Lab01: Search Strategies

Courses `CSC14003`: Intro to Artificial Intelligence
`18CLC6`, `FIT - HCMUS`.
`31/07/2020`

This is an INDIVIDUAL assignment:

- `18127231`: Đoàn Đình Toàn (GitHub: @t3bol90)

About this project:

Original problem description:
http://ashishgupta.me/articles/2019-05-02-AI-Search-Algorithms-Implementations/

In this project, i need to implement 5 algorithms/ strategies to deal with the maze problem.

The **Rubric** from our teacher:

Criteria	**Points**
Implement the search strategies	2.5 (0.5pt each)
Correct list of explored nodes (in correct order)	5 (1pt each)
Correct list of nodes on the path found (in correct order)	2.5 (0.5pt each)

Each search strategies need to keep the correctness of explored nodes' order and path's order.
For more details description for each strategies:

Strategies	**Implement**	**Correct list of explored nodes**	**Correct list of nodes on the path found**
Breadth-first search (**BFS**)	[x] Done	[x] Done	[x] Done
Uniform-cost search (**UCS**)	[x] Done	[x] Done	[x] Done
Iterative deepening search (**IDS**)	[x] Done	[x] Done	[x] Done
Greedy-best first search (**GBFS**) with Manhattan distance as heauristic	[x] Done	[x] Done	[x] Done
Graph-search A* (**Astar**) using the same heuristic as above	[x] Done	[x] Done	[x] Done

For testing propose, i have write a shell script for testing multiple file (as a batch) in `INPUT` folder and give a output file to `OUTPUT` folder.

Enviroment

This project use `python3.8` for algorithms implementation and `linux shell script` for testing, using `PyCharm PE` on `Ubuntu 20.04 LST`.

Install and Run

Tree view

```
├── batch_test.sh
├── DOCUMENT
│   ├── report.md
│   ├── report.pdf
│   └── statement.pdf
├── INPUT
│   ├── input02.txt
│   ├── input03.txt
│   └── input.txt
├── LICENSE
├── OUTPUT
│   ├── input02.txt
│   ├── input03.txt
│   └── input.txt
└── SOURCE
    ├── __init__.py
    ├── maze.py
```

- `SOURCE`: Python file (`maze.py`) for program running.
- `INPUT`: some example mazes in addition to the given example.
- `OUTPUT`: corresponding search results to the mazes in the `INPUT` folder.
- `DOCUMENT`: statement and report file.

Manual input

From project directory, start program by:

```
python ./SOURCE/maze.py
```

This will wating for your input from Keyboard:

The maze is represented in a text file as follows.

- The first line includes a positive integer N only, which indicates the size of the maze.
- Each of the next N × N lines contains an adjacency list of node i. Nodes in the list are unordered and separated by white spaces.
- The last line includes a non-negative integer E (from 0 to N 2 − 1), which indicates the exit.

Then it will print result to your console.

Input from file

```
python ./SOURCE/maze.py -i <input-file-directory>
```

Output from file

```
python ./SOURCE/maze.py -o <output-file-directory>
```

Batch test:

First you need to put all of your input file into `INPUT` folder, then start from project directory:

```
chmod +x ./batch_test.sh
./batch_test.sh
```

Key fuction:

Graph representation:

I'm using a 2D array for graph represent:

```
graph = [[[] for _ in range(n)] for _ in range(n)]
    for j in range(n):
        for i in range(n):
            index = _idx_dec(i, j)
                adj_list = list(map(int, input().split()))
            for element in adj_list:
                graph[i][j].append(element)
```

Search strategies:

BFS - Breadth-first search:

```
def bfs(graph, start, goal):
```

I'm using `deque` from `collections` instead of `queue.queue` for 2 reasons:

- `deque` is faster, `queue` or `Queue` support concurrency process but in this project we did not using any extend parrallel or concurrency. So `deque` is enough.
- source code need to be consistency, but it's ugly when you are using:

  ```
  while <your data structure>:
  ```

  but for `queue.queue`:

  ```
  while <your queue>.empty() is False:
  ```

  Ew :<.

To make sure that you are not visit the node that vistited, i use an array to mark it:

```
visited = [[False for _ in range(n)] for _ in range(n)]
```

UCS - Uniform-cost search:

```
def ucs(graph, start, goal):
```

Once again i use `heapq` instead of `queue.PriorityQueue` for same reason.

To calculate `g(neighbor node) = g(expanding node) + 1`, i use a array named `dist` (distance):

```
dist = [[INF for _ in range(n)] for _ in range(n)]
```

Because maximum length of the path is the total node of the maze so:

```
INF = n*n + 1
```

IDS - Iterative deepening search:

By the textbook's pseudocode:

function ITERATIVE-DEEPENING-SEARCH(*problem*) **returns** a solution, or failure
  for *depth* = 0 **to** ∞ **do**
    result $\leftarrow$ DEPTH-LIMITED-SEARCH(*problem*, *depth*)
    if *result* $\neq$ cutoff **then return** *result*

I implemented a DLS and DLS-RECURSIVE:

```
def dls(graph, start, goal, limit):
    ...
    explorered_node = []
    visited = [[False for _ in range(n)] for _ in range(n)]
    cost = 0

    def dls_recusive(_start, _goal, _limit):
        nonlocal explorered_node, cost, dp_path
        ...
```

DLS-RECURSIVE, implement with recursive as stack, will run until it found goal state or deep limit exceeded.
DLS will return path, explorered node, cost and the state of that limit (found or not found).

IDS will run a for loop with a limit as iterater variable and call DLS with that limit until we found goal state or run out of graph's depth (by check explorered node of 2 times).

```
def ids(graph, start, goal):
    explorered_node = []
    limit = 0
    total_cost = 0
    preex = []
    while True:
        path, exnode, cost, is_found = dls(graph, start, goal, limit)
        limit += 1
        if preex == exnode:
            break
        explorered_node.append(exnode)
```

```
        total_cost += cost
        if is_found:
            break
        preex = exnode
    return path, explorered_node, total_cost
```

GBFS - Greedy-best first search:

Once again i use `heapq`, instead of `g(n)`, GBFS use `h(n)` as a heauristic funtion and weight for `heap`.

```
def manhattan_L1_distance(s, f):
    sx, sy = _idx_enc(s) # For index encoding, eg: 0 -> (0, 0)
    fx, fy = _idx_enc(f)
    return abs(sx - fx) + abs(sy - fy)
```

```
def gbfs(graph, start, goal):
```

So the backbone is the same as `UCS` but i replace `g(n)` by `h(n)`

Astar - Graph-search A*:

A-star is the same backbone as `UCS` and `GBFS` but with `f(n) = g(n) + h(n)`:

```
def astar(graph, start, goal):
    ...
    ...
    ...
    heappush(pq, (h + g, neighbor))
```

Utils function:

Result view of each algorithm:

```
def result_view(function, name, _graph, _start_point,
                _exit_point):
    path, exed_node, cost = function(_graph, _start_point, _exit_point)
        print(name)
        print(f"Start: {_start_point} | Goal: {_exit_point}")
        print("Path:", path if path[-1] != -1 else "No path",
          f"| Cost {(len(path) - 1) if path[-1] != -1 else 0}")
        print("Explored node:", exed_node, f"| Time {cost}")
```

Thanks to `python`, we just need to call `result_view` and pass the `function` to it.

Path backtracking to get path from a to b:

```
def path_backtrack(dp_path, start, goal):
    path = []
    def path_travelsal(s, f):
        nonlocal path
```

```
        gx, gy = _idx_enc(f)
        sx, sy = _idx_enc(s)
        if s == f:
            path.append(f)
        else:
            if dp_path[gx][gy] == -1:
                path.append(-1)
            else:
                path_travelsal(s, dp_path[gx][gy])
                path.append(f)
    path_travelsal(start, goal)
    return path
```

Index decoder and encoder:

```
def _idx_dec(row, col):
    return row + col * n


def _idx_enc(index):
    return index % n, index // n
```

It switch between the label of the node and it's (x,y) in the maze.

Batch test:

```
for file in ./INPUT/*
do
  python ./SOURCE/maze.py -i $file -o "./OUTPUT/`basename "$file"`"
done
```

It's easy to test your own test and compare with my program.

Conclusion:

- What a cool project! Give a special thanks to our lecturers, lab instructors and teacher assistances at `VNU-HCMUS`. I really appreciate your help with my project!
- In this problem, `UCS` and `BFS` has the same path and exploring node because of when convert the maze to graph - it's a weighted graph with equal positive weight (=1).
- `DLS` of `IDS` can implemented by using stack (in place), you don't need to use recursion. (I've tried, but i refer recursion, it's more clean).

References:

- Textbook: Artificial Intelligence - A Modern Approach (3rd Edition).
- Raw testcase of the problem (I just grap the testcase and pipe it to a txt, i do not get this implementation to my source code!) Github
- argparse - for file parser.
- heapq and deque - faster than queue.

Have a Great Day