INDIAN INSTITUTE OF TECHNOLOGY, DELHI

SIMULATION REPORT

The Plight of Pinky

ARNAV TULI | ENTRY NO. 2019CS10424

APAR AHUJA | ENTRY NO. 2019CS10465

Course - COP290 | Prof. Rijurekha Sen

Compiled on May 21, 2021

Contents

Chapter 1

Problem Statement

1.1 Onset:

Once upon a time in a world full of magical creatures, there lived a family of ghosts. The family consisted of **Blinky**, **Pinky**, **Inky** and **two** **Clydes**. They lived peacefully in their *Ghost House*, which was situated somewhere in the middle of the *Ghost Town* (a large square region).
One day, when the family members were carrying out their usual activities, they heard a loud thunderous sound, and the ground beneath them started shaking. Both confused and scared, the ghosts hurried outside their home to get a better idea of what was happening. To their surprise, they saw large walls built all around the place. It seemed as if they came from underneath the ground, thus explaining the shaking of the surface. It was as if the entire *Ghost Town* was converted into a **gigantic maze** by some heavenly force. **Blinky** being quite stubborn was not ready to accept what he was just witnessing, and therefore, decided to go out and investigate further. Deep enough in the maze, he discovered some *large, glowing round objects* (gems). Not knowing what the gems were exactly, he decided to continue forward with his investigation. Not very far from the point where he saw the first gem, he heard a sound, *OM NOM! OM NOM!*. He looked around the corner and could not believe what he saw! It was a large, round yellow-coloured beast, a Pacman! Terrified, he decided to return back home before he gets caught and report his findings. On his way home, just some distance away, he was suddenly overpowered by a **strong** presence of fear. His movements also slowed down, and he was unable to get any nearer to his home. Before he could look back and see what was actually happening, there was a bright flash of light, and the next instant he found himself back at his home, surrounded by his family.
Sharing his experiences with his family, Blinky experienced some weakness. Meanwhile, Pinky, who was watching Blinky as he was returning, remembered seeing a glimpse of the Pacman with the gem, moments before there was a bright flash. Realising that the gem was a key factor behind what happened and also noting that the after-effects of that incident were not good, as Blinky was feeling weaker, she decided to explore these gems even further. Hence, she decided to go out to investigate the gems at night, as she believed that the Pacmen must be asleep at night.

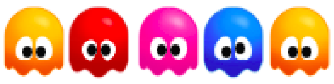


Figure 1.1: The Ghost Family

1.2 Pinky's Adventure:

At night Pinky went out and looked for the exact same spot where there was a bright flash before. To her surprise, there was a wiggly-looking **Portal**. She had never used a portal before, but she was smart enough to know that these portals were not trust-worthy. However, noting that these portals may actually be the key to know more about the **mysterious gems** and *Pacmen* themselves, she decided to take the risk, and *jumped* into the portal. Fortunately, she was teleported to somewhere inside *Ghost Town*. Unexpectedly, she also found a handy **radar** available with her that showed her the locations of the gems (**six** of them) as well as the **entire maze**, including their home. She was both relieved and stressed. Assuming that this device belonged to one of the **Pacmen**, it will not be long before they find out that one of the radars is missing. At the same time, she now knows the locations of all the gems as well as her home. Fortunately, every location in *Ghost Town* is reachable from her current location. Hence, she can try collecting all the gems and return home safely before any mishap takes place. She realised that she has to be quick in doing all of this and hence needs to choose a path that is as short as possible between her current location and her home that also includes all the gems' locations. At the same time, she needs to make these decisions as quickly as possible. Thankfully, Pinky knows a bit about *Graph Theory* herself and believes that it is a doable job.

1.3 Computational Model and Formal Statement:

In order to help **Pinky** with her ordeal, we try to formulate her problem mathematically, and determine some solution.

1.3.1 Maze Representation:

Given that the *Ghost Town* is large square confined region, we can represent the town in the form of **N x N** 2D Array, **MAZE**. Where each entry **MAZE[i][j]** represents a location in the real world *Ghost Town*. MAZE[0][0] represents the top-left corner of the town, when seen from the top. Similarly, MAZE[N-1][N-1] represents the bottom-right corner when viewed from the top. Also, since each entry/block can have four possible connections:- UP, RIGHT, DOWN and LEFT, we provide these attributes to every entry of the maze. These attributes can take the value of *ALLOWED* if there is no wall or *DENIED* in case a wall is present. For example, if there is a wall between MAZE[1][0] and MAZE[0][0], we assign *DENIED* to MAZE[1][0].UP and MAZE[0][0].DOWN.

1.3.2 Various Coordinates:

- Starting Coordinate is (startX, startY), such that its location in maze is MAZE[startX][startY]
- Coordinates of *Ghost House* are (endX, endY), such that its location in maze is MAZE[endX][endY]
- Coordinates of the **six** gems are represented by an array *locations*, such that the location of i^{th} gem is MAZE[locations[i].x][locations[i].y]

1.3.3 Statement:

Given the maze in the form of a 2D Array (MAZE) and locations of all important points, which include (startX, startY), (endX, endY) and *locations* (gem locations), determine a path which starts from (startX, startY) and ends at (endX, endY) which covers all the locations indicated by *locations*.
Constraints:

- **Every location of the Ghost Town is reachable from (startX, startY) in MAZE**
- **Pinky moves with a constant speed**
- **Pinky may not be a human, but she is not a Robot either. Hence, she cannot process loads of data without slowing herself down. Thus, the algorithm should also be feasible and should take Pinky's memory into consideration.**
- **The path should be as short as possible**
- **The algorithm used for computing this path should take as less time as possible**

Note: Given that Pinky has a radar available with her, the solution algorithm may assume that she has the coordinates of each and every gem as well as her home. In other words, she also has a Third Person View of the maze. (Hint: Pinky can perform BFS)

Chapter 2

Algorithm

2.1 Optimal Algorithm

For the given problem, knowing that *Pinky* has access to the entire maze (thanks to the radar!), we can actually think of an optimal algorithm that will guarantee that **Pinky** chooses the shortest path for her travel. This algorithm's **pseudo code** has been described below:

```
START
// Assert:-
// 2D Array/Vector: MAZE[0..N-1][0..N-1]
// Array: locations[0..5]
// Position of Gem #i := locations[i] ^ 0 <= i <= 5
// Pinky's initial position, start := (startX, startY)
// Pinky's home position, end := (endX, endY)

// 0-indexed
Define vector: Points := {start, locations[0], .., locations[5], end}
Define 2D Array: Distance[0..N-1][0..N-1] := All (-1)
// used to store distance between pair of locations in MAZE, indexed by pair
Define HashMap: PairSeparation := {}
index := 0
// INV: PairSeparation stores pair-wise distance between positions in MAZE,
// which have one of the end points in Points[0..index - 1], 0 <= index <= 7
while (index < 7) {
    // perform BFS on MAZE, with Points[index] as the starting node
    // BFS stores the distance of each point from Points[index] in Distance Array
    BFS(Points[index], MAZE, Distance)
    // INV: Pairwise distances of {Pair(Points[index], Points[index + 1]), ..,
    // Pair(Points[index], Points[i - 1]} have been added to PairSeparation,
    // index + 1 <= i <= 8
    for (i := index + 1; i < 8; i := i + 1) {
        // use Distance Array to compute distance
        temp := Distance[Points[i].first][Points[i].second]
        // Add unordered pair to HashMap
        PairSeparation.Add(Pair(Points[index], Points[i]), temp)
    }
    index := index + 1
}
```

```
// all distance calculation done
// choose shortest path, starting from (startX, startY)
Define set: P := Set of all paths of the form (start, l1, l2, .., l6, end),
where, l1, l2, .., l6 belong to locations[0..5], and li != lj whenever i != j

Num := |P|  // Assert:- cardinality of P = 6! = 720
optimumDistance := infinity
optimalPath := ()

// INV: optimalPath stores the most optimal path among all paths in P,
// which have been iterated over by the for loop.
for all p := (a0, a1, a2, .., a6, a7) in P:
    distance := 0
    // INV: distance = pathLength(a0, a1, ... ai). 0 <= i <= 7
    for all i in 0 to 6:
        temp := PairSeparation[Pair(a(i), a(i + 1)]
        distance := distance + temp
    end for
    // Assert: distance = pathLength(a0, a1, .., a7)
    // update optimal path if necessary
    if (distance < optimalDistance) {
        optimalDistance := distance
        optimalPath := p
    }
end for

Send to Pinky: Pair(optimalPath, optimalDistance)

END
```

On observing this algorithm, we notice that it is fairly straightforward and its correctness follows from the comments, connectivity of MAZE and the correctness of the BFS used in the implementation. It just uses standard *Breadth First Search* algorithm in an undirected, unweighted graph MAZE and determines pairwise distance. Next, using this pairwise distance, it goes through all the possible permutations of order in which *Pinky* visits the Gems and chooses the path with the least distance. But how much time will *Pinky* need to figure this all out? We have tried to analyse the time requirement in the following section.

2.2 Time complexity (Optimal):

The graph being represented by a 2D Array MAZE[0..N-1][0..N-1], consists of a total of N^2 nodes, or locations. Also, since each location in MAZE can have only 4 edges incident to it at maximum, the total number of edges is also bounded above by $4N^2$. Hence, the runtime complexity of standard **BFS algorithm** becomes $\boldsymbol{O(V + E) = O(N^2)}$.
This is an asymptotic relation (as N tends to ∞). However, for our problem, it is safe to assume that the time required by *BFS* is $\boldsymbol{cN^2}$, where **c** is some positive real constant. Now, in the

algorithm, *Pinky* needs to perform BFS a total of 7 times. Hence, her total time requirement (for BFS) is $\boldsymbol{7cN^2}$. Next, she needs to analyse every permutation of order of visit that is possible (720 of them). Assuming, that she has a large amount of memory and can easily recall stuffs without wasting too much time, she will requires just some constant amount of time (say **d**) to determine the distance of a particular path. Since, there are a total of 720 paths, she requires time $\boldsymbol{720d}$.
All the analysis till now considers *Pinky* to be a Robot, in the sense that she can process data without introducing significant delays. However, from Point 4 (section 1.3.3), we know that this is not the case. Now, in our algorithm, *Pinky* needs to remember a single BFS queue at a time, she also needs to remember the **Distance** Array (size N^2), **PairSeparation** Map (size 28) and all of **720** paths possible. Assuming that each piece of information that *Pinky* stores introduces a constant delay (say **f**) and knowing the fact that each of 720 paths can be as big as $\boldsymbol{7N^2}$ and BFS queue as big as $\boldsymbol{N^2}$, the total information/recall delay in the worst case is $\boldsymbol{(N^2 + 28 + 720(7)N^2 + N^2) * f = 5042fN^2 + 28f}$. Collecting all the delays together, we get the total time requirement of the algorithm := $\boldsymbol{(7c + 5042f)N^2 + 720d + 28f}$
Note that this is just the time *Pinky* needs to come up with the solution. It does not include her travel time. Now say that the length of the shortest path is $\boldsymbol{l}$. Then knowing that *Pinky* can only move at a fixed speed, the time required by her (for travelling) will be $\boldsymbol{tl}$, where $\boldsymbol{t}$ is a constant. Combining this with total time requirement of algorithm, we get:

$$\textbf{Time Required by Pinky, } \boldsymbol{T_1 := (7c + 5042f)N^2 + 720d + 28f + tl}$$

Is $\boldsymbol{T_1}$ less enough for *Pinky* to reach back home without getting caught? Or is it just too high to be infeasible in the current scenario? We plan to investigate this by analysing another approximate algorithm/solution.

2.3 Approximate Algorithm:

In this section, we describe an alternative approach to the given problem that tries to determine a path (not necessarily optimal) between the starting and ending locations (covering all the gems' locations) which might be more feasible in the given context, in the sense that *Pinky* will require lesser time than that required in the *Optimal Algorithm* (section 2.1). We describe its **pseudo code** below:

```
START
// Assert:-
// 2D Array/Vector: MAZE[0..N-1][0..N-1]
// Array: locations[0..5]
// Position of Gem #i := locations[i], 0 <= i <= 5
// Pinky's initial position, start := (startX, startY)
// Pinky's home position, end := (endX, endY)
currentLocation := start
// INV: Pinky has i gems after ith iteration. Pinky is at currentLocation.
while (locations.isNotEmpty) {
    // perform partialBFS on MAZE, with currentLocation as the starting node
    // partialBFS returns the element in locations closest to the
    // currentLocation along with the path from currentLocation to the element
    (index, path) := partialBFS(currentLocation, MAZE)
    // send path to Pinky
    Send To Pinky: path
    Wait For Pinky to reach her destination
    currentLocation := locations[index]
    // delete this element from locations
    locations.DeleteElementbyIndex(index)
}
// Assert:- Pinky has all 6 gems. Now, move towards home.
locations.AddElement(end)
(index, path) := partialBFS(currentLocation, MAZE)
// send final path to Pinky
Send To Pinky: path
END
```

This algorithm tries to find the nearest gem location from *Pinky's* current location, and provides a path which Pinky can use to reach there. Again, the correctness of the algorithm follows from the comments, connectivity of MAZE and the correctness of partial BFS used in the implementation. Hence, this algorithm is progressive in the sense that *Pinky* only makes a **partial** decision to begin with, and keeps on deciding next location, based on her current location. Comparing this algorithm with the previous one, one obvious advantage is that *Pinky* is not required to memorise/keep track of a lot of data, and is only concerned with storing the location and path to the immediate next position. Below, we do a more formal analysis of the given algorithm and try to determine an expression for the total time required by *Pinky* to complete her adventure.

2.4 Time Complexity (Approximate):

In this algorithm, *Pinky* is performing a partial BFS a total of **seven** times. Partial BFS basically returns the closest gem/end to Pinky's current location. Hence, unlike complete BFS, partial BFS generally takes less time, as it terminates as soon as it adds one of the target locations in the BFS queue. Since time complexity of partial BFS is also proportional to the number of nodes present in the graph, let the time required to do a partial BFS for the i^{th} time be $\boldsymbol{c_i N^2}$, then the total time to do the *seven* partial BFS can be expressed as $\boldsymbol{7c'N^2}$, where $\boldsymbol{c' = \frac{c_1+c_2+..+c_7}{7}}$.
Again, we should not ignore *Pinky's* mental capacity and should also account for delays that may exist due to the large (possibly) amount of data. Now, *Pinky* needs to remember a single BFS queue at a time, along with the path just to her immediate next location. Noting that both the BFS queue and the intermediate path can be as big as $\boldsymbol{N^2}$, and the fact that $\boldsymbol{f}$ is the delay that is introduced per every piece of information that *Pinky* stores (section 2.2), the total information/recall delay in the worst case is $\boldsymbol{f(N^2 + N^2) = 2fN^2}$.
Note that this is just the time *Pinky* needs to come up with the solution. It does not include her travel time. Now say that the length of the travelled path is $\boldsymbol{l'}$. Then knowing that *Pinky* can only move at a fixed speed, the time required by her (for travelling) will be $\boldsymbol{tl'}$, where $\boldsymbol{t}$ is a constant (section 2.2). Combining this with the time requirement of the algorithm, we get:

$$\textbf{Time Required by Pinky}, \boldsymbol{T_2 := (7c' + 2f)N^2 + tl'}$$

2.5 Brief Analysis:

Let us start by computing the difference between T_2 and T_1.

$$\boldsymbol{T_2 - T_1 = [7(c' - c) - 5040f]N^2 + t(l' - l) - 720d - 28f}$$

Note that each of the individual $c_i \leq c$ (**c** defined in section 2.2), hence, c' which is just the average of c_i's, is also less than or equal to **c**. Also, the constants f, d and t are positive, thus, we can re-arrange the above expression to get:

$$\boldsymbol{\tau := T_2 - T_1 = t(l' - l) - \{720d + 28f + [7(c - c') + 5040f]N^2\}}$$

Now some points about this expression:

- l' is the path chosen by the approximate algorithm. Hence, by *optimal* nature of l, it holds that $l \leq l'$. Thus $\boldsymbol{t(l' - l) \geq 0}$.
- From above point it is clear that $\boldsymbol{\tau}$ is a difference of *two* non-negative numbers. Therefore, the magnitude of these two numbers is going to determine whether T_2 is less or T_1.
- For very small values of $\boldsymbol{N}$, the term on the right hand side may be too small, hence, making $\boldsymbol{\tau}$ positive. Thus, indicating that the **optimal algorithm** is the better solution for *Pinky.*
- For large values of $\boldsymbol{N}$, the term on the right hand side may be too large (due to square dependence), hence, making $\boldsymbol{\tau}$ negative. Thus, indicating that the **approximate algorithm** is the better solution for *Pinky.*
- For intermediate values of N, we will have to do some extensive analysis to get a better idea of which algorithm is performing better in terms of time required. We plan to do this via **simulation** of the approximate algorithm and its comparison with the optimal algorithm.

In the following **two** chapters, we describe the design of our simulation and the analysis of the results we obtained.

Chapter 3

Simulation

In this chapter, we discuss our simulation of the aforementioned algorithm (section 2.3) that we believe will help Pinky in finding a reasonably small path in a short duration of time, thus, satisfying the constraints of the Problem Statement (Section 1.3.3).

Keeping different aspects of the algorithm in mind, we decided to simulate the following things:-

- **Ghost Town** and the *Maze*. For simulation purpose, we have taken Ghost Town to be a square of dimension *16*. Hence, the maze is represented by a 2D-Array consisting of *256* elements (MAZE, Section 1.3). We are **not** simulating our maze generation algorithm as that is not a direct requirement of the algorithm. However, we mention that we are using *Random Maze Generation* algorithm to generate different mazes every time the simulation is run. The algorithm works by putting random walls all over the place, while ensuring that no closed areas are formed.

- All the important **Locations**. These include *starting point*, *ending point* and the locations of all the *gems*. In order to make the visuals more aesthetically pleasing, we decided to colour code these locations, so as to not cause any confusion whatsoever. The colour scheme used:- **Green** for *Starting Block*, **Red** for *Ending Block* and **Blue** for every *Gem Block*. (see Figure 3.1)

- Pinky herself. *Pinky* is the protagonist of this problem, hence, any simulation without her will be incomplete. We have simulated her *movement*, as she moves from one location to another. To not make it seem like a static ghost is moving on its own, we have also **animated** her movement. This is done to make the simulation even more realisitc and pleasing to watch.

- **Pathfinding Mechanism**. In the algorithm, we have specified that when *Pinky* reaches a location, she performs a partial *Breadth First Search* in order to pinpoint the next location. We have simulated this algorithmic search by means of an ***Expanding Circle***. This circle is centred at *Pinky's* current location and keeps on increasing in size till *Pinky* pinpoints the next location. (see Figure 3.2)

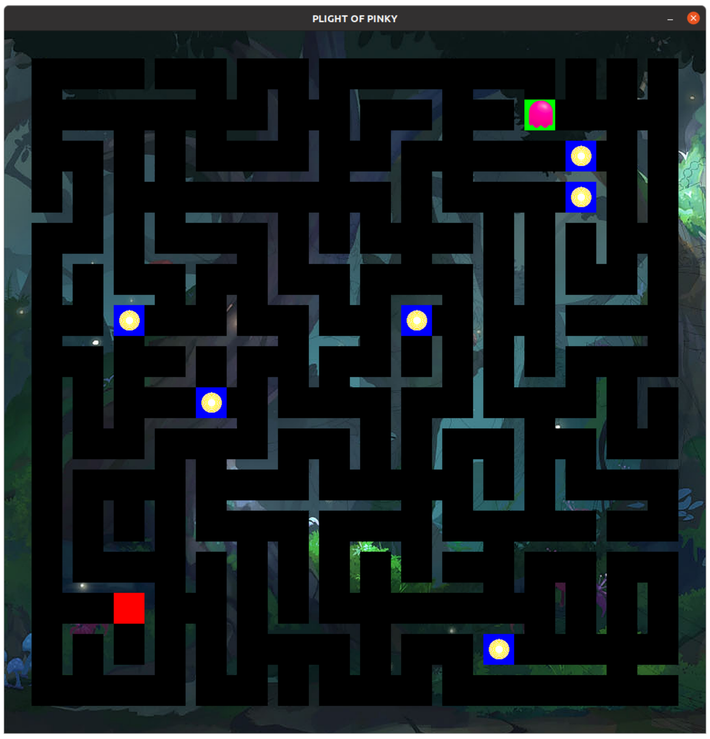


Figure 3.1: Sample **Maze** (with positions marked): **Start**, **End**, **Gems**

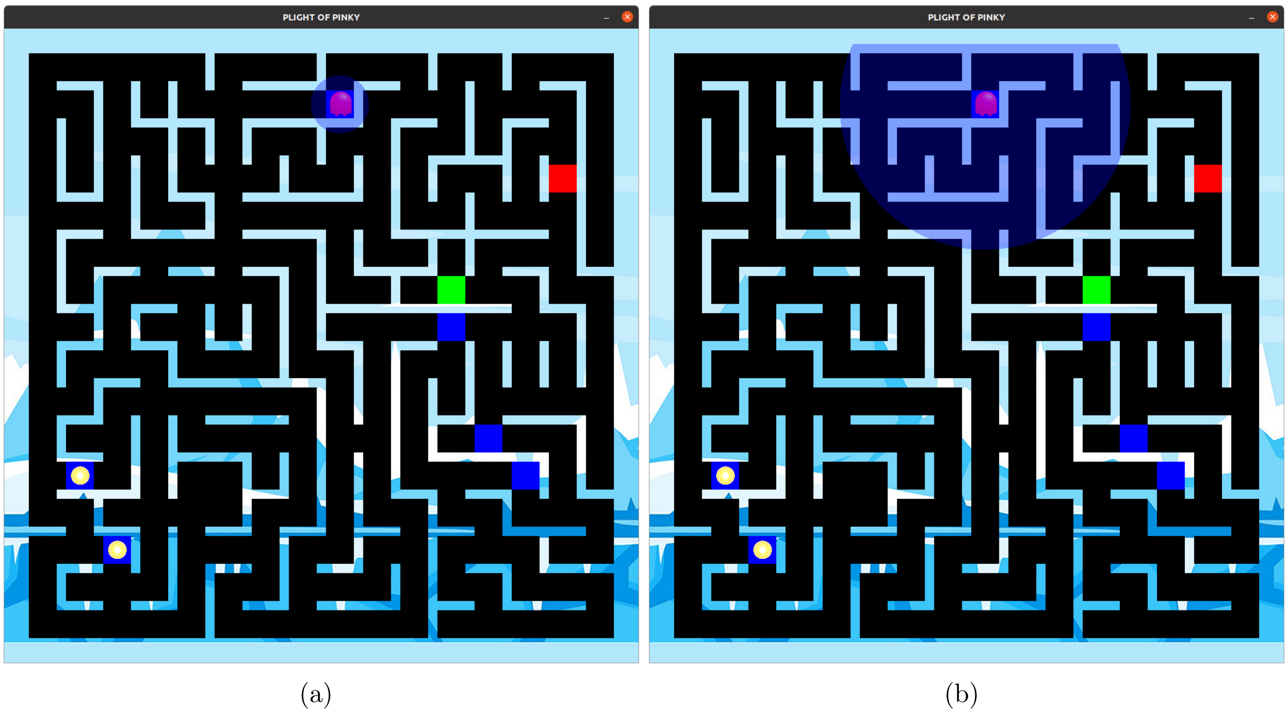
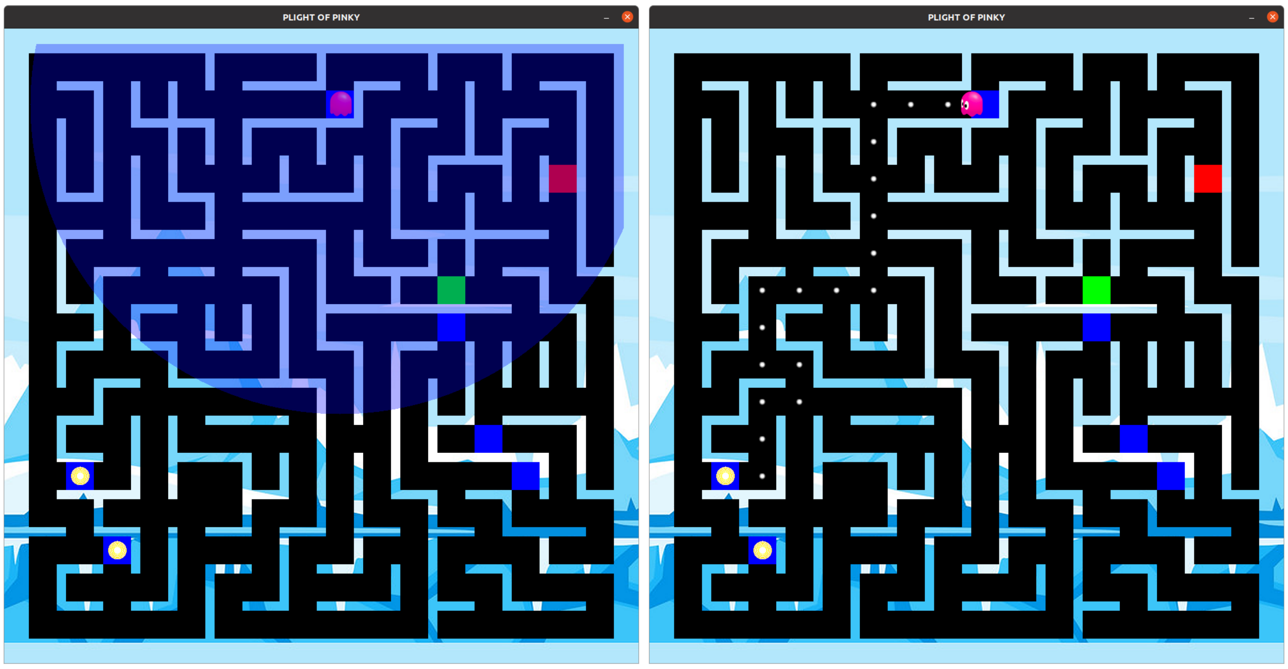


Figure 3.2: Pinky searching for next location, expanding her area of search with **BFS**

- **Path**. Once *Pinky* locates the next destination, we also highlight the path the she will be taking to reach her destination by means of *small-white* dots. This basically simulates the fact that *Pinky* is not only searching for her next destination, but is also determining a suitable **route**. (see Figure 3.3)



(a) (b)

Figure 3.3: Pinky searching and determining a suitable **path** (represented by dots)

3.1 Simulation Features:

In order to make our simulation more interactive and aesthetically pleasing, we have added a couple of features. These include:-

- **Background Theme:** A random background theme is selected every time the simulation is started. This can act as a breath of fresh air in case you are running the simulation multiple times. *Maze walls are blended in with the background for aesthetic effect*
- **Sound Effects:** Various sound effects corresponding to different simulation activities have been added, to be more indicative of what is happening. These include effects for- *starting*, *ending*, *movement* and *successful search*.
- **Pause and Resume**. It is also possible to pause and resume our simulation, in case you have something urgent to do, and cannot miss the simulation at the same time.
- **Frame Rate and Delays:** Our simulation runs at nearly *60 FPS* and has appropriate delays put in place to ensure that the frame rate does not fluctuate much.

- **Pinky's Speed:** By default, *Pinky* moves with a speed of **1 Pixel per Frame**, which basically means 60 Pixels per second in our simulation. However, this value is modifiable, and we allow the user to increase this speed to *5 Pixels per Frame* through a command line argument.
- **Radar Scan:** The radius of scanning circle, or *Expanding Circle* does not increase at a constant rate. Rather, it first increases slowly (1 Pixel per Frame), and then as its size increases, it achieves a steady growth rate of *5 Pixels per Frame*. This is done to ensure that the animation is viable even when two locations are very close to each other, as high increase rate in this case will search too quickly and the animation will be visible only for a few seconds (or a fraction).
- **Path Highlighting:** We have already mentioned that we highlight the path chosen by *Pinky* for her travel. Also, we keep on shrinking the path as Pinky progresses towards her destination. Hence, the number of white dots keep on decreasing. (see Figure 3.4)

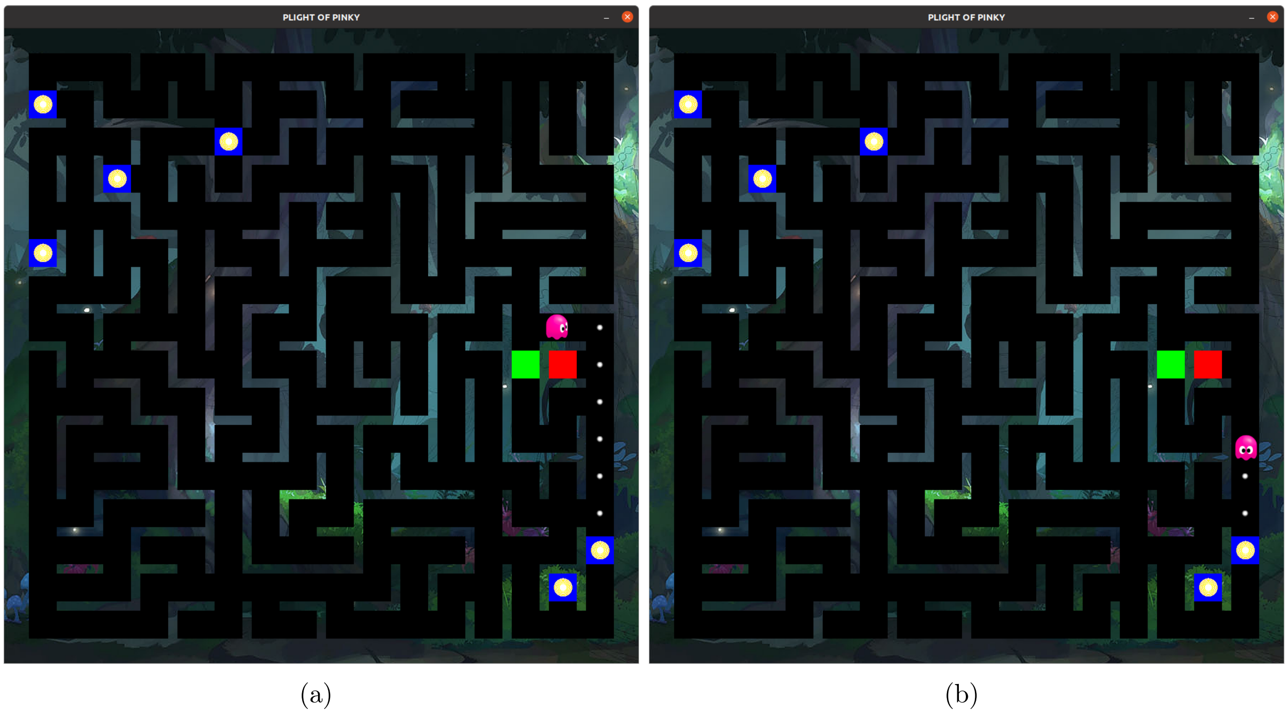


Figure 3.4: Pinky moving towards her destination.The highlighted *path* keeps on decreasing

3.2 Data Structures Used:

We have used a variety of data structures for different aspects of the simulation. These include *maze*, *Pinky*, *eatables* (or Gems), *Rendering* class etc. For exact details and definitions of these classes, see our GitHub Repository Simulation. Here, we describe them in brief:

- As mentioned before, we represent the maze in the form of a 2D Array (or vector) of *Blocks*. Each *Block* represents a location in the **Ghost Town**. Also, each block has five attributes associated with it that tells us about the connectivity between this block and its four neighbours (*up, right, down, left and degree*)
- Each location (like start, end, gem etc) is represented by a *pair* of integers (0-indexed). (0, 0) refers to the top-left block, and (N - 1, N - 1) refers to the botttom-right block. For our simulation purpose, we have used N = 16.
- We also have a vector that stores the locations of the *gems* and the *ending block*. This vector is used by *Pinky* for making her decisions.
- For *Pinky*, we required variables to store her current location, her destination location, the path which she is going to take (vector), neighbouring walls (so as to avoid them), her velocity, and some flags to keep track of various simulation activities like, simulation end, destination reached, path found etc.
- For path highlighting and Gem rendering, we have implemented an Eatable class. The highlighted path is actually made up of a number of small-sized eatables (stored in a vector), which are consumed by Pinky as she proceeds. This helps in creating the *Path Shrinkage* effect with time. Also, Gems are just large-sized eatables, which are also consumed/collected by Pinky when she reaches the destination block.
- For creating various animation effects, we have used *sprite sheets* and frame counters to create smooth animations. The *Radar Scan* is also implemented using timely increment of the scan radius.
- *SDL, SDL_image and SDL_mixer* have been used in addition to structures mentioned above to bring our simulation to life.

Chapter 4

Analysis

In this chapter we analyse the results of our **simulation** (chapter 3) and compare the approximate algorithm with the *optimal* algorithm (chapter 2).

Details of the Simulation:

- *Number of iterations* = 100
- *Maze dimension*, $N = 16$
- *Number of gems* = 6
- *Can two gems be at the same location* = NO
- *Maze status* = CONNECTED
- *Algorithm used:* Approximate Algorithm (section 2.3)
- *Protagonist:* Pinky

Objective of the simulation:

- *Simulate Approximate Algorithm*
- *Determine **optimal** path length*, l
- *Determine **approximate** path length*, l'
- *Compute the difference*, $l' - l$
- *Analyse this difference and extend the analysis done in section 2.5*

Definition of Path length: Any path in our maze can be represented as a sequence of nodes (or locations), $P := (v_0, v_1, .., v_n)$, where there may be a repetition of nodes. The length of such a path (or distance travelled by *Pinky*) is simply defined as n, or the number of nodes - 1.
With the details and objectives of simulation mentioned, let us get to the real data and analyse it to get some more insight into the behaviour of $\boldsymbol{\tau}$ (section 2.5).

4.1 Models for Approximate Path Length

Thanks to our analysis code (*analysis.cpp*), we get a ready-made csv file consisting of the optimal (l) and approximate (l') path lengths travelled by *Pinky*. We can then easily compute the difference $l' - l$, using this data set. On the next page, we represent our findings in the form of a *histogram* and a *line graph*.

4.1.1 l-Independent Model:

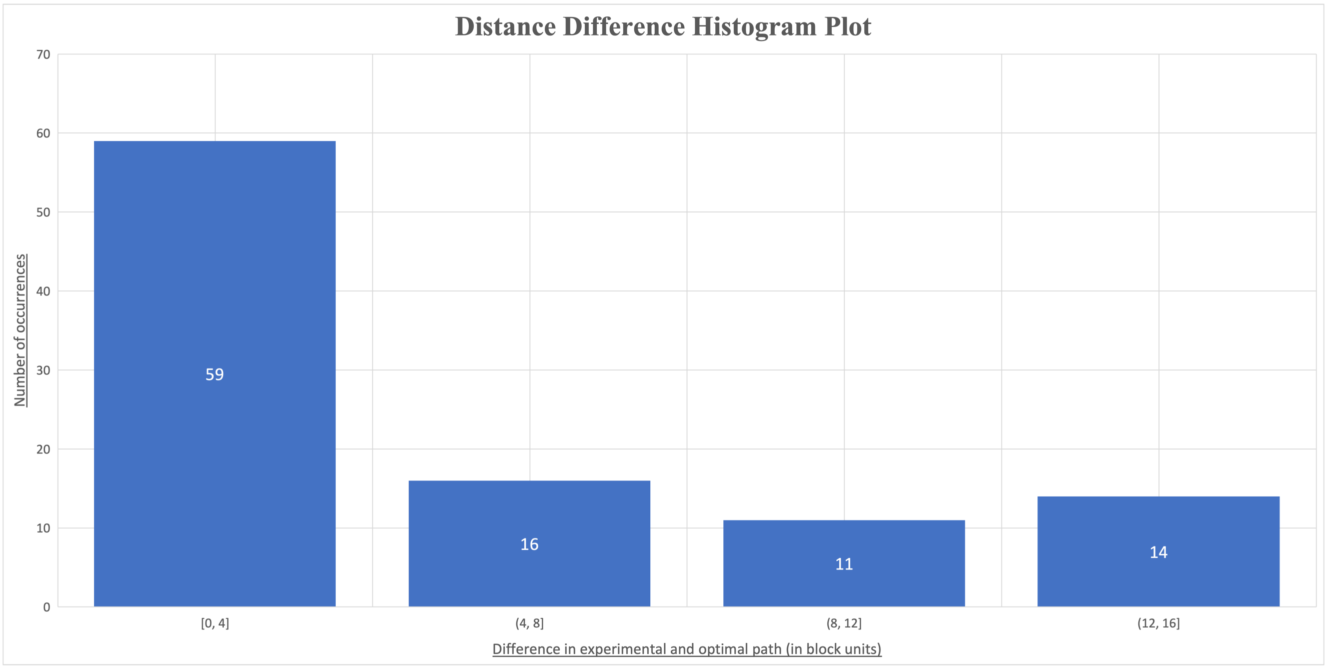


Figure 4.1: Histogram plotting the *No. of occurrences* vs the *path length/distance difference*

Since, there were only *100* total iterations (or occurrences), the height of each bin (in the graph) also represents the *percentage* of cases. Although, the sample set is not large enough for these percentages to generalize, still we report our findings and try to make some conclusions.

Percentages:

- $0 \leq l' - l \leq 4$: 59%
- $5 \leq l' - l \leq 8$: 16%
- $9 \leq l' - l \leq 12$: 11%
- $13 \leq l' - l \leq 16$: 14%

Mean Difference: **5.04**
Standard Deviation of Difference: **5.2724**
Maximum Difference: **16**

From the above statistics, we observe that the approximate algorithm performs quite well for nearly **60%** of the total cases, with the difference in path lengths being less than or equal to 4 units. Also, the maximum difference encountered in the given data set is **16**, which is not surprisingly of the same order as the maze dimension, N.

This can be explained as in the approximate solution, *Pinky* might have to move from one side of the maze to another multiple times, which otherwise in optimal path may not be required. Hence, one way to model l' is by the following equation:

$$\boldsymbol{l' := l + \delta N}$$

Where δ is a positive constant. If we analyse this method further, we can also provide an upper bound for δ, as *Pinky* chooses a redundant path only when making a decision. Since, she makes a decision only *seven* times, she can choose a redundant path for a maximum of **seven** times. However, as the data set is not big enough so as to allow us to make tight assumptions like these, we are not going to bound δ as of now.
Now, using this value of l', we get an expression for τ by substitution (section 2.5)

$$\boldsymbol{Eq.1 \equiv \tau := t\delta N - \{720d + 28f + [7(c - c') + 5040f]N^2\}}$$

4.1.2 l-Dependent Model:

Another way of modelling l' is by expressing it as some fraction of the optimal path l. We represent this idea through the following graph:

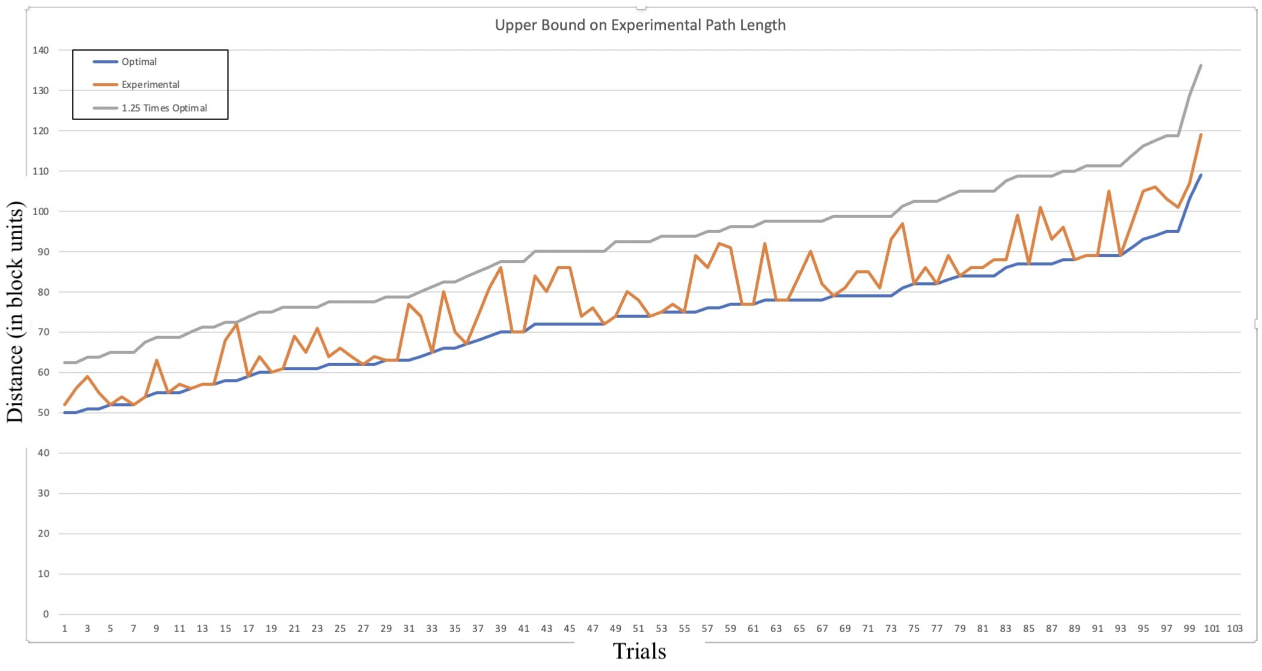


Figure 4.2: Line graph plotting $1.25l$, l' and l with the trial/simulation number

Maximum Optimal path length: **109**
Maximum Approximate path length: **119**

From the graph, it can be seen that on the given data set, the *experimental* length (approximate) never exceeds 1.25 times the *optimal* path length. Although, this constant 1.25 may not be a real estimate of how large the deviation can be in reality (as the data set is not big enough), it still provides us with some idea of how we can model the path length that we get by the approximate algorithm. Below we define this model more formally:

$$\boldsymbol{l' := l + \gamma l}$$

Where γ is a positive constant, similar to δ in section 4.1.1. For this particular data set that we analysed, we can upper bound γ with 0.25. But again, this bound may not be general enough to be used in all contexts. Hence, we do not bound it as of now.
Now, using this value of l', we get an expression for τ by substitution (section 2.5)

$$\boldsymbol{Eq.2 \equiv \tau := t\gamma l - \{720d + 28f + [7(c - c') + 5040f]N^2\}}$$

4.2 Summary of Models for l'

In this section, we summarise both the models mentioned above. The time difference equation for the two is given below:

$$\boldsymbol{Eq.1 \equiv \tau := t\delta N - \{720d + 28f + [7(c - c') + 5040f]N^2\}}$$

$$\boldsymbol{Eq.2 \equiv \tau := t\gamma l - \{720d + 28f + [7(c - c') + 5040f]N^2\}}$$

Some comments:

- $\boldsymbol{N}$ depends only on the *Ghost Town* (dimension of square)
- $\boldsymbol{\delta}$,$\boldsymbol{\gamma}$ and $\boldsymbol{l}$ depend on the maze structure and the locations of the Gems, Start and End points. These will vary even when the *Ghost Town* is the same.
- $\boldsymbol{t}$,$\boldsymbol{d}$ and $\boldsymbol{f}$ are constraints related to *Pinky's* speed, math skills and mental capacity respectively, and will be fixed for a given ghost/object.
- $\boldsymbol{c}$ is the BFS constant, and it should be independent of the size of *Ghost Town* and maze structure, provided that the maze is completely connected.
- $\boldsymbol{c'}$ is the *partial* BFS constant and it is dependent on the maze structure and locations of Gems, Start and End locations.

We describe some *plausible* methods for determination of these constants in **Future Scope**.

Chapter 5

Conclusion: What should Pinky do?

In this chapter, we conclude our discussion on the given problem statement and decide what algorithm should *Pinky* ultimately use. We use the two models described in *Chapter 4* along with the data set that was used for analysis to help us make this decision.

$$\boxed{\text{Given Maze Dimension: } N = 16}$$

- If we use the ***l-Independent*** model, we see that for almost 60% of times, using approximate algorithm gives very small values of δ (≤ 0.25). Hence, using *Eq.1* (section 4.1.1), we get that $\tau < 0$ (Assuming that constants t and c are not extraordinarily small or large, which is reasonable to think of). Hence, this model suggests that *Pinky* should use the *approximate* algorithm for the given maze.

- If we use ***l-Dependent*** model, we see that $\gamma \leq 0.25$ for the given data set (consisting of 100 data points). Again, observing that the maximum value of optimal path is *109* on the given data set, $\gamma l \leq 0.25 * 109 \ll N^2$, hence using *Eq.2* (section 4.1.2), we get that $\tau < 0$ (Again assuming that the constants t and c are not extraordinarily small or large). Hence, this model also suggests that *Pinky* should use the *approximate* algorithm for the given maze.

In conclusion, for the given maze, *Pinky* should prefer the approximate heuristic to collect all the gems and reach home before she gets caught by the *Pacmen*.

Chapter 6

Future Scope

Below we provide some unexplored research ideas for the Steiner TSP and it's applications in a maze like setting. We suggest the following experiments to be conducted on sufficiently large mazes and running them over millions of simulations for better results.

- **Graph Spanners:** We may use additive and multiplicative distance spanners to reduce the computation cost of each Breadth First Search run. Few important questions that need to be answered are -
  - What is the most optimal spanner? This requires a utility-runtime tradeoff analysis.
  - What is the improvement in speed compared to the nearest-neighbour heuristic?
  - What is the accuracy loss with respect to our old heuristic? Use $\boldsymbol{\gamma}$-value as proxy.

- **Other Heuristics:** Compare the nearest neighbour algorithm with other heuristics such as
  - Greedy Heuristic
  - Insertion Heuristic
  - Christofides Method

- **Machine Learning:** We can use Convolution Neural Networks(CNN) to predict whether *Pinky* should follow the approximate path or the optimal path. Further, similar ML based algorithms can also be applied for approximations of constants that were mentioned before such as δ, γ and c'.

Chapter 7

References

- Heuristics for the Traveling Salesman Problem | Ch. 6: Future Scope