0/1 KnapSack using Genetic Algorithm

Pratik Goyal (B18BB024)

1. Introduction

I was practicing dynamic programming questions when the local search topic was going on in the class. Most of the dynamic programming questions are either optimizations tasks or combinatorial problems. I realised many of these problems can be implemented as search tasks and could be solved using search algorithms. And in the real world where users are not concerned about finding the most optimal solutions, local searches can solve these questions very efficiently.

One such problem is the 0/1 Knapsack problem. It is a combinatorial optimization task. I have implemented genetic algorithm (local search) to solve the 0/1 knapsack problem. A dynamic programming approach gives the optimal solution, however genetic algorithms give a suboptimal solution. One potential benefit of using genetic algorithms could be that we may increase the number of items or the capacity of the knapsack. Dynamic programming approach fails to do that. Furthermore we can control the number of iterations and hence control the time taken for the execution of the program.

The novelty lies with the fact that solving the knapsack problem using genetic algorithms will free the algorithm complexity from the knapsack size. The complexity of the dynamic programming approach of solving knapsack is directly proportional to the knapsack capacity. Which makes it infeasible to run the program for a high value of knapsack capacity. If we are not concerned about finding the optimal answer then genetic methods can be used. Furthermore, time complexity of the proposed algorithm can also be changed however that would indeed affect the final result.

2. Problem definition

We have a knapsack of capacity ‘C’. And we are given n items together with their respective weights and values. We want to choose items and put them in the knapsack such that their total weight does not exceed C. and we want to maximize the combined value of the items picked. The simpler version of the problem is called 0/1 Knapsack where the quantity of each item is restricted to 1.

In the real world the same idea can be implemented as follows: Pratik has a list of items that he wants to buy from the shopping mall. However, as he is broke he can not buy all of them. There is a price for every item and there is a satisfaction value associated with them. Satisfaction value is nothing but how useful the item is for Pratik. Now we need to find a way such that Pratik could get maximum satisfaction from the limited money he has.

The data required is:

- Knapsack capacity = ‘C’
- Number of items = ‘n’
- An array of weights = W1, W2, W3, ……… , Wn
- An array of values = V1, V2, V3, …………., Vn

Our task is to choose k items such that:

- $\sum_{i=0}^{k} Wi \leq C$ (constraint)
- $\sum_{i=0}^{k} Vi$ is maximum possible

3. Background survey

The most simple method to solve is to consider all the possible combinations and then take the one which satisfies our constraints. This method would be exponential. As for every item we have two possible options. Either we take the item or we don't. Therefore, for n items the time complexity would be O(2^n). And to store all of them it will take O(n*(2^n)) space.

The recursive approach shows us that there are overlapping subproblems. Hence, by storing the solutions to these subproblems, recomputations can be avoided and the algorithm can be made more efficient. This is a classic case of dynamic programming where the table is constructed in a bottom-up fashion. Here we consider all the possible weights from 0 to W as columns and then consider the weight array as rows. Each cell of the dp table, dp[i][j] holds the maximum value that “j” weight can have considering only the weights from 1 to ‘i’th.

For a given weight ‘j’ and i^{th} weight, we have two possibilities

- Case 1: W_i > j:

  In this case we can not include W_i as it would exceed the capacity of the knapsack hence the current state will be the same as the state when only i-1 items were included. Hence, dp[i][j] = dp[i-1][j]

- Case 2: $W_i \leq j$

  In this case we could arrive at an optimal solution by either including or excluding the weight depending on the value gained at the end of inclusion or exclusion.
  Value obtained after exclusion: dp[i-1][j] (same as Case 1)
  Value obtained after inclusion: V_i + dp[i-1][j-W_i] (value of ith weight added to the maximum value obtained by considering i-1 items for j-W_i weight, which is precomputed in the dp table)

The final result will be stored in dp[N][W] as it denotes the maximum value that W weight can have considering(including/excluding) all the items in the knapsack.

The complexities of the dynamic programming approach (N = number of items; W = capacity of the knapsack) would be :

1. Time complexity: $O(N * W)$
2. Space complexity: $O(N * W)$

4. Discussion

If the idea of knapsack is used in real world problems to solve a combinatorial task for example: packaging problem etc. We don't need the most optimal answers. We are happy with sub optimal answers as long as it is giving satisfactory results. In my implemented method it is always possible to get better outputs by increasing the number of iterations. This does not cost any more space nor adds to the time complexity. As the genetic algorithm follows the notion of randomness, for different iterations the answer would be different. And as we will increase the number of iterations the chances of getting a better final result will increase.

5. Algorithm

As discussed earlier the most optimal algorithm is dynamic programming. If we construct this as a search task then it can be solved using local searches genetic algorithms or other local searches might work too. However, we will need to compromise with the optimality of the solution.

Dynamic programming approach will not be feasible for large inputs and large knapsack capacity however, my approach using local search does not depend on the knapsack capacity.

6. Algorithm Complexity

While implementing the algorithm I have considered a population set of size N. Where, N is the number of items available. Therefore, For ‘K’ iterations :

1. Time complexity: $O(K * (N^2))$
2. Space complexity: $O(N^2)$

The growth is directly proportional to N^2

However, the size of the population set can easily be changed. Therefore, if we decrease it to $sqrt(N)$, then the complexity can be converted to O(N). The thing to note here is that if we decrease the number of population it will at the end of the day affect our final result. As these populations are randomly chosen. Decreasing the size of population will affect the chances of getting a better final result. The rich population is always the better.

7. Worked example

Let’s consider the test case 1 of the project.

- $N = 4$
- $Capacity\ (C) = 10$
- $weight = \{5,\ 4,\ 6,\ 3\}$
- $satisfaction = \{10,\ 40,\ 30,\ 50\}$

The method consists of five steps:

1. Generating sets of random population
2. Fitness function and selection
3. Crossing over
4. Random mutation
5. Output best fitted population

We will generate sets of random populations in the first step. For the above example it will look something like this.

$\{\{1,\ 0,\ 0,\ 1\},\ \{1,\ 1,\ 0,\ 1\},\ \{0,\ 0,\ 1,\ 0\},\ \{0,\ 1,\ 0,\ 1\}\}$

Note here that 1 => we are taking that item in the knapsack, 0 => we are not taking that item in the knapsack. The size of the population set would be N and each population will consist of N individuals, therefore, the complexity N^2.

Now we will assign the fitness score as follows:

```
If the final weight of chosen items > knapsack capacity:
        Fitness Score = -1
Else:
        Fitness Score = ∑ Value or Satisfaction points of chosen items
```

$\{\{1,\ 0,\ 0,\ 1\}\{60\},\ \{1,\ 1,\ 0,\ 1\}\{-1\},\ \{0,\ 0,\ 1,\ 0\}\{30\},\ \{0,\ 1,\ 0,\ 1\}\{90\}\}$

After that we sort such that the best fitted population will be on top

$\{\{0,\ 1,\ 0,\ 1\}\{90\},\ \{1,\ 0,\ 0,\ 1\}\{60\},\ \{0,\ 0,\ 1,\ 0\}\{30\},\ \{1,\ 1,\ 0,\ 1\}\{-1\}\}$

Now we do crossing over at a randomly chosen point between two adjacent populations:

Let's assume for the first two the random point is 3. Therefore there won't be any changes as the 3rd and 4th index for them is the same. And for the last two the random point is 4 then we will interchange the last index. And we will also calculate the new score and associate it with the respective population.

$\{\{0,\ 1,\ 0,\ 1\}\{90\},\ \{1,\ 0,\ 0,\ 1\}\{60\},\ \{0,\ 0,\ 1,\ 1\}\{80\},\ \{1,\ 1,\ 0,\ 0\}\{50\}\}$

Now finally we would introduce random mutations. A mutation here means that we will convert the value of a randomly chosen index i from Ai to $\neg Ai$. Meaning if it is 0 then we will make it 1 and vice versa.

As we can not just generate mutations in the whole data I decided to introduce mutations in the 10% of N. Therefore in this example we won't have any mutations as $floor(10/100 * 4)\ =\ 0$

Now we will again calculate the fitness score and then output the best result. For this example it would be $\{0,\ 1,\ 0,\ 1\}\{90\}$. Note here that 90 is also the maximum satisfaction (optimal answer) we can get (calculated using dynamic programming).

8. Summary and Conclusion

Although using genetic problems for solving 0/1 knapsack has its own benefits. It overcomes some limitations of dynamic programming and it is any day better than the recursive approach. However, I found some very easy techniques by which this whole algorithm can be tricked. Just consider the case when the knapsack capacity is infinite. I.e. we can just collect all the items and maximize the satisfaction. But here in genetic algorithms getting a desired population which looks like 1, 1, 1, 1, ... 1 is highly unlikely. But it is interesting to note here that however it looks intuitive, but even the dynamic programming approach can not be used here, as dynamic programming directly depends on the knapsack capacity. Recall the complexities $O(N * W)$. This is one of the key points of my approach that it does not depend on the knapsack capacity.

Another key point is that we can make a trade off between the size of population set and number of iterations. If we decrease the size of the population set it will affect the final output and will make it worse. But don't that will decrease the time and memory complexity of the algorithm. To ensure good results now we can increase the iterations. Now increasing iteration will just affect the time complexity not the memory complexity. However, this is not advisable to do that, as increasing iterations does not really guarantee producing good results however increasing population does. But still if given bound on memory constraint we may consider doing this.

For the above discussed example where N = 4, we saw that getting an optimal answer was very easy and most of the time even a single iteration can output the optimal answer.

For large values of N = 20, 100 iterations give satisfactory results. However, 1000 iterations most of the time will give the optimal result. Until unless you are very unlucky. And 1000 iterations take around 1 sec to execute.

For N = 200, 1000 iterations is recommended to get satisfactory results. And finding the optimal answer is not really feasible. However, increasing the iteration will surely get us closer to the optimal answer. A hard time constraint like in an hour we will be able to make 20,000 iterations and it will most times give the optimal answer.

References

1. 0-1 Knapsack Problem | DP-10
2. Solving Knapsack Problem
3. Local Search and Optimization