UNIVERSITY OF SCIENCE
FACULTY OF INFORMATION TECHNOLOGY

Searching for the Knapsack

Introduction to Artificial Intelligence

TRIỆU NHẬT MINH - 21127112
BÙI ĐỖ DUY QUÂN - 21127141
HOÀNG ĐỨC VIỆT - 21127203

Lecturers:

Nguyễn Tiến Huy

Nguyễn Trần Duy Minh

26th April 2023

Contents

1 Assignment Plan & Self-assessment

Student ID	Work	Completion
21127112	Genetic algorithm, Video demo	100%
21127141	Branch and bound, ClassData & GenerateTest, Report	100%
21127203	Brute force, Local beam algorithm, Report	100%

2 Device used for testing

- Device: Dell Vostro 5502
- Processor: 11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz (8 CPUs), 1.4GHz
- Memory: 16GB Ram
- Operating System: Windows 10 Home Single Language 64-bit (10.0, Build 19045)
- Compiler: Python 3.10.11

3

4 Algorithm 1: Brute force searching

4.1 Description

- Checking all possible states to find an optimal solution.

4.2 Pseudo code

Algorithm 1 Brute force searching

```
1: for each State in all States do
2:     if (State is not overweight, State includes all classes, the value of State is greater than the previous one) then
3:         Result = State
4:     end if
5: end for
```

4.3 Explanation

a General explanation

- We add each item to the knapsack, so this presents a new state.
- Check if this new state meets the conditions and is better than the previous, so the result will be updated.

b Deal with big size of items

- We calculate that our device can solve 200.000 problems per second.
- The original brute force searching runs in exponential time $O(2^n)$.
- For example: If n = 30, the algorithm must walk through $2^{30} = 1073741824$ cases. It takes nearly 1 hour and 30 minutes to solve.
- Another example: If n = 40, there are $2^{40} = 1099511627776$ cases. It takes up to nearly 64 days to complete.

c Optimization

- We already knew that brute force is an algorithm to find the best answer. However, the running time is really slow, so our team will optimize it by removing as many wrong cases as well.
- Before running the algorithm, we will sort all the states in ascending order of weight and descending order of value.

- When the algorithm is being executed, if the current weight is greater than the capacity, we can break it instantly and move to check the next possible state.
- Finally, we will sort the resulting state back to the original position.

4.4 Time complexity

- $O(2^n)$
- Where: n is the number of items.

5 Algorithm 2: Branch and bound

5.1 Description

- Using an **admissible function** to find an upper bound of each node (item) to decide whether we choose that node (item) or not.
- Each node has only been traveled only once.

5.2 Pseudo code

Algorithm 2 Branch and bound

```
1: for each item in Knapsack do
2:    if upper bound function of this item is satisfied then
3:       add this item to Knapsack
4:    end if
5: end for
```

5.3 Explanation

a General explanation

- Each item has 2 states, choose or not choose.
- We check the upper bound of each to decide whether we choose or not.
- After the decision, move to decide on other items.

b Formula to choose items

- We define the formula to find the upper bound of the next item:

  Upper bound[i] = Heuristic[i+1] + (Capacity - Weight[i+1]) * Heuristic[i+1] / Weight[i+1]

- We will check the upper bound between when we choose the next item and when we do not choose it, and we will choose the greater one to decide whether should we add the item or not.
- Then, we will sort all the states base on the ratio (heuristic / weight) in descending order.
- Finally, we sort again to its original position.

5.4 Time complexity

- $O(n \log n)$
- Where: n is the number of items.

6 Algorithm 3: Local beam search

6.1 Description

- This algorithm is quite similar to **Brute force**. However, at each level, we only keep **K** nodes, which have the best heuristic value.

6.2 Pseudo code

Algorithm 3 Local beam search

```
1: while BaseState not empty do
2:    Create an empty NewState
3:    for each State in BaseState do
4:       Make new State
5:       if calcWeight(State) > Data.Capacity then
6:          Do not add the State to NewState
7:       else
8:          IsResult = 0 {This state is not the leaf node}
9:          Add new State to NewState
10:      end if
11:   end for
12:   if IsResult and State has all classes and Heuristics(State) > Heuristics(Result) then
13:      Result is updated
14:   end if
15:   Update NewState to BaseState
16: end while
```

6.3 Explanation

a General explanation

- When a new item is added to the base state, we check if its weight is larger than the capacity, we don't choose it.
- Since we can add an item to the state, this is not the leaf node.

b Optimization

- The base state is [0,0,...].
- The optimization is similar to **Brute force**.

6.4 Time complexity

- O($k \times n^2$)
- Where: n is the number of items, and k is the number of states.

7 Algorithm 4: Genetic algorithm

7.1 Description

- This algorithm bases on natural selection. It uses cross-over, mutations, etc to find other successors. After several generations, natural selection will find the best solution.

7.2 Pseudo code

Algorithm 4 Genetic algorithm

```
1: GeneratePopulation {Do it once}
2: for each Population in GenerationSize do
3:    while in Population do
4:       CrossOver
5:       Mutation
6:       Sort the offspring based on the fitness
7:    end while
8: end for
```

7.3 Explanation

a General explanation

- Initialize a population once.
- Crossover, Mutation, and Sort the offspring in a population.
- Repeat the process for all initialized populations.

b Generate Population

- The default population is 100.
- To optimize the result, the first state which should be added to the base state is the result of branch and bound.
- We already have n states that take each and only one item.
- We also random 100 times the population's size.
- After all, it is sorted by heuristic to find n best states for the base state where n is the population's size.

c Crossover

- 50% of the offspring is crossbred by the best previous state and other states, which is chosen randomly.

- The remainder of offspring is chosen randomly from those previous states in the range from **1** to **population size / 6**.

d Mutation

- We define the mutation rate at 20%.
- It will flip the bit at a random position of a random state. If it is not overweight, the state will be accepted to be changed.

7.4 Time complexity

- O($k \times n^2$)
- Where: n is the population size, and k is the number of generations.

8 Evaluation

Algorithm	Brute force	Branch and Bound	Local beam (k = 1000)	Genetic
Time complexity	$O(2^n)$	$O(n \log n)$	$O(n^2 \times k)$	$O(n^2 \times k)$
Speed	Slowest	Fastest	Average	Average
Stability	Yes	Yes	No	Yes
Accuracy ($n \leq 20$)	Best	Worst	Average	Average
Accuracy ($n > 20$)	Best	Average	Worst	Average

Note:

- Assumes that **n** is the number of items excepts for Genetic algorithm where **n** is the population size.
- Brute force algorithm is always the slowest, but it is the most accurate algorithm among 4.
- Branch and bound algorithm is the fastest of all. However, the accuracy is not stable.
- For $n \leq 20$, Local beam seems to be faster than Genetic algorithm. When n approaches 400, Genetic is likely faster.

9 Conclusion

- In conclusion, with the small data sets (smaller than 40 items), we should use **Brute force** to solve the Knapsack problem with the most accurate result.
- In terms of data sets that are bigger (from 41 to 1000), we can use the 3 remains algorithm. However, the **Local Beam** is the least optimization among the 3 algorithms.

10 Visualization

- Our project video link: YouTube

11 References

- Admissible function for branch and bound