The Penniless Pilgrim Riddle

Table of contents:

1. Introduction: The Penniless Pilgrim Riddle

1.1. Informal description of the riddle

After months of travel, you've arrived at Duonia, home to the famous temple that's the destination of your pilgrimage. The walk from the welcome center to the temple is some blocks walking away, and you were handed the brochure about the city rules. This city is… unique, in that:

- There's a strange tax: The tax begins at 0, increases by 2 silvers for every block you walk east (right), doubles for every block you walk south (down). A walk west-bound (left) or north-bound (up) will however decreases it by

2 and by half, respectively.

- The city's rule, as per the temple's religion, also forbids you treading back any path you've walked, but crossing is allowed.

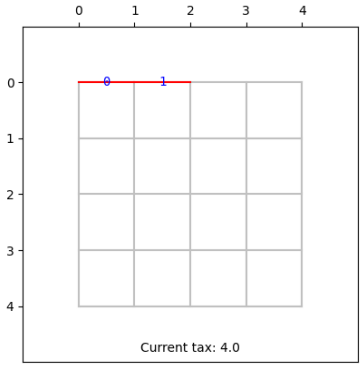
You just realized you did not bring any silver at all. Even worse, arriving at the welcome center already sets you back some silvers. As a pilgrim, you do not want to enter citizen's blocks, nor break any rules. How would you reach the temple, located at the very lower-right intersection, without paying any tax?

In this project:

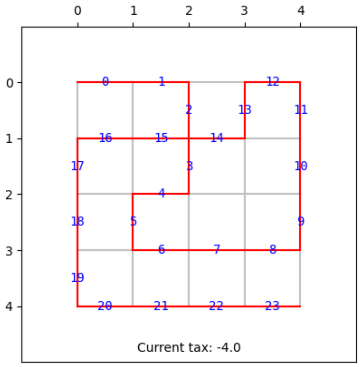
- The board size is randomized, the initial walked road is randomized.
- If multiple solutions are found, a solution is considered better if the final tax is lower.

1.2. Images

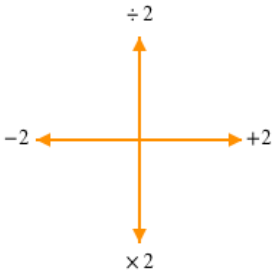
> The game we researched is inspired by the video "Can you solve the penniless pilgrim riddle? - Daniel Finkel" of TED-Ed: https://youtu.be/6sBB-gRhfjE



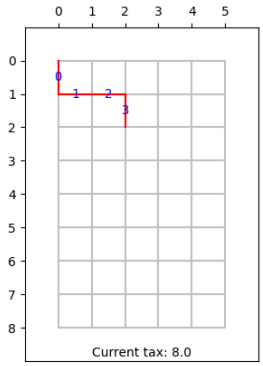
The initial state of the game in the video



Its best solution we found using pure brute-force, which is considered better than the solution provided in the video



The tax fluctuate differently based on direction



One of the randomized initial states

2. Description of the problem

2.1. Purpose of the problem

The Penniless Pilgrim Riddle is performed by a mapping gameplay. We classify it as a problem solving by searching, specifically Single-state problem (The environment is known, observable, discrete and deterministic).

Help us to understand basic techniques of AI and advance more technical skills: building a program using Python, game developing, computational thinking, observation and statistics,...

2.2. Search problem formulation

2.2.1. Components of a state

Each state is composed by 4 components:

- `board_size`: the size of the board counted by squares, remains constant throughout each game. It is of the form `(m, n)`, where m and n are the numbers of rows and columns, respectively. Therefore, there are $m + 1$ horizontal lines and $n + 1$ vertical lines, that makes $(m + 1)(n + 1)$ intersections and $(m + 1)n + (n + 1)m = 2mn + m + n$ moves on the board.
- `walked_moves`: the sequence of moves that the pilgrim has walked through. It is a list of tuple, in which each tuple is of the form `(x, y, <DIRECTION>)`, `(x, y)` is the coordinate of the pilgrim (`current_pos` below). And the `<DIRECTION>` only has 4 values `R`, `L`, `U` and `D`, which are Right, Left, Up and Down, respectively.
- `current_pos`: the current position of the pilgrim. It is of the form `(x, y)`, which is their coordinate. `(0,0)` and `(m,n)` are the positions of the intersections in the upper-left corner and the destination, respectively.
- `current_tax`: the current number of silvers that the pilgrim is owing the city. It is a real number.

2.2.2. Initial states

The board size is randomized, from 4 to 8 each side. Therefore, there are $5 \times 5 = 25$ different board sizes. There are 2 states for each board size that differ in initial forced moves. The initial moves of the board size `(m, n)` are randomized using the following algorithm:

- Calculate the number of forced moves for the 2 states. Respectively, they are $\lfloor \frac{\sqrt{m \times n}}{2} \rfloor$ and $\lceil \frac{\sqrt{m \times n}}{2} \rceil$.
- Randomly generate the 2 states. The current position and the current tax will follow the rules after moving on that road, starting from `(0, 0)` and `0`, respectively.
- If the two numbers of moves above are equal (if and only if $\frac{\sqrt{m \times n}}{2}$ is an integer), check if the initial position of the second state is identical to the initial position of the first state. If so, regenerate the second state until they are different.

2.2.3. Actions

`R`, `L`, `U` and `D`. As mentioned, going outside the board or treading back to any road that is walked through is not allowed.

2.2.4. Path cost

$$Old\ cost \rightarrow \{(L, -2), (R, +2), (U, \div 2), (D, \times 2)\} \rightarrow New\ cost$$

2.2.5. Output

For each algorithm, analyze the

- Time complexity.
- Space complexity.

For each instance of the problem (initial state), practically find and analyze the

- Sequence of moves from the initial position to the temple and the remaining silver “owed”.

- Time complexity (number of nodes expanded in order to solve the puzzle).
- Space complexity (number of nodes kept in memory).

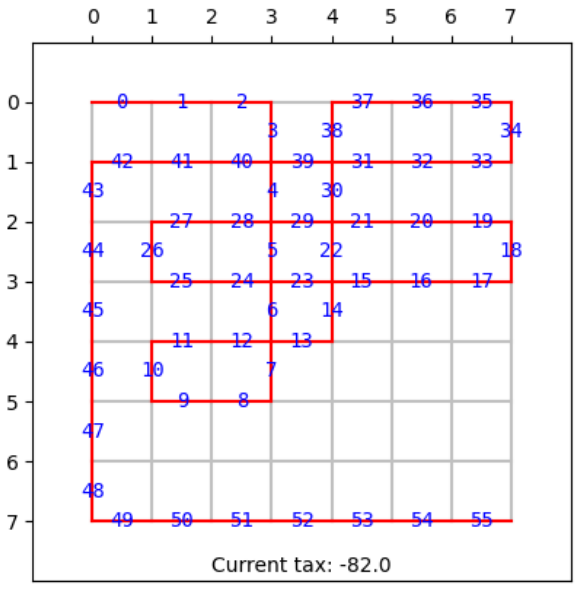
3. Explaining choice of algorithms, and their parameterization

Three algorithms are chosen: Depth-first Search, A-star and Recursive Best-First Search (respectively DFS, A* and RBFS).

3.1. Depth-first Search

3.1.1. The reasons to choose Depth-first Search (DFS)

- The basic, uninformed search algorithm. Quick to implementation, this helps us grasp the problem first-hand and prepare for the two other algorithms.
- If searching from the goal to start and using clockwise loop as tax 'cheating', DFS can quickly form one by expanding all-Left, then all-Up, then all-Right, then find a satisfying path.
  This greatly reduces space/time complexity as DFS now only has to search on (m-2, n-1) board.
- UCS is clearly not usable here (tax changes based on direction);
  Search space is finite so DLS is not usable.
  IDS (ID-DFS) was very well considered, until we found this was a DFS solution (Depth = 55):



That had left BFS and DFS to the choice.

3.1.2. Comparing with Breath-first Search (BFS)

- Only in this problem. Averaging/assuming two branches per node, $c = m \times n$ nodes with depth $d = \frac{c}{2}$.

	DFS	BFS
Principle	Expand nodes of the same branch, until depth reached.	Expand all nodes of the same depth.
Space complexity (Theory)		

Space complexity (Practice)	$O(2 \times c)$	$O(2^{d+1})$
Time complexity (Theory)	$O(2 \times (m-2) \times (n-1))$	$O(2^{d+1})$
Time complexity (Practice)	$O(2^c)$	$O(2^{d+1})$
	$O(2^{(m-2)(n-1)})$	$O(2^{d+1})$

In conclusion, DFS is more suitable for the problem than BFS. We picked DFS as our first algorithm to get ahold of the problem.

3.2. A-star Search (A*)

3.2.1. The reasons to choose

- A* Search algorithm is simply a tried-and-true search strategy.
- We could've picked IDA*, but given that g, the current tax, fluctuates WILDLY, we couldn't decide on a time limit t_limit and an iter.
  Besides, the heuristic for A* is guaranteed not admissible, thus not complete, since `0 ≤ h ≤ h'` and the final tax `h'≤ 0`. Going for IDA* with an uncertain function `f` and uncertain cost `g` is the same as running A* but much more time-consuming for tiny space improvement.

Therefore we initially picked A*, but still were indifferent about RBFS vs. IDA*.

- As we needed to test the algorithms as we write, we tried A* with `h = 0`, basically making it a brute-force search. Poor my laptop's fan enduring 5-minute search on a 4x4 board that A* found in a blink.
  Needless to say, brute-force was hasted out-of-the-window, and only was used as internal testing for smaller boards.
- We've discovered ARA* (Anytime Repairing A*) that, in my opinion, fits this quite well: find a feasible path, then improve it to match our "optimal" solution.
  Well, that it would be world peace, until 2 weeks prior to deadline, when one of us out of blue decided to run brute-forcing again and got bamboozled: there's a solution for our default 6x6 board with end `tax = -248` (previously we thought one with `tax = -240` is optimal).
  `improveSolution` function (for ARA*) does not work efficiently anymore, so I decided to drop it. (After all, we already have 3 algorithms)
- Since this problem involves a single agent: the pilgrim, and the environment is wholly static, deterministic and fully observable, all 2-agent algorithms like a-b pruning are out-of-question fairly quickly and early into the problem discussion.
- None of us found any local search useful (or at-all usable), therefore they are ignored.
  That leaves us with base A* search and IDA* or RBFS.

We are still undecided if A* is the way to go, because, at first glances (and later loosely confirmed), there seemed to be no way to formulate mathematically the heuristic that truly matches its definition of “an estimated cost of cheapest path from any node to goal”.
However, with the lack of informed search strategies, we stuck with A* as a baseline for other algorithms. RBFS and IDA* are still indifferent to us.

3.2.2. Heuristic

NOTE: affected by the sudden find in true OPTIMAL path, which indirectly makes our heuristic obsolete, we've decided to not find OPTIMAL path anymore, only FEASIBLE path (any path that satisfies all conditions).

This "heuristic" is not truly a heuristic. It doesn't estimate true cost. It merely guides A* to find a satisfying path quickly.
We actually spent 50 cumulative hours working on JUST heuristic, none takes wings. Considering our heuristic is *by default of problem* inadmissible, we decided to go with heuristic that guides the algorithm to find path quickly, instead of an estimation.

For the formula, assuming board size is (m, n) current node is at (x, y), we have:

$h = n \times y \times \frac{n-y}{2} + x$

It seems random? It indeed is. This tells A* to go down first, thus making it easier to make counter-clockwise loop(s), therefore 'cheating' tax.

3.3. Recursive Best-first Search (RBFS)

3.3.1. The reasons to choose

- Instead of keeping all the nodes in memory, Recursive best-first Search uses recursion to “forgot” expanded node and store a f_limit for the best alternative path to backtrack later.
- Basically, it’s like a combination between A* Search and Depth-first Search that cost less space because of inheriting the space complexity from DFS, and is more efficient if it uses more memory instead of re-expand many nodes.
- The heart of the algorithm:

```
h = 1 * manhattan distance of node to goal
g = past_cost * length of the traveled path
```

  - Neither a consistent nor an admissible heuristic: The manhattan distance of a node to goal is always the shortest path so `h` always evaluate the shortest path but cause of the cost change after a move, h is not satisfied `0 ≤ h ≤ h'`, which means h is not admissible and of course not consistent. ⇒ It’s not an actually heuristic but a navigation function instead, since our goal is not OPTIMAL but FEASIBLE indeed.
  - The reason for choosing this heuristic is by comparing values and data, it gives an acceptable result in both time and space-consuming. In a limit of 100000 recursions, the algorithms can solve $49/50$ states in the data sample.

3.3.3. Parameter

- **State**: state of each recursion.
- **Node**: the node expand of each recursion.
- **Goal node**: the goal of the problem that every node aim to.
- **f_limit**: the best alternative path that if the present path can reach the goal, backtrack happen with.

4. Implementing the algorithms

4.1. Depth-first Search

4.1.1. The order of implementing steps

1. **Model the problem**
   a. `State`: Already implemented as a whole from the start. Consisting of:
      i. `current_pos`: current position on board.
      ii. `current_tax`: current tax accumulated so far.
      iii. `walked_moves`: all moves agent has taken so far.
      iv. `available_moves_list()`: function returning all possible moves.
      v. `path`: variable exclusive to our DFS algorithm, storing path leading up-to `State`.

b. `START` and `GOAL`: `State`s with fixed `current_pos`.
For this problem and our way of DFSing, `current_tax` of `GOAL` is 0.

c. `t_limit`: time limit. For DFS, since it either finds solution extremely fast or not at all after 60 seconds, we decided to limit it to **10 seconds. (**There is no difference in solved states when `t_limit` is 10 or 60)

i. Supplementary variables:

1. `t_start`: a variable storing the start-searching time
2. `t_limit_r`: a boolean telling whether `t_limit` is reached

2. **Apply the algorithm to the problem**
A simple recursive DFS implementation is good enough.

```
#!/pseudo/code
func dfs(curr) -> None:
  if time_limit_reached(): exit();
  for move in curr.available_moves_list():
    temp = copy(curr)
    temp.move_to(move)

    if temp.current_pos == START.current_pos and \
       temp.current_tax >= START.current_tax:
      save_path();
      exit();

    temp.path = copy(curr.path);
    dfs(temp);

main:
  dfs(GOAL);
```

3. **Fix running and logic errors**
Nothing to say here, really.
Except, merci beaucoup StackOverflow. :3

4. **Test with the prepared sample data**

4.1.2. The difficulties in implementing

Beside the needs of decent CPU and plenty of RAM, which one of us has (Kaby Lake i7, 16GB RAM), to run and debug the algorithm unrestricted, there is no real setbacks. After all, this is a basic, uninformed search.

4.2. A-star Search

4.2.1. The order of implementing steps

1. **Model the problem**

a. `State`: Already implemented as a whole from the start. Consisting of:

i. `current_pos`: current position on board.

ii. `current_tax`: current tax accumulated so far.

iii. `walked_moves`: all moves agent has taken so far.

iv. `available_moves_list()`: function returning all possible moves.

v. `parent`: variable exclusive to our A* algorithm, storing reference to `State` that moved to it. (node's parent)

vi. `f`, `g` and `h`: respectively A* function, current_tax and heuristic of `State`. g is not

b. `START` and `GOAL`: `State`s with fixed `current_pos`.

c. `t_limit`: time limit. For A*, we are on board with the standard 60-second limit.

i. Supplementary variables:

1. `t_start`: a variable storing the start-searching time
2. `t_limit_r`: a boolean telling whether `t_limit` is reached

2. **Finding heuristic**
   It was previously mentioned in 3.2.2 (and partly in 3.2.1 as well). We however will summarize:

   - (At lack of research experience and/or time,) we concluded that the heuristic for this problem cannot be admissible.
   - Due to heavy instability in cost fluctuation (and as mentioned the lack of research experience and/or time), none of our suggested heuristic actually worked for the majority of boards.
   - For the two above reasons, we had agreed to slightly “bend” the definition of heuristic to be “a function that manipulates A* to search the more potentially satisfying path”.

3. **Words to code. Problem to solve**

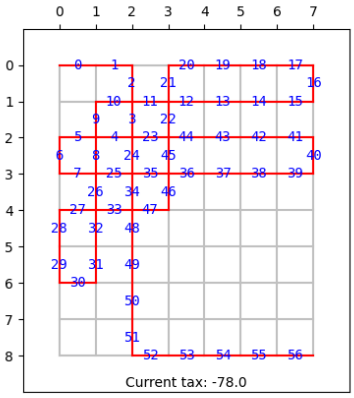
```
#!/pseudo/code

def __astar_h(cur):
  m, n = cur.board_size;
  x, y = cur.current_pos();
  return n * y * (n-y) / 2 + x;

def astar():
  setup_variables();

  # list of ready-to-expand nodes
  _open = list();
  _open.append(START);

  # while not stopped, continually expand every nodes
  while _open:
    # ABORT: time limit
    if time_limit_reached(): exit();

    _curr = _open.pop_node_with_minimum_f(); # node.f

    # ABORT: solution found
    if _curr.current_pos == GOAL.current_pos:
      if _curr.current_tax <= 0:
        found_path();

    lst = _curr.available_moves_list();

    if move_lst = []: continue;
    for mv in move_lst:
      temp = copy(_curr);
      temp.move_to(mv);
      temp.f = temp.current_tax + __astar_h(temp);
      #   .f       .g             .h

      temp.move = mv
      temp.parent = _current;

  path = [];
  node = copy(GOAL);
  while node.parent is not None:
    path.append(node.move);
    node = node.parent;

  return path[::-1] # reverse of path
```

4. **Debug and test with prepared sample data**

4.2.2. The difficulties in implementing

A few hiccups in programming at first, but we managed through in a few days. After all, the breadwinner of A* comes from the making of heuristic formula.

- It is clear that cost at any node (node.`current_tax`) is not predictable. Despite that, we have tried many attempts of heuristic that hold true to its spirit, to our vains (including, but not limited to, path deviating far from our desired solution, path forming weird shapes that is counter-intuitive, ...)
- 50 hours in was our limit. We decided not to follow A* heuristic's definition any longer. Instead, we went on to create a heuristic that "guides" A* to the right way. Without regards to its optimism, admissibility and consistency, a heuristic formula was quickly made. However...
- We quickly found out that in order to shape A* to our desire, the formula would need to be very complicated, be it multiple conditions and multiple formulae per each of, or a formula that cannot fit on two lines of teammate's 27-inch monitor (emphasis: two). Similarly, a plain formula would result in path looking extremely counter-intuitive (like below).



- For the sake of simple heuristic, understandable in a peek, without over-arching presentation time limit, we settled on a simple heuristic instead, accepting that solution is imperfect.
- For the above reasons, we also dropped OPTIMAL path searching, simply because we could not design a true-to-spirit heuristic (in a limited time-manner), as in we have to expand a large amount of nodes to be able to conclude that one solution was optimal.
  (This was fine on smaller boards like 4x4, but even our teammate's laptop has 16GB of RAM gave up on a 5x6 board, consuming 13/16 GB. Google Colab *might be* able to handle 8x8 and up, if not for its timeout stopping the search. Even if we had more RAM, it was regardless sluggish.)

In conclusion: our heuristic strives for "guiding" A* to a possible solution instead of estimating its distance to goal. This problem and time prohibit us from making a truly good heuristic (if any) that can find OPTIMAL path in a reasonable timeframe; and FEASIBLE path quickly, while is space-friendly enough for 8GB of RAM, which is the norm for most PC nowadays.

TL;DR: our heuristic = simple formula, is space-friendly, is not admissible, can be pessimistic, guides A* to solution.

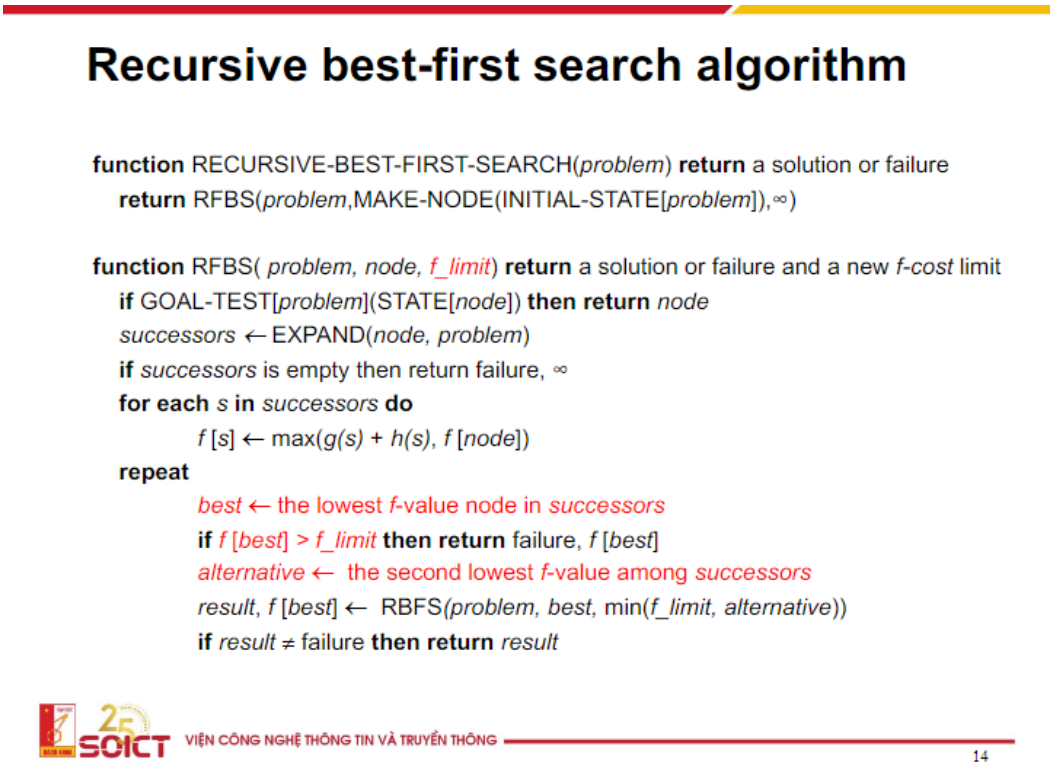
4.3. Recursive Best-first Search

4.3.1. The order of implementing steps:

1. **Model the problem**
   To imagine the algorithm in the best way, the first thing to do is model the problem. This is a simple undirected weighted graph problem. Therefore, each vertex is a node, which has properties below:
   - The `origin`: the starting point.
   - The `past_cost`: the cost going from a node to another and equals to the tax traveled.

- The `child` and the `parent`:
  - The child node is the next available node that satisfies not going outside the board and not coming back to the traveled path.
  - The parent node is a node that its child node is the present node
- The `pathway`: the list of Node having been passed to the present node

2. **Find heuristic**
   Next step is finding a heuristic, an important part that decides the accuracy and the complexity of the algorithm. A good heuristic can find the optimal way to the goal while also reduce time as well as space complexity.
3. **Apply the algorithm to the problem**



4. **Debug the error and test with the sample data**
   Most time-wasting step. The sample data is a list of 50 states of the game.

4.3.2. The difficulties in implementing of each step

1. **Model the problem with Node**

   Difficulties in establishing the relationship between the instance of the problem and the properties of a node, and in expressing significantly every character of each node's property in code.

2. **Finding heuristic**

   Estimating the distance to the goal is unlikely to be correct because of non-uniformly changing past cost between exponential and linear $(+2, -2, *2, /2)$.

3. **Apply the algorithm having been learned to the problem**
   - The next Move can be duplicated which can cause the algorithm to stuck in an infinite loop.
   - If the heuristic is not good enough, many branches might be skipped in the way to the goal and RBFS may not find the right solution.

- RBFS is a memory-bound method, the formula of space complexity is a linear function but on the other hand, the time complexity is an exponential function because of having to expand one node multiple times. Therefore balancing the running time and the space is difficult.

4. **Debug the error and test with the sample data being prepared**
    - StackOverflow is any developer's best friend, junior or not.
    - Data samples may be unsolvable, and not easy to prove.

5. Comparing the results of the algorithms

Testing conditions and methodology:

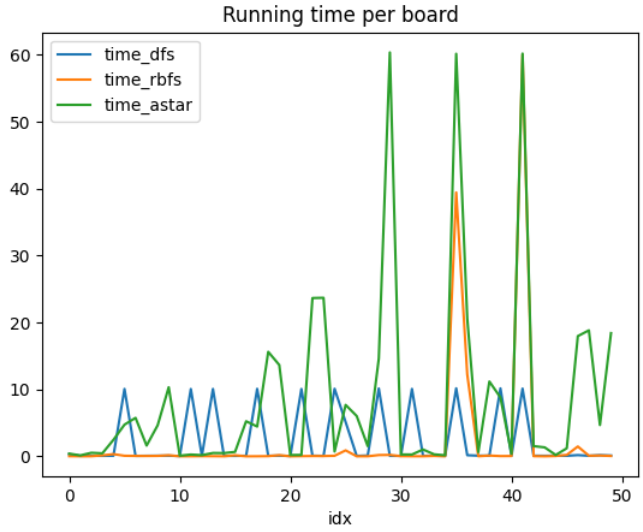
- Dell Inspiron 7306 (Core i5-1135G7 4.20 GHz, 8GB RAM, Windows 11, SSD, charged 100%).
- All algorithms are run in Visual Studio Code with Python 3.9.9 and all third-party dependency upgraded to latest version.
- Only start running when the OS is stablized (CPU usage less than 5%, RAM usage 4.5 GB, Disk usage less than 2%); Airplane mode turned on to prevent random disruption.
- All output data files, directories are created prior to testing to reduce randomness in I/O activities.
- Step of each run:
    - Restart the computer.
    - Open Visual Studio Code, Task Manager and wait for about 5 minutes (for OS to stablize).
    - Execute algorithm on 50 pre-generated boards.

NOTE: Due to random spikes in CPU usage under Windows, all CPU data are recorded manually, thus might be susceptible to bias.

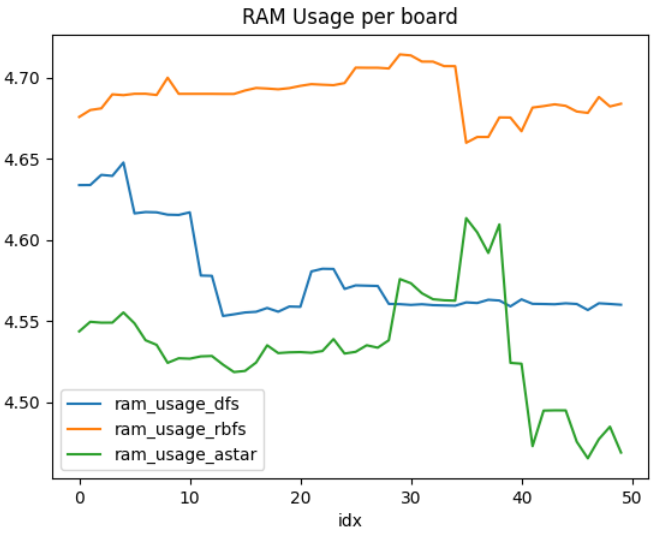
Algorithms	DFS	RBFS	A* Search
Board solved (out of 49)	28	49	47
Total Number of Iterations	70808	150951	45049
Total Running Time for solved Boards (s)	111.3305	57.5106	252.1998
Average Memory Consumption (GB)	0.2	0.3	0.1
Peak Memory Consumption (GB)	0.4	0.4	0.2
Peak CPU usage (%)	67	100	45
Average CPU usage (%)	40	100	38
Peak CPU Clock-speed (GHz)	3.65	2.87	3.81

GRAPH:

- Running time per board
  Measure: seconds
  Scale: 1:1
  Legends: Running time for DFS, RBFS and A* Search, respectively from up to down.

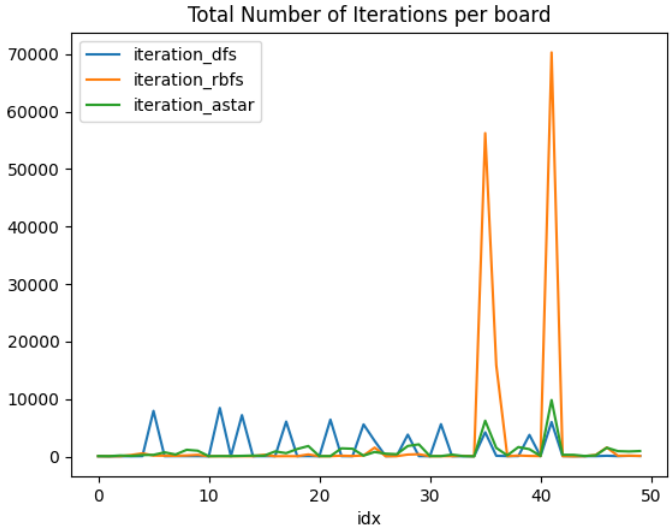


- OS Memory Consumption per board.
  Measure: GB
  Legends: OS Consumption while DFS, RBFS and A* Search run, respectively from up to down.
  (Note: RAM before running is always 4.5 GB)

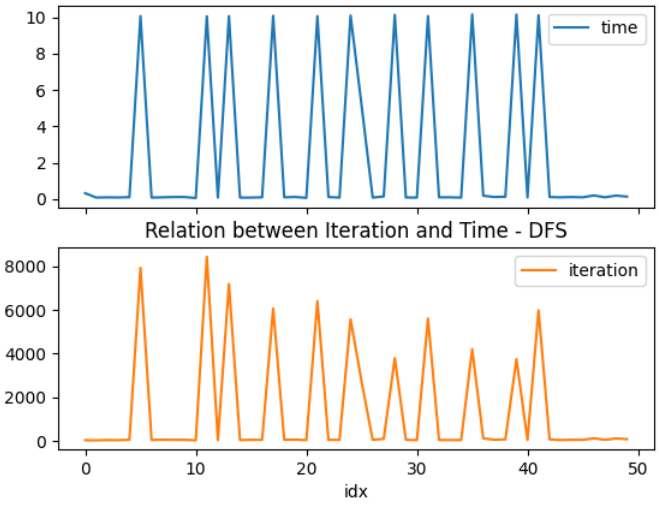


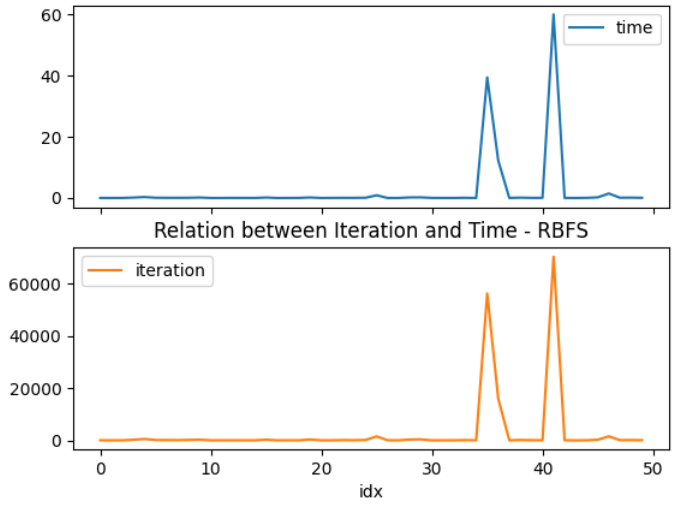
- Comparing Total Iteration count:

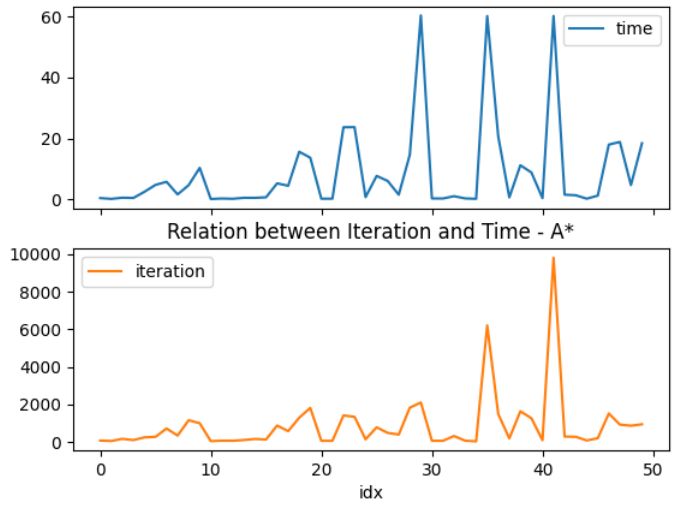
  Measure: number of times



- Loop/Recursive Iteration in relation to time per board
  Measure: seconds, number of times
  [RECAP]







- Did it find a solution?
  - RBFS performed the best out of the 3 algorithms we implemented, having solved all 49/49 board; successfully terminated by time limit on 1 unsolvable board.
  - A* finished 2nd, losing out on 2 solvable boards by time limit.
  - DFS, expectedly, performed the worst out of 3, with only 28 boards solved.
- How fast did it run?
  - RBFS once again took the top for fastest total running time, averaging 1.17 seconds per solved board.
  - Next in line was DFS, managing to find solution in less than a second for all solved boards.
  - Slowest one was A*, with running time ranging from 0.1 to 30 seconds, averaging 5.37 seconds per solved board.
- How space-efficient was it?
  - A* took the cake for this, only having to expand less than 50000 nodes in total and requiring no more than 0.2 GB of RAM.
  - RBFS is the worst of 3, needing thrice as much node expansions as A* and in average 0.3 GB of RAM.

6. Conclusion and possible extensions

6.1. Analytic conclusion

- Recap about Search Algorithms
  - RBFS is most suited for this problem, having the least running time and the best performance.
  - A* Search is inefficient due to its function having to take into account the cost up-to the expanding node, which as we have mentioned is unpredictable. It is thus nigh-impossible (for us) to come up with a heuristic that is always optimistic for this particular problem.
    - We could have come up with a heuristic that depends on path cost, but then what separates A* and (R)BFS if we do so.
  - Our thinking DFS would find solution extremely quickly if expanded from `GOAL` was partially correct. However, it choked on all boards where `START` has only one way out to reach `GOAL`, and that adjacent node has no more than 2 possible moves.
    - Explanation:
      - When DFS searches to the node next to `START`, called node A, it has 2 possible moves.
      - If the move to `START` is not problem-satisfied, it will move the other way. After that, it will expand every single node yet unexpanded, none of which would lead to `START` since A is unreachable, thus `START` is also unreachable.
      - By the priority of node expansion, it might expand an absurd amount of nodes before actually finding a solution.

Based on these findings, we understand better the way to implement different searching algorithms in a specific programming language, as BFS, DFS, A star and RBFS. The conditions and basic search strategies can be enhanced (by finding the good enough heuristic) to cater to the formulations of distinct problems.

6.2. Possible extensions of the Penniless Pilgrim Riddle

- `GOAL` is also randomized.
- “Remove” some roads: under construction, 2 adjacent city blocks are merged, etc...
- Limit of path length (the pilgrim can only walk up to 17 roads before getting exhausted).
- Extension of limit of path length (after 17 roads walked, the pilgrim can hire a biker. The biker charges +0.8 per road moved, paid in full when arriving at `GOAL`).
- Shortcuts with unique cost: There might be a tunnel from (1, 3) to (6, 7) with cost *3, then +128 (cost the same going the other way, all expressions are linear, use once only).

7. List of tasks

Group 17

1. **Hoàng Trần Nhật Minh 20204883** minh.htn204883@sis.hust.edu.vn
   - Leader (100%)
   - Game (100%)
   - Test (50%)
   - Analysis (5%)
   - Report (20%)
2. **Nguyễn Hoàng Phúc 20204923** phuc.nh204923@sis.hust.edu.vn

- Test (50%)
- A* (70%)
- DFS (100%)
- Analysis (85%)
- Report (30%)

3. **Lý Nhật Nam 20204886** nam.ln204886@sis.hust.edu.vn
   - RBFS (80%)
   - Analysis (10%)
   - Report (20%)
4. **Lê Thảo Anh 20200054** anh.lt200054@sis.hust.edu.vn
   - A* (30%)
   - Video (100%)
   - Report (15%)
5. **Đỗ Xuân Phong 20219701** phong.dx219701@sis.hust.edu.vn
   - RBFS (20%)
   - Slide (100%)
   - Report (15%)

8. List of bibliographic references

- TED-Ed's video: https://youtu.be/6sBB-gRhfjE
- PySimpleGUI's demo program for `play.py`: https://github.com/PySimpleGUI/PySimpleGUI/blob/master/DemoPrograms/Demo_Matplotlib_Embedded_Toolba

9. Hyperlinks of our project

- GitHub repository: https://github.com/htnminh/AI-intro-project
- Website: https://htnminh.github.io/AI-intro-project/
- Demo video: https://drive.google.com/drive/folders/1umrgx2-0w48aSoWVvgOEyPI1revnNV8j?usp=sharing
- Report's Notion page: https://htnminh.notion.site/The-Penniless-Pilgrim-Riddle-e3bbfaf5d7b949fdadf5a898df1f8883