Lab 2

姓名：游灏溢
班级：F1903302
时间：15/10/2021

Abstract

This lab is mainly about the graph search algorithms. Based on the lab1, we propose 4 new algorithm, UCS, DFS with iteration deepening, greedy search and A^*. Besides, we also discuss others about dfs and heuristics.

Content

Exercise1

Number of 'Lakes'

Based on the dfs algorithm, we propose the following algorithm.

Algorithm 1: Search for the number of lakes

Input: Problem
Output: Number of lakes

```
points_set ← DFS(get_start()) #get all the points in the graph
arrived_points ← None
number ← 0
for v ∈ points_set do
    if v ∈ arrived_points then
        arrived_points.add(v) neighbors ← get_successor(v, wall = 1)
        for u ∈ neighbors do
            if u ∈ arrived_points then
                dfs(u) and update the arrived_points.
                number ← number + 1
return number
```

It's necessary to mention that since we are not able to access to visit the layout, we cannot judge the points to be lake or land. As a results, we can only obtain all points from DFS the start points and run the DFS on the lakes based on the $get\_successor()$.

If we can access the layout, the algorithm can be modified.

DFGS with Iterative Deepening

Increasing Function

Here we define two types of increasing function.

- polynomial fomula: Here we use $f(x) = x + 1$ since we begin counting at $x = 0$.
- exponential fomula: Here we use $f(x) = 2^x$.

```python
def increase_function1(times:int):
    return times+1
def increase_function2(times:int):
    return 2**times
```

DFGS Algorithm

Algorithm 2: DFGS with Iterative Deepening

```
Input: Problem
times ← 0
while true do
    point_stack ← Stack()
    max_depth ← increasing_function(times)
    point_stack.push((get_start(), depth))
    while not point_stack.is_empty() do
        u ← point_stack.pop()
        if is_goal(u) then
            return
        neighbor ← get_successor(u, wall = 0)
        if u.depth< max_depth then
            for v ∈ neighbors do
                point_stack.push(v, u.depth + 1)
    times ← times + 1
```

When running the code, I discover that if the path from start to goal is too long, it will cost plenty of time. So we improve the algorithm by recording the points that has visited.

Time and Memory Analysis

Time consumption

There are many methods to count the time. For instance, the python has a function $clock()$ to obtain the system time. However, when the scale of the layout is small, the difference cannot be distinguish by the gap of seconds. And the running time of the code is also based on the utility of the cpu and its state.

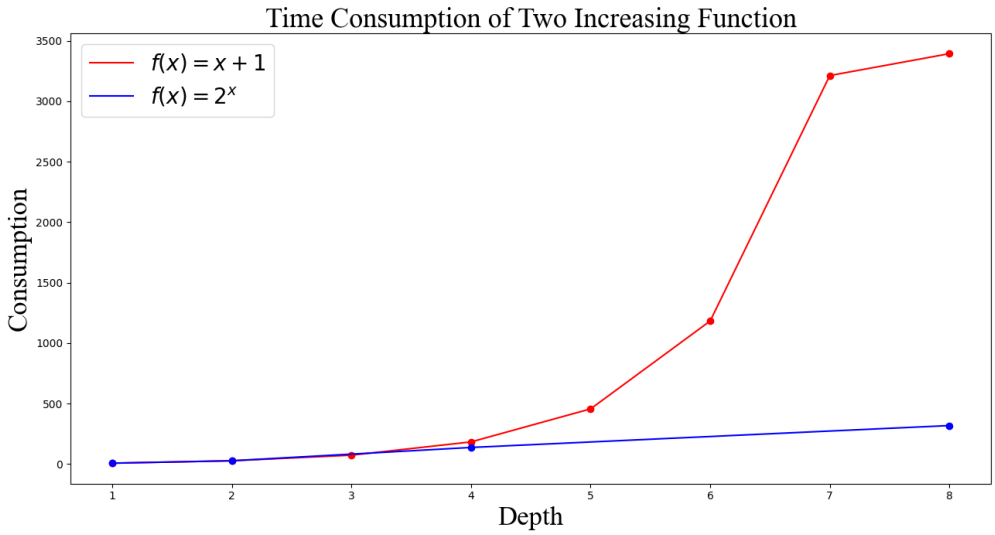
So here I propose another method to count the number of time from the number of instructions the algorithm has. And the number of instructions is mainly based the number of the iterations, so the variable time add 1 everytime it enter an iteration.

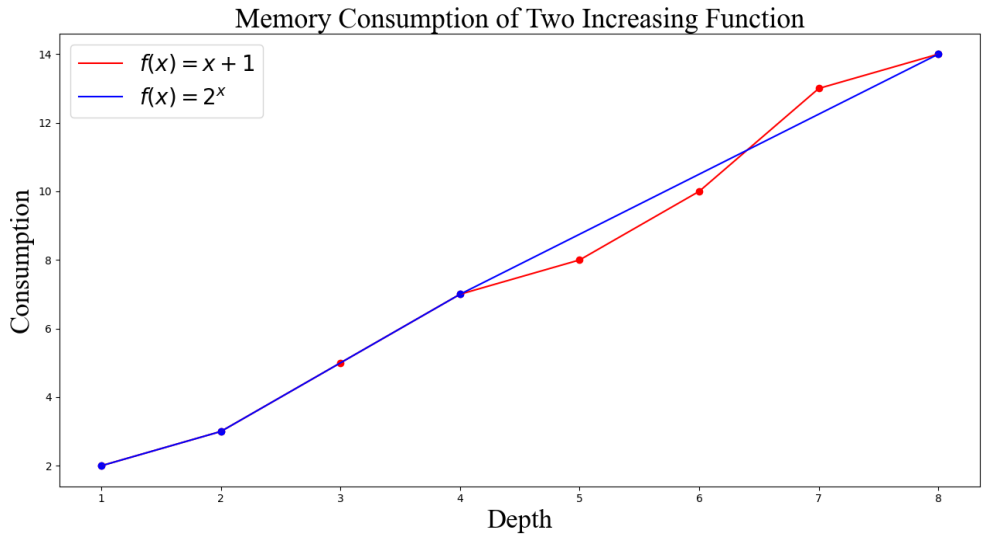
Memory consumption

Consider of the variables in the algorithm, we can find the usage of memory is mainly from number of the items in data structure **Stack**. So we use the maximum number of items in **Stack** to represent the memory consumption.

Performance analysis.

Based on the Maze_lab2_1_1.lay, we plot the results below.





As for the memory consumption, the difference is very little between them. For the time consumption, when depth is small, the difference is also little, but when depth begin to increase, the first function will cost much more time than the other. That's because the function 1 costs lots of time in depth from 5 to 7, which is useless for searching the goal, but after searching for depth 4, function 2 directly jumps to the depth 8, which reduces plenty of time.

However, only one layout cannot represent the performance of these two functions in different scale of data. So I design new layouts and test functions on different scale of layout. The performance of these two increasing functions are shown below.

Layout name	Maze_lab2_1_1.lay	Maze_lab2_1_2.lay	Maze_lab2_1_3.lay
distance from start to goal	8	5	16
path distance(function1)	8	5	16
total time consumption(function1)	3394	409138	38797667
total memory consumption(function1)	14	8	35
path distance(function2)	8	7	16
total time consumption(function2)	319	3091	20864403
total memory consumption(function2)	25	15	35

To test the performance on different layout, I design two new layout with shorter distance (**Maze_lab2_1_2.lay**) and longer distance (**Maze_lab2_1_3.lay**). For more details of the layouts, please refer to the layout files. To test these layouts, you are supposed to modify several parameter of the **main.py** and **layout.py**

```
def parse_args():
    ...
    parser.add_argument("--layout_name", type=str, default="Maze_lab2_1_2")
#or parser.add_argument("--layout_name", type=str, default="Maze_lab2_1_3")
```

main.py

```
LAYOUTS = {
    ...
  "Maze_lab2_1_1": "./examples/Maze_lab2_1_1.lay",
  "Maze_lab2_1_2": "./examples/Maze_lab2_1_2.lay",
  "Maze_lab2_1_3": "./examples/Maze_lab2_1_3.lay",
  "Maze_lab2_2_1": "./examples/Maze_lab2_2_1.lay",
  "Maze_lab2_3_1": "./examples/Maze_lab2_3_1.lay",
}
```

layout.py

Comparing two increasing functions, we can figure out that function2 increases much faster than function1. So we can find several difference

- For memory consumption, the difference between two functions is little, function2 may use larger memory. That's because maximum usage of memory will happen for the last time when the depth limit is the largest. And the limit depth of function2 will exceed the factual distance more than function1.
- For time consumption, when the scale is small, we prefer the function1, otherwise we prefer the function2. That's because we the scale is small, function1 will reach the distance more accurately while function2 will exceed the limit. However, when the scale is large,

function1 will waste large of time on the depth shortest than the distance, but function2 will reach the distance limit faster.

As a result, the memory consumption is closed for two functions. Only if you have very limited memory you need to use the functions increasing slower. Otherwise you should pay more attention on time consumption. When the scale is small, we prefer the function with lower increasing speed, otherwise we perfer the higher increasing speed one as scale is large.

Exercise2

Uniform-Cost Graph Search(UCGS)

The main part of the algorithm is similar to DFGS. We only need to substitute the PriorityQueue for Stack, because we need to fetch the shortest path in the container.

Using heap (PriorityQueue) requires the strict partial order between items in heap. However, elements combining distance, points and path cannot be compared in **python**. So I construct a new class to solve this.

```
class UCS_item:
    def __init__(self,distance,position,path=[]):
        self.distance=distance
        self.path=path
        self.position=position
    def __lt__(self,other):
        return self.distance<other.distance
```

The distance represents the cost from the start point to this position.

```
now=PQ.get()
successor=problem.get_successors(now.position,0)
for item in successor:
    ...
    PQ.put(UCS_item(now.distance+item[2],item[0],new_path))
```

Greedy Graph Search(GGS)

Similar to the UCGS, the only difference is to compare the expectation of the distance from goal to this point.

```
class heuristc_Euclidean_item:
    def __init__(self,distance,position,path=[]):
        self.distance=distance
        self.path=path
        self.position=position
    def __lt__(self,other):
        return self.distance<other.distance
...

PQ.put(heuristc_Euclidean_item(Heuristic1(item[0],goal),item[0],new_path
))
```

As for the expectation of the distance from goal to the point, we use the heuristic function, which is to count the Euclidean distance.

```
def Heuristic1(state1, state2):
    return math.sqrt((state1[0]-state2[0])**2+(state1[1]-state2[1])**2)
```

Exercise 3

A^* Graph Search

The main part of the algorithm is similar to above 3 algorithm, but compare the distance from start to this point adding expectation distance from goal to this point.

```
class A_star_item:
    def __init__(self,priority,distance,position,path=[]):
        self.priority=priority
        self.distance=distance
        self.path=path
        self.position=position
    def __lt__(self,other):
        return self.priority<other.priority
    ...
Heuristic=Heuristic1
    ...

PQ.put(A_star_item(Heuristic(item[0],goal)+now.distance+item[2],now.dist
ance+item[2], item[0],new_path))
```

Besides, the heuristic function has more choice, only do we need to guarantee the consistence of the heuristic function. So we can use the other heuristic function, such as Manhattan distance

```
def Heuristic2(state1, state2):
    return abs(state1[0]-state2[0])+abs(state1[1]-state2[1])
def Exercise2_3_1(problem):
    ...
    Heuristic=Heuristic2
```

It also important to mention that we only need to guarantee the consistence of the heuristic function. Under the condition of that, the closer between this two value, the less time the program will consume. So Heuristic2 will have a better performance than Heuristic1.

Changing Heuristics Function

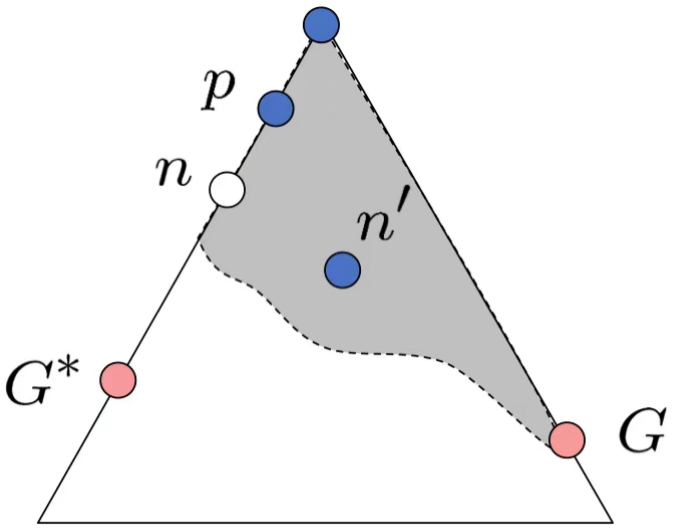
As for the new heuristic function:

$dis(P,G)=|x_p-x_g|+|y_p-y_g|-\mathbb{I}\{|x_p-x_g|\neq|y_p-y_g|\}$ and function $h_3(P)=dis(P,goal)$.

Firstly, h_3 do not have consistence property. For instance, goal=(1,1), P=(2,2), G=(2,3) and there is no wall. We can find $h_3(P) = 1, h_3(G) = 3, h_3(G) - h_3(P) = 2$ but $cost(P\ \ to\ \ G) = 1 < h_3(G) - h_3(P)$.

However, the consistence requirement is not necessary. Although h_3 do not have property, using h_3 as heuristic function in the A^* still get an optimal solution.

Proof:



If the solution is not optimal, consider the search tree from start point. Let $f(x) = cost(start\ \ to\ \ x) + h_3(x)$

So there are some n on path to G^* that hasn't enter into heap when we pop the G. Take the highest n in the search tree. Let p be the ancestor of n that was in the heap when n' was popped.

Then we have following two property:

- $f(p) \leq f(n) + 1$
- $f(n) \leq f(n') - 2$

So we can have $f(p) < f(n')$. So p must be expanded before n', contradiction!

<*> For the two property, the reason is below

- Consider the indicator function $\mathbb{I}$ in calculate $dis(goal, n), dis(goal, p)$. Let $g(p) = \mathbb{I}\{|x_p - x_{goal}| \neq |y_p - y_{goal}|\}$. We know the Manhattan distance satisfies consistence and $h_3 + g$ is exactly the Manhattan distance. So we have $(h_3 + g)(p) - (h_3 + g)(n) \leq cost(p\ \ to\ \ n) = 1$. So $f(p) - f(n) \leq g(n) - g(p) \leq 1$.
- Since the graph is a grid. For two path P and P^* from $Start$ to $Goal$. If P^* is optimal while P is not. We have $len(P) \geq len(P^*) + 2$. Assume the actions follow P is $a(1), a(2) \dots a(len(P)), a(i) \in \{w, s, e, n\}$.
  $\sum \mathbb{I}\{a(i) = w\} - \sum \mathbb{I}\{a(i) = e\} = \sum \mathbb{I}\{a'(i) = w\} - \sum \mathbb{I}\{a'(i) = e\}$ and
  $\sum \mathbb{I}\{a(i) = n\} - \sum \mathbb{I}\{a(i) = s\} = \sum \mathbb{I}\{a'(i) = n\} - \sum \mathbb{I}\{a'(i) = s\}$.
  So $D = \sum \mathbb{I}\{a(i) \in \{w, s, e, n\} - \sum \mathbb{I}\{a'(i) \in \{w, s, e, n\}$ is an even number. And we know $D > 1$ so $D >= 2$, so that $f(n) \leq f(n') - 2$.