required to compute the fire advancement steps ahead of time).
   C) A* Search algorithm is used by all three strategies to find the shortest path from current position of agent to goal position. A* search algorithm is also used by Maze Validation to ensure that there is a path from start cell to goal cell. Manhattan distance between current position of agent and goal position is used as an admissible heuristic.

5. Strategy 1
   It searches a path from start cell to goal cell only once. Agent keeps on traversing the computed path until - either agent reaches goal cell (success) or agent catches fire (failure). Boolean result indicating the agent's success or failure is returned by strategy 1.

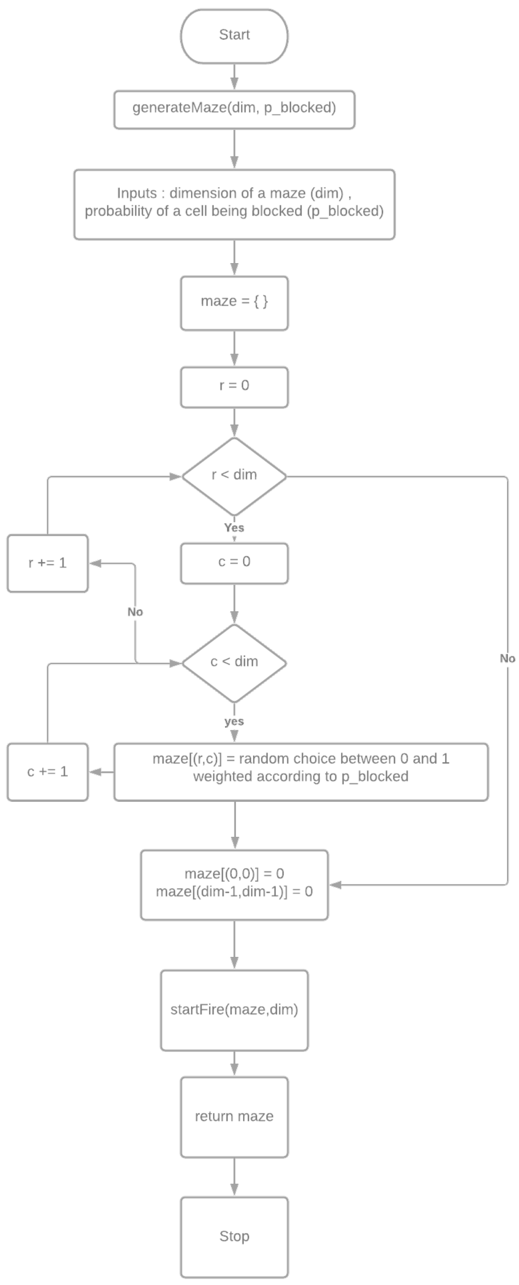
6. Strategy 2
   It repeatedly searches a path from current cell to goal cell. If path is found, it updates current cell of an agent to the first cell in a computed path and discards the remaining path since it computes a new path from updated current cell. Boolean result indicating the agent's success or failure is returned by strategy 2.

7. Strategy 3
   It repeatedly searches a path from current cell to goal cell in a prospective maze with simulated fire advancements generated by using Monte Carlo Simulation. If a path is found, it updates current cell of an agent to a first cell in a computed path and discards the remaining path since it computes a new path from updated current cell. Boolean result indicating the agent's success or failure is returned by strategy 3.

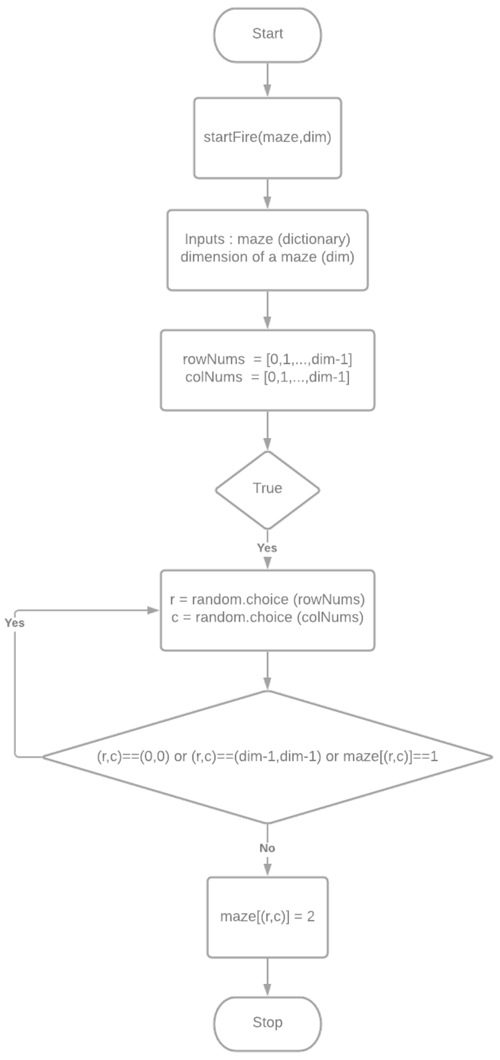
3 Detailed Design

3.1 Maze Generation

Maze is a dictionary of key:value pairs of the form (rowNumber,colNumber) : cellStatus. rowNumbers and colNumbers range from 0 to (dim-1) where dim is the dimension of a maze. Possible values of status are : 0 (cell is open) , 1 (cell is blocked) , 2 (cell is on fire). A nested iteration through all rows and columns sets the status of every cell to either 0 or 1 based on a random choice weighted according to the probability of a cell being blocked. After exiting the nested loop, statuses of start and goal cells are overwritten to 0 as they are always open. A randomly selected open cell (excluding start and goal cells) is set on fire (it's status is changed to 2). The generated maze is then returned to the caller function.



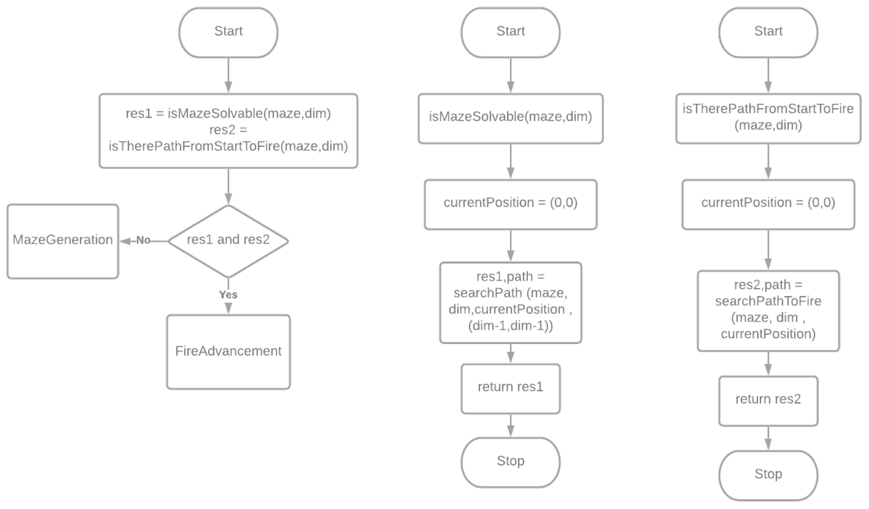
Start
generateMaze(dim, p_blocked)
Inputs : dimension of a maze (dim) , probability of a cell being blocked (p_blocked)
maze = { }
r = 0
r < dim
Yes
No
r += 1
c = 0
No
c < dim
yes
c += 1
maze[(r,c)] = random choice between 0 and 1 weighted according to p_blocked
maze[(0,0)] = 0
maze[(dim-1,dim-1)] = 0
startFire(maze,dim)
return maze
Stop



Start
startFire(maze,dim)
Inputs : maze (dictionary) dimension of a maze (dim)
rowNums = [0,1,...,dim-1]
colNums = [0,1,...,dim-1]
True
Yes
r = random.choice (rowNums)
c = random.choice (colNums)
Yes
(r,c)==(0,0) or (r,c)==(dim-1,dim-1) or maze[(r,c)]==1
No
maze[(r,c)] = 2
Stop

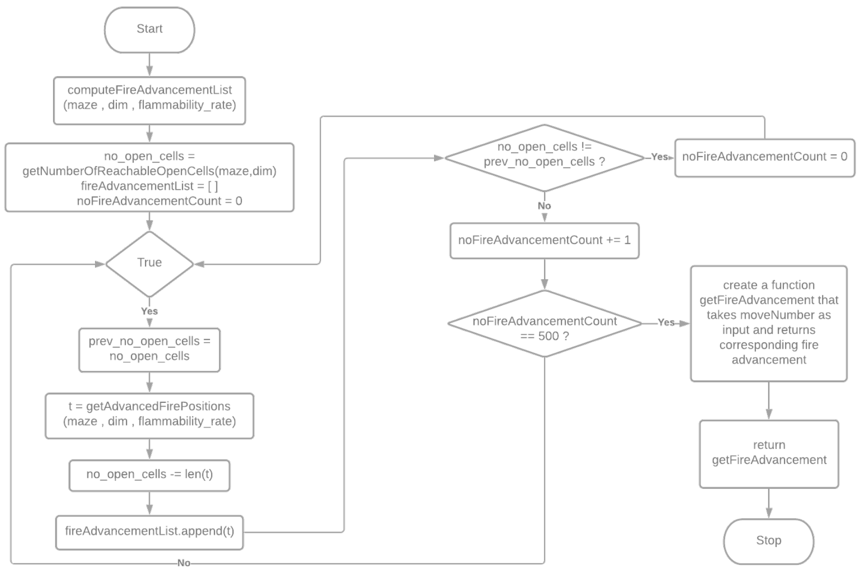
3.2 Maze Validation

The generated maze is validated by searching for two paths. Firstly, A* search algorithm is called to determine if path exists from start cell (0,0) to goal cell (dim-1,dim-1). Secondly, Uniform cost search algorithm is called to determine if path exists from start cell (0,0) to a cell initially set on fire. If boolean results returned by both searches are true (maze is valid), then the system proceeds to the Fire Advancement module. If either or both results returned by searches are false (maze is invalid), then the system backtracks to Maze Generation module.


Start
res1 = isMazeSolvable(maze,dim)
res2 = isTherePathFromStartToFire(maze,dim)
res1 and res2
No
MazeGeneration
Yes
FireAdvancement
Start
isMazeSolvable(maze,dim)
currentPosition = (0,0)
res1,path = searchPath (maze, dim,currentPosition , (dim-1,dim-1))
return res1
Stop
Start
isTherePathFromStartToFire (maze,dim)
currentPosition = (0,0)
res2,path = searchPathToFire (maze, dim , currentPosition)
return res2
Stop

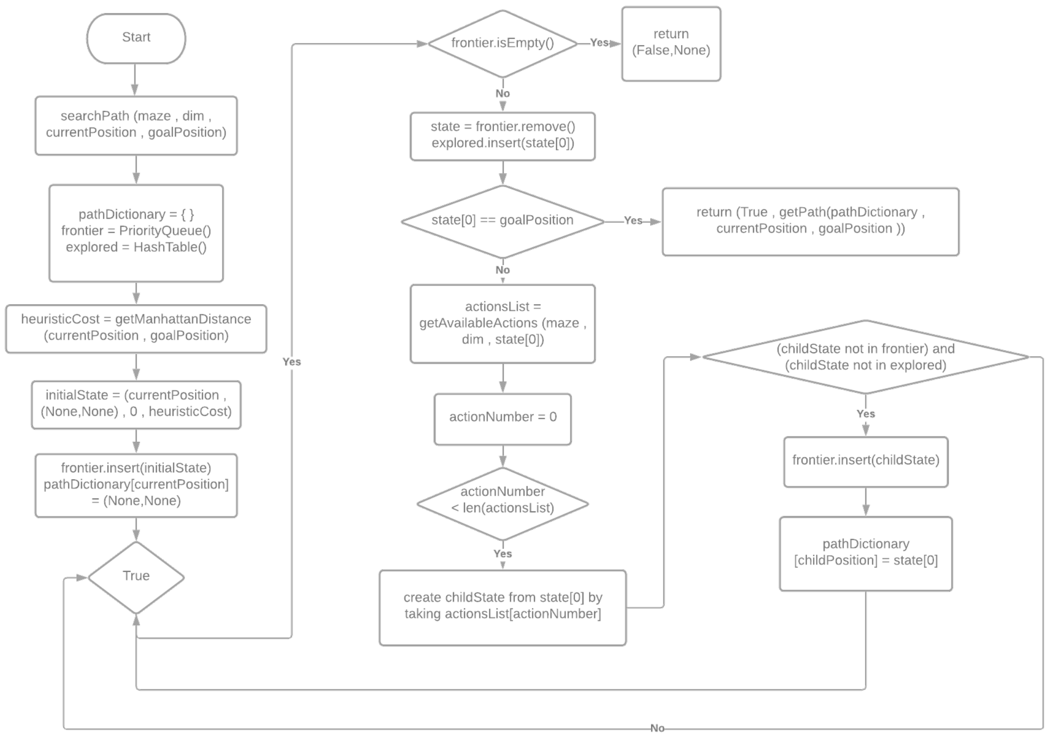
3.3 Fire Advancement

A list of fire advancement steps is computed ahead of time only to present exactly the same maze structure and fire behavior to different strategies so as to yield better comparisons between them over lesser number of runs. Each item of fire advancement list is a tuple of cell positions (row,col) that have been newly set on fire. The module first computes number of open cells reachable from start cell by running breadth first search algorithm. This number represents the upper bound on maximum number of cells that can be set on fire. It then repeatedly makes fire advancements until 500 consecutive advancements have not set any new position on fire. This is used as a terminating condition to indicate that all possible positions have already been set on fire and no new advancements are possible. Probability of this terminating condition being wrong for an open cell with 1 neighbor on fire and with flammability rate = 0.1 is of the order of 10^{-23}. As flammability rate increases and count of burning neighbors increases, probability of terminating condition being wrong approaches 0. The computed list of fire advancements is attached with getFireAdvancement function object using the state retention mechanism of closure function. This makes the fire advancements list accessible only through getFireAdvancement function by passing a current moveNumber.

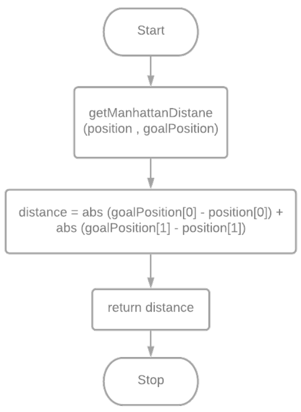

Start
computeFireAdvancementList (maze , dim , flammability_rate)
no_open_cells = getNumberOfReachableOpenCells(maze,dim)
fireAdvancementList = [ ]
noFireAdvancementCount = 0
True
Yes
prev_no_open_cells = no_open_cells
t = getAdvancedFirePositions (maze , dim , flammability_rate)
no_open_cells -= len(t)
fireAdvancementList.append(t)
No
no_open_cells != prev_no_open_cells ?
Yes
noFireAdvancementCount = 0
No
noFireAdvancementCount += 1
noFireAdvancementCount == 500 ?
Yes
create a function getFireAdvancement that takes moveNumber as input and returns corresponding fire advancement
return getFireAdvancement
Stop

3.4 Search Algorithms

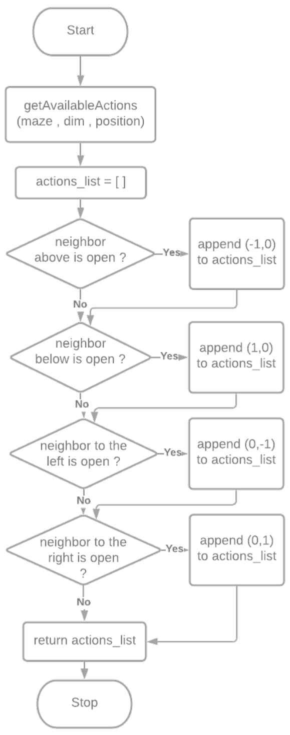
There are three search algorithms : A) A* search algorithm is used by all three Strategies to find the shortest path to goal B) Uniform cost search algorithm is used by Maze Validation to search a path from start cell to a cell initially set on fire . C) Breadth first search algorithm is used by Fire Advancement to count the number of reachable open cells. All three algorithms are implemented as graph-search algorithms. A* search uses a priority queue to store the states to be expanded and a hash table to store the explored states. A* also uses Manhattan distance as a heuristic and orders states in a priority queue by evaluation cost (path cost + heuristic cost). Uniform cost search also uses a priority queue to store states to be expanded and a hash table to store the explored states. Breadth first search uses a FIFO queue to store the states to be expanded and a hash table to store explored states. Dictionary of key:value pairs of the form (rowNumber,colNumber):1 indicate the explored states. List of tuples of the form (position,parentPosition,pathCost,heuristicCost) is used to store priority queue.



Start
searchPath (maze , dim , currentPosition , goalPosition)
pathDictionary = { }
frontier = PriorityQueue()
explored = HashTable()
heuristicCost = getManhattanDistance (currentPosition , goalPosition)
initialState = (currentPosition , (None,None) , 0 , heuristicCost)
frontier.insert(initialState)
pathDictionary[currentPosition] = (None,None)
True
Yes
frontier.isEmpty()
Yes
return (False,None)
No
state = frontier.remove()
explored.insert(state[0])
state[0] == goalPosition
Yes
return (True , getPath(pathDictionary , currentPosition , goalPosition))
No
actionsList = getAvailableActions (maze , dim , state[0])
actionNumber = 0
actionNumber < len(actionsList)
Yes
create childState from state[0] by taking actionsList[actionNumber]
(childState not in frontier) and (childState not in explored)
Yes
frontier.insert(childState)
pathDictionary [childPosition] = state[0]
No



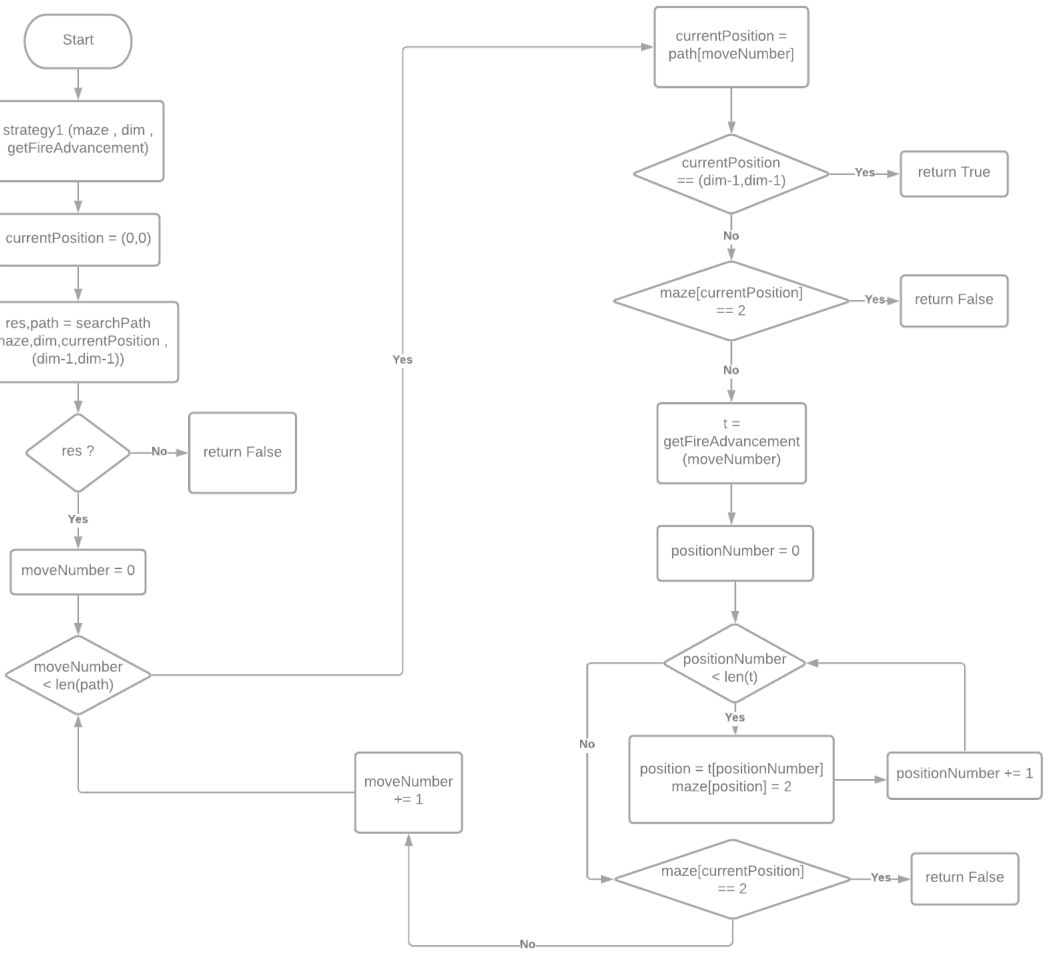
Start
getManhattanDistane (position , goalPosition)
distance = abs (goalPosition[0] - position[0]) + abs (goalPosition[1] - position[1])
return distance
Stop



Start
getAvailableActions (maze , dim , position)
actions_list = [ ]
neighbor above is open ?
Yes
append (-1,0) to actions_list
No
neighbor below is open ?
Yes
append (1,0) to actions_list
No
neighbor to the left is open ?
Yes
append (0,-1) to actions_list
No
neighbor to the right is open ?
Yes
append (0,1) to actions_list
No
return actions_list
Stop

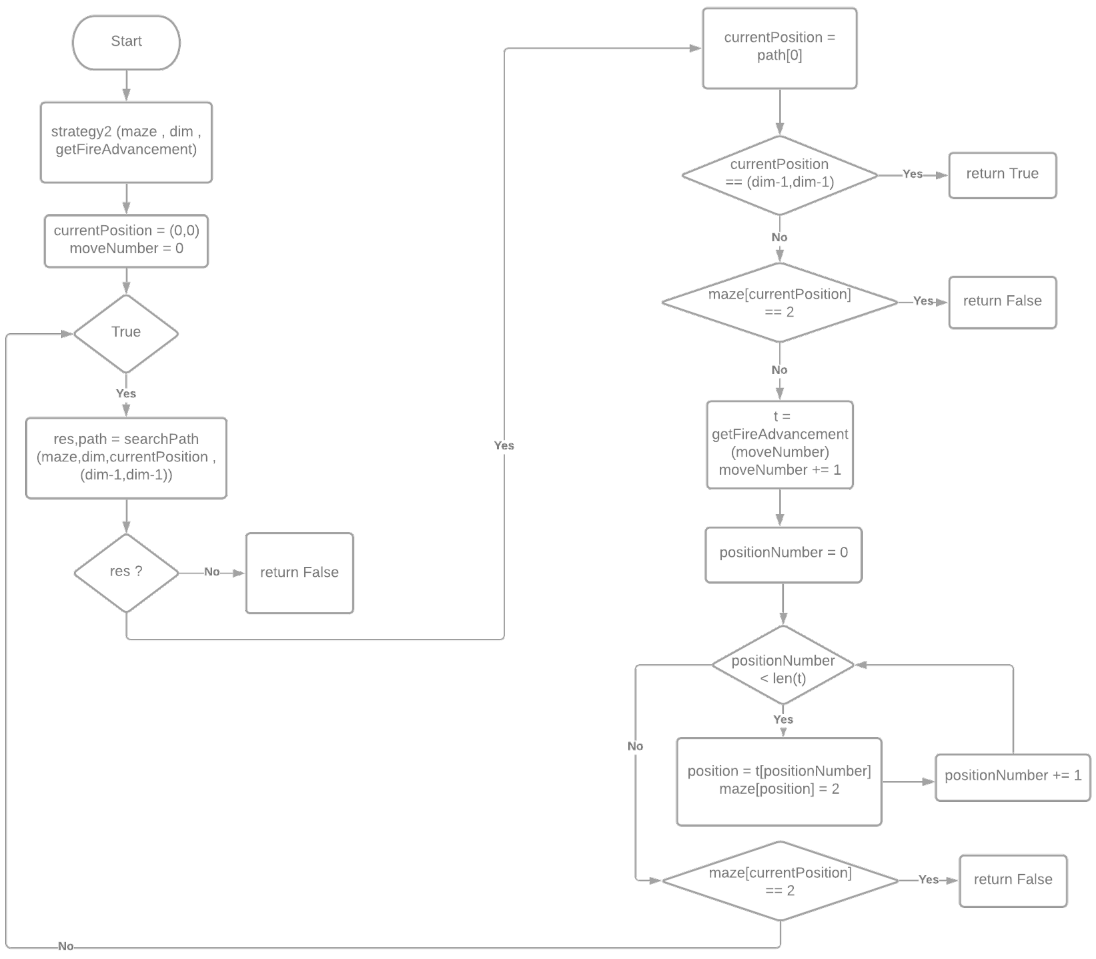
3.5 STRATEGY 1

It sets current position to a start cell and searches a path from current position to goal position. If a path is not found, it returns failure. If a path is found, it traverses the path until either agent reaches goal position or agent catches fire. Each move of an agent alternates with fire advancement. Fire advancement after a particular move is obtained from getFireAdvancement function by passing a move number.


Start
strategy1 (maze , dim , getFireAdvancement)
currentPosition = (0,0)
res,path = searchPath (maze,dim,currentPosition , (dim-1,dim-1))
res ?
No
return False
Yes
moveNumber = 0
moveNumber < len(path)
Yes
currentPosition = path[moveNumber]
currentPosition == (dim-1,dim-1)
Yes
return True
No
maze[currentPosition] == 2
Yes
return False
No
t = getFireAdvancement (moveNumber)
positionNumber = 0
positionNumber < len(t)
Yes
position = t[positionNumber] maze[position] = 2
positionNumber += 1
No
maze[currentPosition] == 2
Yes
return False
No
moveNumber += 1

3.6 Strategy 2

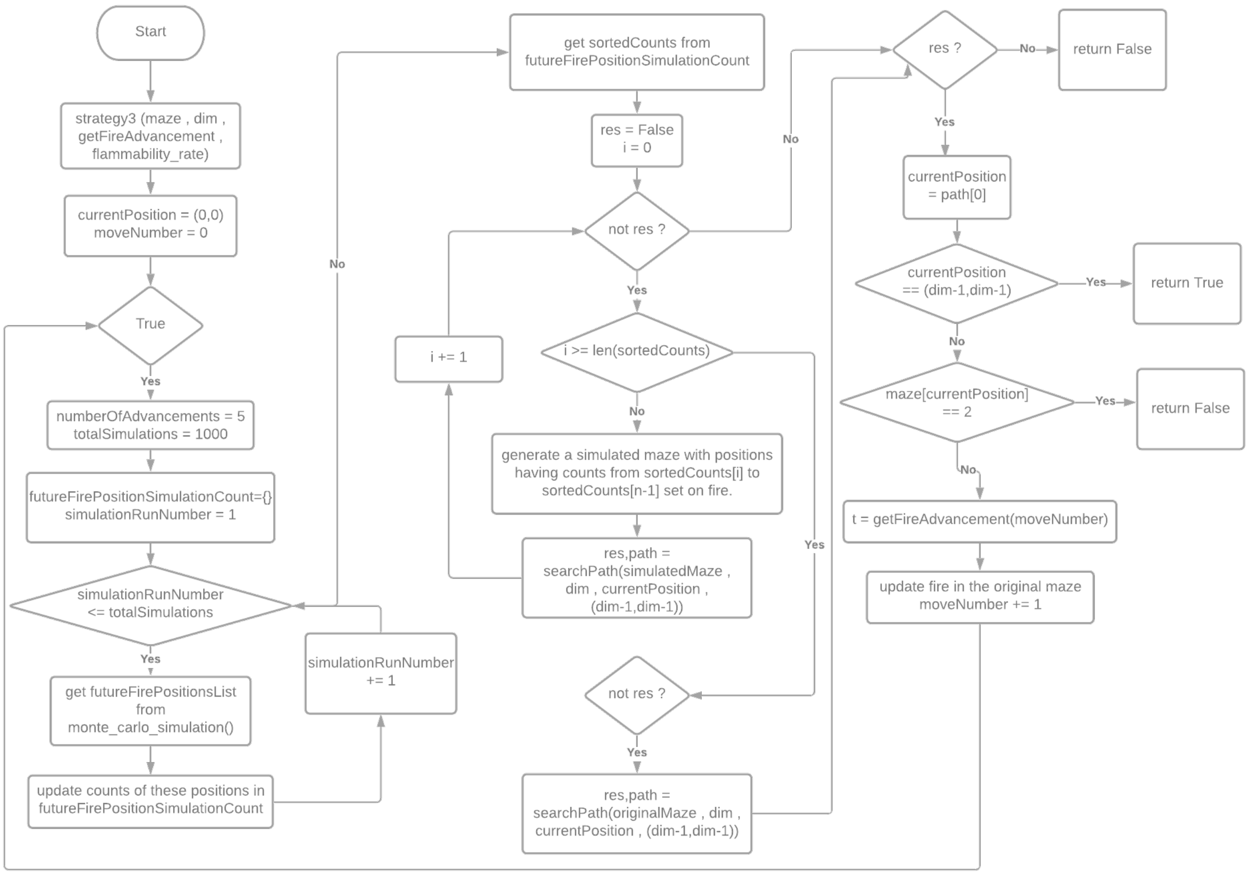
It sets current position to a start cell and repeatedly searches a path from current position to goal position. If path is not found, it returns failure. If path is found, it updates current position to the first position in path and discards the remaining path since it recomputes the path from updated current position. Each move of an agent is alternated by fire advancement. Fire advancement after a particular move is obtained from getFireAdvancement function by passing a move number.



Start
strategy2 (maze , dim , getFireAdvancement)
currentPosition = (0,0) moveNumber = 0
True
Yes
res,path = searchPath (maze,dim,currentPosition , (dim-1,dim-1))
res ?
No
return False
Yes
currentPosition = path[0]
currentPosition == (dim-1,dim-1)
Yes
return True
No
maze[currentPosition] == 2
Yes
return False
No
t = getFireAdvancement (moveNumber) moveNumber += 1
positionNumber = 0
positionNumber < len(t)
Yes
No
position = t[positionNumber] maze[position] = 2
positionNumber += 1
maze[currentPosition] == 2
Yes
return False
No

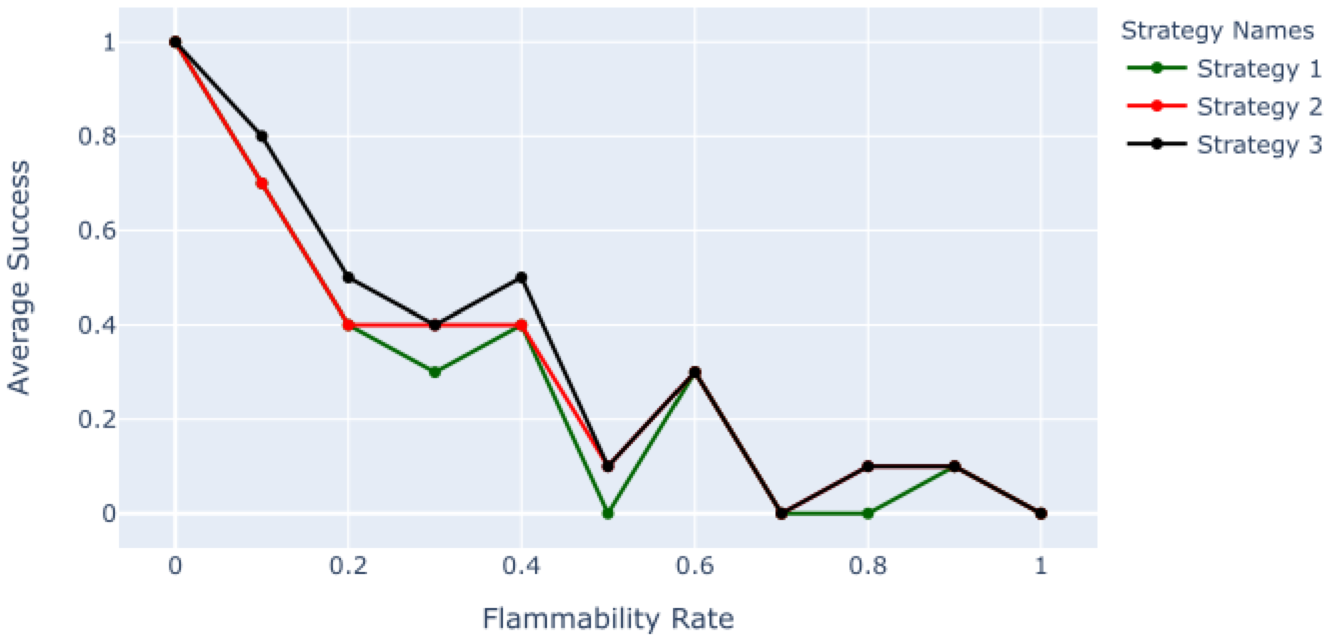
3.7 STRATEGY 3

It takes into account potential future states of fire by using Monte Carlo Simulation. Strategy 3 has two hyperparameters : A) Number of fire advancements to take into account B) Number of simulations to run. One simulation performs a given number of fire advancements with a given flammability rate and returns a list of positions that have newly been set on fire in a simulated maze. A dictionary is used to store key:value pairs of the form position:count (number of times a position has been set on fire throughout all simulations). A list (sortedCounts) of these counts is sorted in ascending order and is used to repeatedly create a prospective maze with simulated fire positions. The positions already on fire in the original maze are kept on fire in a simulated maze as well. First simulated maze has all positions with counts from sortedCounts[0] to sortedCounts[n-1] on fire. Similarly, ith simulated maze has all positions with counts from sortedCounts[i-1] to sortedCounts[n-1] on fire. If a path is not found in ith simulated maze, (i+1)th simulated maze is generated. If a path is found in ith simulated maze, then the current position is updated to the first position in computed path and remaining path is discarded as new simulations will be generated and new path will be computed from an updated current position. If at any point of time, a path is not found in all n simulated mazes, then it just computes a path in original maze with no fire simulations and behaves like strategy 2. This ensures that strategy 3 necessarily performs as good as strategy 2. The only cases where strategy 3 may perform worse than strategy 2 is when simulated fire positions are misleading and a path is found in such simulated maze.
Acronym for strategy 3 is **RISE** - **R**un through **I**mminent maze by **S**imulating **E**nvironment.

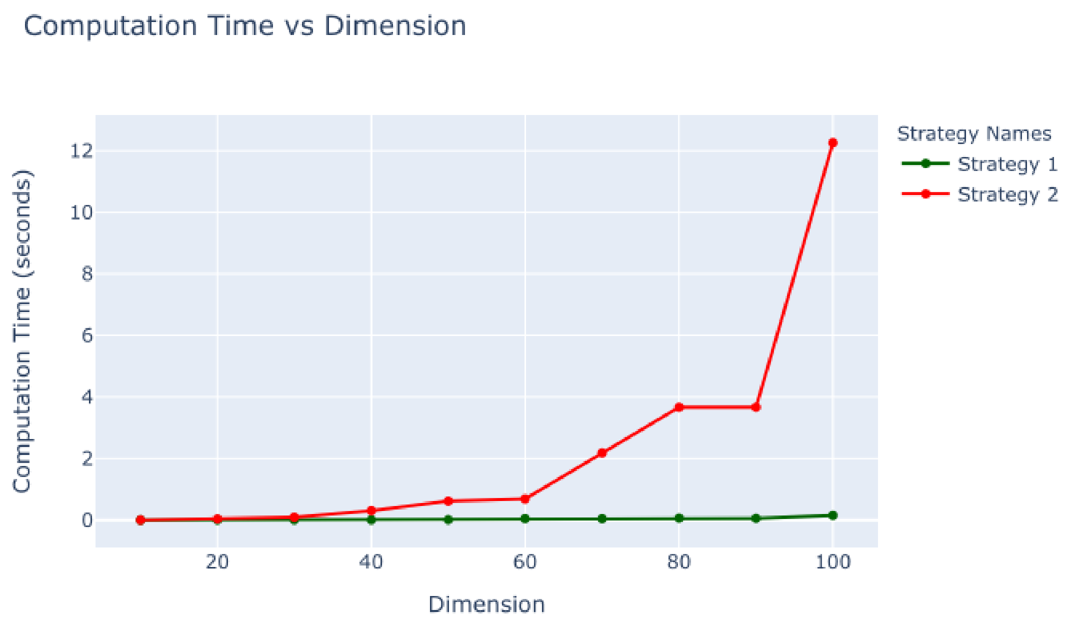

Start
strategy3 (maze , dim , getFireAdvancement , flammability_rate)
currentPosition = (0,0) moveNumber = 0
True
Yes
numberOfAdvancements = 5 totalSimulations = 1000
futureFirePositionSimulationCount={} simulationRunNumber = 1
simulationRunNumber <= totalSimulations
Yes
get futureFirePositionsList from monte_carlo_simulation()
update counts of these positions in futureFirePositionSimulationCount
simulationRunNumber += 1
No
get sortedCounts from futureFirePositionSimulationCount
res = False i = 0
not res ?
Yes
i >= len(sortedCounts)
No
generate a simulated maze with positions having counts from sortedCounts[i] to sortedCounts[n-1] set on fire.
res,path = searchPath(simulatedMaze , dim , currentPosition , (dim-1,dim-1))
i += 1
Yes
not res ?
Yes
res,path = searchPath(originalMaze , dim , currentPosition , (dim-1,dim-1))
No
res ?
No
return False
Yes
currentPosition = path[0]
currentPosition == (dim-1,dim-1)
Yes
return True
No
maze[currentPosition] == 2
Yes
return False
No
t = getFireAdvancement(moveNumber)
update fire in the original maze moveNumber += 1

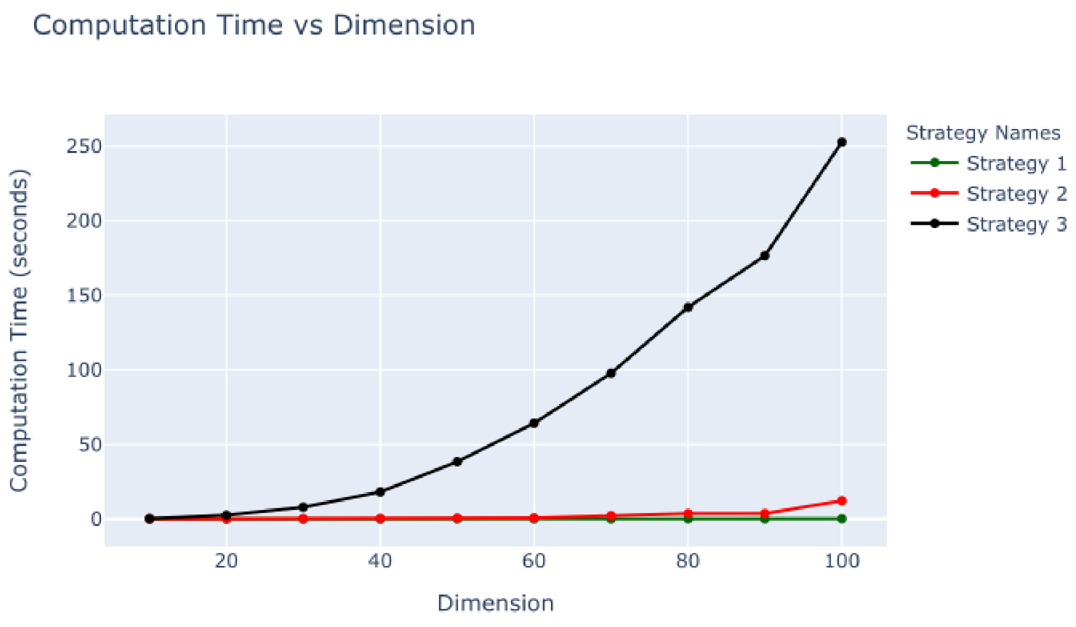
4 Strategic Comparisons

4.1 Average Successes



This is a line chart showing flammability rate of a maze on x-axis and average success of all three strategies on y-axis. Average success has been computed for mazes with dimension 8 and blocked density 0.3 over 10 runs . Average success of strategy 3 has been computed with static hyperparameters: number of advancements = 10 and total simulations = 10. Since maze validation step discards unsolvable mazes, all three strategies achieve perfect average success for flammability rate = 0 (no fire advancement). As flammability rate increases, the average successes of strategies begin to decrease. Because the averages are computed over lesser number of runs, they do not exhibit a consistently decreasing pattern (averages will converge in probability to the true expectations when large number of trials are performed). For flammability rates 0.1, 0.2 and 0.4, Strategy 3 has achieved more successes in comparison to strategies 1 and 2 by factoring future fire positions using Monte Carlo Simulation. At other flammability rates, strategy 3 has necessarily performed as good as strategy2 when it failed to find paths in simulated mazes but succeeded in the actual mazes. It can also be observed that the obtained comparison is better and fairer as all three strategies have computed average successes over identical maze structures and fire advancements.





These are the line charts showing dimension of a maze on x-axis and time taken to reach the goal cell from start cell in seconds for all three strategies on y-axis. Computation times are measured for the mazes of varying dimensions with flammability rate of 0.1 and blocked density of 0.3 . First image is showing same lines of strategy 1 and 2 from the second image in a magnified form by omitting strategy 3.Time taken by strategy 3 has been computed with static hyperparameters: number of advancements = 5 and total simulations = 10. Strategy 1 takes the least amount of time as it runs A* search algorithm only once. Strategy 2 takes slightly more time than strategy 1 as it runs A* search algorithm at least $2 * dim$ times. Strategy 3 takes substantially more time than other two strategies because it performs 5 steps of fire advancements 10 times for an agent to make one move. In strategy 3, agent runs A* search algorithm at most $len(sortedCounts)$ times in order to make one move. This suggests that strategy 3 can perform better in terms of computation time if the two hyperparameters are changed dynamically based on maze structure, flammability rate and proximity of fire. With greater flammability rates, lesser number of fire advancement steps need to be simulated to obtain prospective fire positions. Proximity of fire also plays a role in deciding the number of advancements but obtaining proximity information requires additional search. To run more simulations in reasonable computation time, distributed computing can be used where multiple worker nodes simulate the fire advancements in parallel.

5 Bottlenecks and several ways to manage them

Following bottlenecks have been observed in data generation:

1) Making sure the current maze is solvable:
As the dimension of a maze increases, the amount of time required to search a path from start cell to goal cell also increases. Computation time can be reduced by using bidirectional breadth first search that reaches only half of the depth reached by A* search algorithm.

2) Making sure there is a path from agent to the initial fire source:
Computation time of this search can also increase with the dimension of a maze when initial fire is located further away from a start cell. If initial position of a fire is recorded, then bidirectional breadth first search can speed-up the search.

Computation time of above two maze validations can be reduced further by searching for goal cell as well as initial position of fire in a single scan of search space (maze).

3) Shortest path computation:
Computing a path from start cell to goal cell can be possibly accomplished in lesser time by depth-first search that loses optimality of a path. As maze is constantly burning down, it is advantageous to take a shortest path to the goal cell. Computation of shortest path is expensive where use of informed search instead of uninformed search yields better search times. For the same reason, this system uses A* search algorithm in all three strategies.

4) Fire advancement in a maze:
Fire advancement requires counts of burning neighbors for all the open cells in a maze. Scanning through the maze ($O(dim^2)$) every time to compute the counts becomes expensive as the dimension of a maze increases. This can be handled as follows - while agent is searching for a path, counts of neighbors can be computed parallely by means of multi-threading.

5) Number of fire advancement steps to simulate:
Fire advancement is computationally expensive step as it requires scanning through all the cells in a maze to update fire positions. Computation time of this stage can be reduced to some extent by keeping a list of currently open cells and scanning only through them.

6) Number of simulations to run:
Running a simulation is also computationally expensive as it involves k fire advancement steps. Monte carlo simulation yields better average successes when more random samplings of future fire positions are obtained. On the other hand, more number of simulations increase the computation time significantly when running in local mode. Distributed computing can play a key role in performing more simulations by parallelization. Multiple worker nodes can simultaneously perform simulations and return their results to the master node (agent).

7) Computing vicinity of a fire:
Information about the vicinity of a fire can guide agent in deciding hyperparameters of strategy 3 but obtaining that information requires additional search operation in a maze.

6 Conclusion

Strategy 2 outperforms strategy 1 by recomputation of a path in an updated environment. Strategy 3 is expected to outperform strategy 2 (and hence strategy 1) when large number of simulations have been performed to yield consistent results about future fire positions.
Strategy 1 is computationally least expensive. Strategy 2 is slightly more expensive than strategy 1 due to recomputation. Strategy 3 is computationally most expensive due to the fire advancements and simulations. When agent has no time limitations, they can prefer strategy 3 over strategy 2. When agent has time limitations, then strategy 2 and strategy 3 with fewer number of advancements and simulations seem to pose commensurate benefits. When agent has to take an action instantly, strategy 1 is the only viable option.

7 Future Work

1) Perform two maze validation steps in a single scan of a maze.
2) Implement multi-threading to count the number of burning neighbors in parallel while agent searches for a path.
3) Use apache spark to perform multiple simulations of fire advancements in a cluster.

8 References

1) https://en.wikipedia.org/wiki/A*_search_algorithm
2) https://en.wikipedia.org/wiki/Monte_Carlo_method
3) https://en.wikipedia.org/wiki/Dijkstra
4) https://en.wikipedia.org/wiki/Breadth-first_search
5) https://plotly.com/python/line-charts/
6) https://www.oreilly.com/library/view/learning-python-5th/9781449355722/