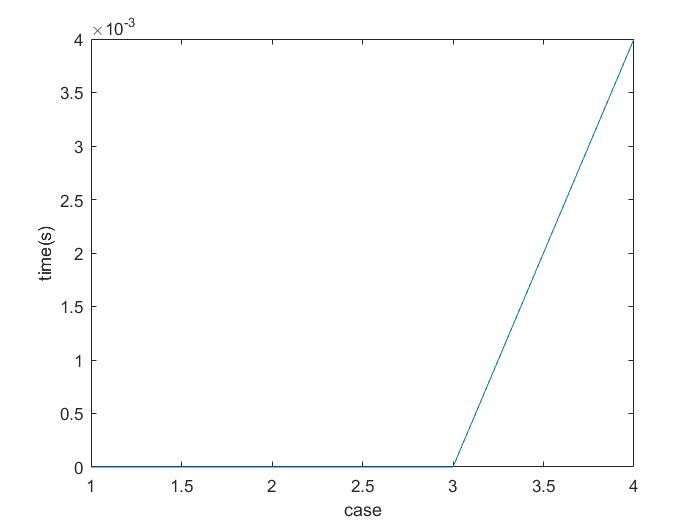
|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Case | *#* of food | GD | | | DP | | |
| *Total distance* | CPU  time  (s) | Memory  (KB) | *Total distance* | CPU  time  (s) | Memory  (KB) |
| case1 | 4 | 20 | 0s | 11.692MB | 14 | 0s | 11.692MB |
| case2 | 1000 | 154742 | 0s | 11.692MB | 150426 | 0s | 11.692MB |
| case3 | 10000 | 1550932 | 0s | 11.812MB | 1512940 | 0s | 11.808MB |
| case4 | 100000 | 30931698 | 0.004s | 13.572MB | 30092520 | 0.004s | 13.572MB |

B02703036 財金四 林耘寬

1.

2.



GD and DP have the same lines since their CPU time are the same.

3.

d [i] = min{d[j] + dist2origin(j+1) + dist(j+1, i) + dist2origin(i)}

Consider the formula above, d[i] is the shortest distance after picking up the ith food.

Here, we suppose that i is the last food and j is the second last food that the ant chooses to go back to origin. Now, let d[i] be the optimal solution and suppose d[j] is not the optimal solution and the ant chooses to go back at food j+1.

If d[j+1] is the shortest distance after picking up the i+1 th food. Consider the formula again, we can easily find out that dist[j,j+1] < origin(j). We also know that

dist[j+1, j+2] > origin(j+2). If not, the ant might not choose to go back at food j+1.

Thus, d[i] will be smaller if we choose to go back at food j+1, instead of j. This reaches the contradiction since d[i] is the optimal solution. Then, d[j] is also the optimal solution that the ant will choose to go back at food j. By definition, this question has the optimal substructure.

4.

for(int j = 1; j <= n; j++){

int cur\_cap = food[j].weight;

int temp\_dis;

for (int target = j-1; cur\_cap <= cap && target >= 0; target--){

temp\_dis = food[target].distance + origin(food[target+1]) + dist(target+1, j, food) + origin(food[j]);

cur\_cap += food[target].weight;

if(temp\_dis < food[j].distance){

food[j].distance = temp\_dis;

food[j].back = target;

}

}

}

You can see the part of the source code. There are two loops in the function. The worst case will become O(n^2)

Best case will be O(n) if the weight of each item is really heavy.

However, we usually won’t reach the worst case because the capacity is not large enough to let us carry too much food.

5.

Consider the formula in question 3, suppose the bag isn’t full after collecting food j, but it is full after picking up food j+1. Then, by the definition of greedy algorithm. Using easy algebra, we can simply find out that the difference between going back at food j and food j+1 is origin(j) + dist(j+1, j+2) – dist(j, j+1\_ - origin(j+2)

Hence, if the equation above is less than zero, the greedy algorithm fails.

6.

for (int k = 1; k <= n; k++){

cur\_gdcap += foodGD[k+1].weight;

if (cur\_gdcap > cap){

foodGD[k].distance = foodGD[temp].distance + origin(foodGD[temp+1]) + dist(temp+1, k, foodGD) + origin(foodGD[k]);

foodGD[k].back = temp;

temp = k;

cur\_gdcap = foodGD[k+1].weight;

}

else if (k == n){

foodGD[k].distance = foodGD[temp].distance + origin(foodGD[temp+1]) + dist(temp+1, k, foodGD) + origin(foodGD[k]);

foodGD[k].back = temp;

}

else

continue;

}

By the source code above, we can find that the worst, best case of GD algorithm is Theta(n).