Recursive Backtracker

### Conceptual Overview

Recursive Backtracker algorithm makes a maze by carving passages between adjacent cells, starting from a random initial cell. It follows these steps:

1. It starts from a random cell
2. Then, it randomly chooses a neighbour cell that hasn’t been visited yet
3. It moves to the selected cell: It carves a passage to the selected cell and then marks it as ‘visited’
4. Now, it repeats steps 1-3 until the algorithm completes
5. If it reaches a cell with no unvisited neighbours, it goes back to the last cell that has unvisited neighbours and continues the process from that point on
6. The algorithm completes when it backtracks to the starting cell and there are no unvisited neighbours left in the whole maze

### Algorithmic Details

The maze is represented as a 2D grid of cells.

Each cell can either be a wall (0) or a passage (1).

The function carve\_passage\_from(cx, cy, grid) is responsible for carving passages.

Breakdown of the key parts:

* At first, it creates a completely full canvas of walls, the entire canvas is represented by 0’s.
* A random starting cell is then chosen randomly with coordinates start\_x and start\_y
* carve\_passage\_from(cx, cy, grid) has this logic:
  + First, it generates a list of potential neighbouring cells .
  + It then randomly shuffles these directions to ensure random maze paths.
  + After that, it checks every single neighbour:
    - If the neighbour is within bounds and is a wall (grid[nx][ny] == 0), and the next cell in the same direction is also a wall, a passage is carved:
      * It marks the current neighbour and the next cell in that direction as a passage (1).
      * It then recursively calls carve\_passage\_from for the next cell in that direction.

Implementation in my code

486 *# Recursive Backtracker algorithm for maze generation*

487 def recursive\_backtracker(height, width):

488 def carve\_passage\_from(cx, cy, grid):

489 directions = [(cx - 1, cy), (cx + 1, cy), (cx, cy - 1), (cx, cy + 1)]

490 random.shuffle(directions)

491 for (nx, ny) in directions:

492 if 0 <= nx < height and 0 <= ny < width and grid[nx][ny] == 0:

493 if 0 <= nx + (nx - cx) < height and 0 <= ny + (ny - cy) < width and grid[nx + (nx - cx)][ny + (ny - cy)] == 0:

494 grid[nx][ny] = 1

495 grid[nx + (nx - cx)][ny + (ny - cy)] = 1

496 carve\_passage\_from(nx + (nx - cx), ny + (ny - cy), grid)

497 maze = [[0] \* width for \_ in range(height)]

498 start\_x, start\_y = random.randint(0, height - 1), random.randint(0, width - 1)

499 maze[start\_x][start\_y] = 1

500 carve\_passage\_from(start\_x, start\_y, maze)

501 return maze

Thanks to [Aryan Abed-Esfahani on medium](https://aryanab.medium.com/maze-generation-recursive-backtracking-5981bc5cc766) for the core ideas about this algorithm.

Prim’s Algorithm

Conceptual Overview

1. It first starts from a single cell and adds its walls to the list
2. After that, it chooses a random wall from the list
3. If the wall separates a cell in the maze from a cell outside the maze, then remove the wall and add the new cell to the maze.
4. Then add the walls of the new cell to the list.
5. Repeat steps 1-4 until all cells are in the maze

Algorithmic Details

* At first local\_maze grid is initialised with all the cells as 0 (walls) of given height and width
* Visited Set and Walls Set: Two sets are maintained - visited for cells included in the maze, and walls for the walls to be considered.
* The starting cell is added to the list, and so are its walls added to a walls list
* The algorithm iterates through the walls list
  + A random wall is selected.
  + The wall's neighbouring cells are examined:
    - If exactly one of the neighbouring cells is visited, the wall is removed (marked as a passage in local\_maze), and the other cell is added to visited.
    - The walls of the newly added cell are added to walls, except those that would lead back into the maze.
  + The selected wall is then removed from the walls set.
* Completion: The process continues until the set walls is empty, meaning all cells are part of the maze.

Implementation in my Code

503 *# Prim's algorithm for maze generation*

504 def prims\_algorithm(height, width):

505 local\_maze = [[0] \* width for \_ in range(height)]

506 visited = set()

507 walls = set()

508 start = (0, 0)

509 visited.add(start)

510 local\_maze[start[0]][start[1]] = 1

511 walls.update({(0, 1), (1, 0)})

512 while walls:

513 wall = random.choice(list(walls))

514 x, y = wall

515 neighbors = [(nx, ny) for nx, ny in [(x-1, y), (x+1, y), (x, y-1), (x, y+1)] if (nx, ny) in visited]

516 if len(neighbors) == 1:

517 nx, ny = neighbors[0]

518 local\_maze[x][y] = 1

519 visited.add((x, y))

520 for dx, dy in [(x-1, y), (x+1, y), (x, y-1), (x, y+1)]:

521 if 0 <= dx < height and 0 <= dy < width and (dx, dy) not in visited:

522 walls.add((dx, dy))

523 walls.remove(wall)

524 return local\_maze

Thanks to [vishal rana on medium](https://vishalrana9915.medium.com/understanding-prims-algorithm-e6514a6e483c) about the core ideas of Prim’s algorithm.

As you can see in his article, he doesn’t even mention using it for mazes. That’s exactly why I chose to use this algorithm, it isn’t suitable for mazes so I wanted to experiment with maze creation with it.

A\* Algorithm

Conceptual Overview

A\* finds the shortest path from a start node to a target node. The key aspects of A\*:

* The heuristic function estimates the cost to reach the target from a node.
* A cost function keeps track of the total cost from the start node to the current node.
* The priority queue stores nodes to be explored, prioritised by the estimated total cost (actual cost so far + heuristic).

Algorithmic Details

* heuristic(a, b) calculates the estimated cost from node a to node b. My function uses manhattan maps which are good for mazes
* get\_neighbors(pos) returns accessible neighbouring cells in the maze, making sure that they are within bounds and not walls
* Priority Queue (frontier): Nodes are processed based on their priority, which is the sum of the actual cost so far and the heuristic cost to the target
* Two dictionaries track the best path to each node (came\_from) and the cost to reach each node (cost\_so\_far)
* Pathfinding Process:
  + Add the start node to the frontier with a priority of 0.
  + While the frontier is not empty:
    - Get the node with the lowest priority
    - If it's the target node then the loop is broken
    - For every neighbour that the current node has it calculates the new cost to reach this neighbour
      * If the new cost is lower or if the neighbour has not been visited before, it then updates cost\_so\_far and came\_from
      * At last, it adds the neighbour to the frontier with the new updated priority
* Finally, it re-creates the path again, starting from the target node and it then traces back using came\_from to form the path

Implementation in my code

275 *# A\* search algorithm for pathfinding in maze*

276 @staticmethod

277 def a\_star\_search(maze, start, end):

278 def heuristic(a, b):

279 return abs(a[0] - b[0]) + abs(a[1] - b[1])

280

281 def get\_neighbors(pos):

282 neighbors = []

283 for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:

284 x, y = pos[0] + dx, pos[1] + dy

285 if 0 <= x < len(maze) and 0 <= y < len(maze[0]) and maze[x][y] == 1:

286 neighbors.append((x, y))

287 return neighbors

288

289 frontier = PriorityQueue()

290 frontier.put((0, start))

291 came\_from = {}

292 cost\_so\_far = {}

293 came\_from[start] = None

294 cost\_so\_far[start] = 0

295

296 while not frontier.empty():

297 current = frontier.get()[1]

298 if current == end:

299 break

300 for next in get\_neighbors(current):

301 new\_cost = cost\_so\_far[current] + 1

302 if next not in cost\_so\_far or new\_cost < cost\_so\_far[next]:

303 cost\_so\_far[next] = new\_cost

304 priority = new\_cost + heuristic(end, next)

305 frontier.put((priority, next))

306 came\_from[next] = current

307

308 current = end

309 path = []

310 while current != start:

311 if current not in came\_from:

312 return None

313 path.append(current)

314 current = came\_from[current]

315 path.append(start)

316 path.reverse()

317 return path

Thanks to [PandaMan on medium](https://panda-man.medium.com/a-pathfinding-algorithm-efficiently-navigating-the-maze-of-possibilities-8bb16f9cecbd) for helping me out with the core ideas behind A\* algorithm