Analysing methods of solving NP-complete problems

How do Brute-Force, Dynamically Programmed, and Machine Learning algorithms compare in solving the 0/1 Knapsack Problem, based on speed and accuracy?

Computer Science

Word Count: 3309

Table of Contents

Introduction

The Knapsack problem is a frequently studied combinational optimisation problem based on the idea that someone is trying to fill a knapsack of limited carrying capacity with valued and weighted items to maximise the stored value in the knapsack. The variant of this problem being investigated in this essay is the 0-1 knapsack problem, proven NP-complete around 1980, in which only one of each item can be placed into the knapsack, and said items are indivisible.

An example of this could be an explorer who has found some treasures in a temple. Each item has both a value and a weight, and the explorer wishes to maximise the value they bring back, however their bag can only carry up to a certain weight. Here, they must calculate the optimal combination of items which the bag can handle yet carries the most value.

Similar issues present themselves in the real world often, thus making algorithms which handle them highly relevant. They are important in a wide range of fields, involved in "financial modelling, production and inventory management systems, stratified sampling, design of queuing network models in manufacturing, and control of traffic overload in telecommunication systems" (K Badiru, 2009, p. 2). Knapsack cryptosystems may become increasingly important in the future as a more reliable post-quantum cryptographic system compared to currently dominant forms of encryption (R Diaz et al, 2021).

This study will compare a variety of algorithms on the issues of time complexity and accuracy in their theoretical basis, and experiment on them in a variety of circumstances. The algorithms being studied include dynamically programmed solutions, as well as one

approximation algorithm, the genetic algorithm, as well as a baseline poor performing algorithm using simple brute-force.

Such an investigation is important as it will provide an insight into the practicality of Knapsack Problem algorithms when dealing with a simpler, single-dimensional form of the problem, analysing the trade-offs involved in the different algorithm. The nature of np-complete problems may sometimes necessitate heuristics as alternatives to perfectly accurate algorithms.

Algorithms – Theory

Focus of Comparison

Arguably the most important aspects of any algorithmic solution to this problem are speed and accuracy. Algorithmic speed, otherwise known as time complexity, is most commonly measured through big O notation, which makes a function O(n) = T where n is the input size in bits, and T is the time taken to finish. This takes the worst scaling aspect of the function to classify an algorithm's time complexity, typically focusing on the worst-case scenario, yet other cases can and will be studied alongside that in this study.

Time Complexity is highly relevant to the Knapsack Problem as it is considered NP-Complete. The P here stands for Polynomial, meaning a time complexity that scales to a polynomial function, and the N stands for non-deterministic, meaning that this kind of algorithm can be solved by an infinite number of Turing machines in P time. Any algorithm that can solve a problem in P is considered fast, however it is not known whether P=NP, or in other words, whether all NP problems can be completed in P time. "Complete" refers to the idea that

every NP-Complete problem can be used to simulate all other tasks in NP. As there is no known fast algorithm to solve the Knapsack problem, all non-heuristic algorithms analysed here will fall outside of the polynomial time complexity class.

Accuracy is more self-explanatory, referring to whether the algorithm will always provide a correct answer, and shall be analysed more closely in the genetic algorithm section where perfect accuracy is lost to gain a faster and better scaling approximation algorithm in return.

Time and accuracy, when relevant, shall be the two main aspects of algorithms this essay shall focus on. In particular, the experimental section shall focus more on time usage in an average case scenario, instead of the typical worst-case scenario for time complexity.

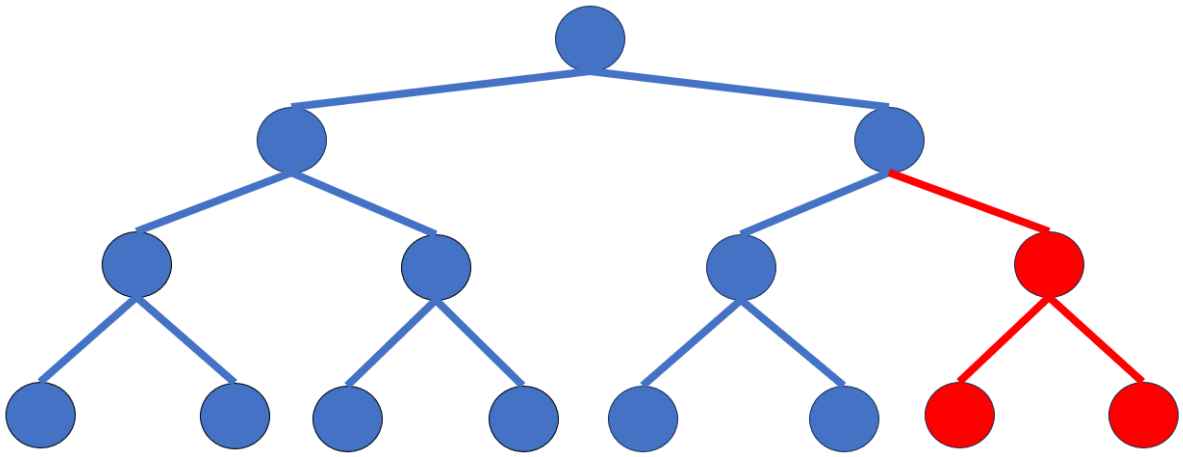
Brute Force

The brute force approach to the knapsack algorithm is one of the simplest yet is not commonly used due to its poor scaling. It finds the highest possible value by looking through each possible of combination of inputs, returning the highest found total value.

Like all algorithms studied here, it views the issue as a series of decision problems, each decision being whether an item should be added to the knapsack. This can be visualised through a decision tree like so:

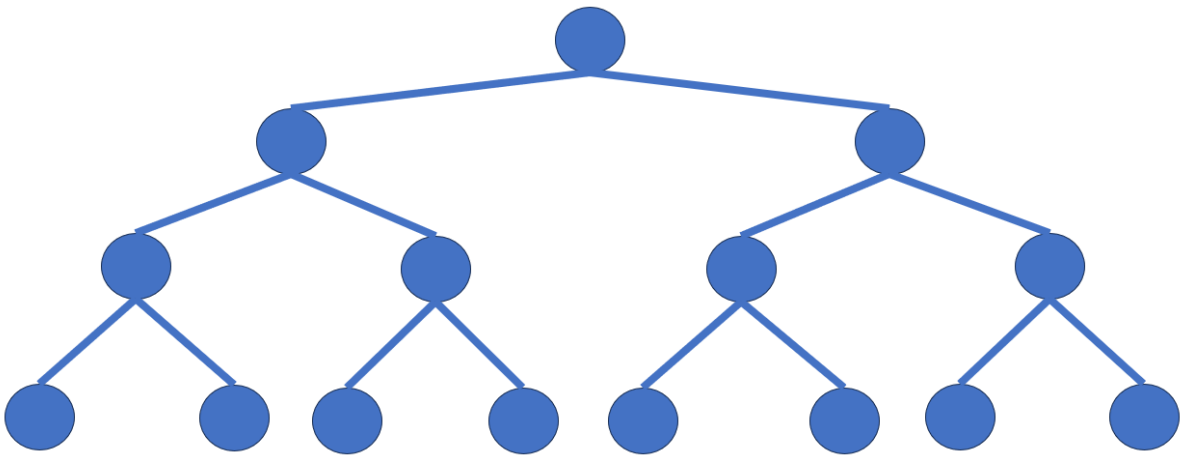
figure 1 – a decision tree showing the combinations a brute force knapsack algorithm may solve. In this instance, the tree-coloured red is pruned off, as the weight is too high in that circumstance to add item number 2, and thus all combinations with both item 1 and 2 can be disregarded. Each node represents a point a decision must be made, other than the bottom

nodes which represent the end of the tree.



In a worst-case scenario, all items are able to fit into the knapsack, and therefore all paths must be completely followed through. As can be seen in figure 2, this leads to the number of paths (each path representing one combination to be tested) increasing exponentially, multiplying by 2 for each new item. This gives the brute force algorithm a very poorly scaling time complexity of $O(2^N)$, with N being the number of items to be put into the knapsack. (GeeksForGeeks, 2023)

figure 2 – Decision tree of knapsack problem where all items in a list of three can fit into the knapsack, with each layer of decisions doubling per item.



While most cases will not be as bad, as certain paths within the decision tree will be pruned off when the weight limit is reached early (seen in figure 1), this algorithm will still typically scale poorly. This shall be analysed further in the testing phase of this essay.

Dynamic Programming

Dynamic programming is “a computer programming technique where an algorithmic problem is first broken down into sub-problems, the results are saved, and then the sub-problems are optimized to find the overall solution” A dynamic programming solution requires two features that occur in the knapsack Problem, overlapping subproblems and optimal substructure. These are some of the most common algorithms used when solving this problem. (C BasuMallick, 2022).

One flaw in the previous algorithm is that certain states may be repeated, meaning that their respective calculations will be repeated. This can be highly inefficient, especially when dealing with large numbers of items where many paths of the decision tree may be repeated. To deal with this, this approach stores the states of each individual decision made, allowing the algorithm to reuse previously calculated answers if they come up again.

Here, every possible moment in the decision tree can be stored in an array for the weight remaining in the knapsack, and index of the item. Thus, since all moments being calculated in the algorithm are stored within this array, with one calculation taking a constant amount of time, the time complexity must be O(NW), with N representing the number of items being inputted, and W representing the maximum weight of the knapsack. (GeeksForGeeks, 2023)

Initially, this may seem to be a fast solution to the knapsack problem, as the number of operations scales linearly to the number of items or the numeric maximum weight of the knapsack. This, however, is misleading, as input size is counted in bits for time complexity. While doubling the number of items will only double the calculation time, doubling the size of the binary input of Weight leads to much poorer scaling. The amount of bits needed to represent an integer value is $\log_2 x$ (rounded down) +1, with x representing the integer. Thus, using the real bit-based input size, this leaves the algorithm with $O(N*2^w)$, with w representing the number of bits used to represent the weight of the knapsack. Thus, while this algorithm scales linearly based on the number of items, it scales quadratically based on the size of the input of the weight, leaving it with a pseudo-polynomial time complexity.

This solution can be programmed in two main variants, a top-down and bottom-down structure. Both types shall be analysed in this essay.

Memoisation – Top-Down

In the top-down method, the algorithm starts by checking the decision at the top of the tree, and then going down each path recursively, one by one. It stores these values in a table, checking at each stage in the decision tree if a corresponding table position has already been filled, thus preventing the algorithm from repeating already solved problems.

While this type of solution is typically more intuitive to implement, this can lead to recursive function nesting which slows down the algorithm, as well as requiring expensive return functions. This recursion can be a limiting factor in this case as code shall be written in

python 3, which has a max recursion limit of 1024, preventing such an algorithm from exploring a deeper knapsack problem.

There is potential for multithreading as different paths of the tree can be run in unison, as the order of storing does not matter when filling in an array in this manner (Aryan Jain, 2022, *runtime*). This shall not be explored further however, as this essay is focusing on solving the problem with one thread.

Tabulation – Bottom-Up

The tabulation solution works in a similar way to memoisation, except it works by building up a table of stored values, using previous values to work through the problem.

This, while sharing the same time complexity with memoisation, is usually considered faster for problems in which all possible answers must be tested (such as in the knapsack problem) (Aryan Jain, 2022). Building from the bottom up prevents recursion by simply referring to previous answers in the table instead of calling a nested function.

Genetic Algorithms

Genetic algorithms represent an alternative approach towards the knapsack problem. Often when no polynomial time algorithm can be found, it may be useful to utilise an approximation algorithm instead, which can guess the answer in a much faster time. This type of approximation algorithm is defined by a genetic representation of a solution that has a measurable fitness, the maximum value of the chromosome. In this case, whether to include an item can be represented as a binary gene, part of one potential solution or

chromosome. Other approximation algorithms exist for the knapsack problem such as the greedy algorithm, but these will not be analysed here.

A simple genetic algorithm functions by pitting various solutions against each other and comparing their fitness. Fitter chromosomes will be able to pass on their superior genes to the next generation of chromosomes, which slowly evolve to converge upon better and better solutions, based off the Darwinian evolution of the natural world.

Genetic algorithms can be structured in a variety of ways. In this case, multiple methods are used to generate fitter generations. All of these emulate real-world evolution in some way.

- Crossover – Crossover functions by selecting two fit chromosomes from the population, labelled the parent chromosomes, which are then mixed to generate two new child chromosomes, possessing genetic material of both parents. The type of crossover being used here is one-point crossover, the simplest, in which the first parent chromosome is copied up to a randomly selected point, at which the remaining part is copied from the second chromosome.
- Tournament Selection – To select the parents, one method to do this is tournament selection. This randomly picks two chromosomes from the population and returns the fittest. This allows genetic variance by picking from the whole population, while still favouring fitter individuals.
- Mutation – Mutation works by randomly swapping genes in a chromosome. This allows a more diverse population of chromosomes, allowing fitter chromosomes to be created and more options to be explored. Typically, a very low mutation rate is used, as in real genetics, to still allow strong genes to pass on and lineages to form.

- Reproduction – Reproduction is the simplest method, simply taking some chromosomes and directly replicating them into the next generation. Here, it is utilised to directly carry over the best chromosome of the last generation, to prevent the most efficient found solution from randomly dying out due to unlucky mutations, as well as to reinforce the fittest gene population.

One of the drawbacks of a genetic algorithm is that it is often highly difficult to pinpoint the precise parameters and structure that will be most efficient for approximating a solution, and these will vary wildly based on the inputted information and desired outcome (e.g., the level of accuracy). This contrasts with the algorithms shown above, which can be more easily recreated and tweaked to suit the needs of the programmer. This will have a negative impact on the efficiency of some genetic algorithms in practice, however it could be argued that this means genetic algorithms have a greater potential for efficiency gains.

The big O notation of this algorithm is more difficult to calculate and will vary based on how the algorithm is made. In this case, time complexity should be based on the number of generations G, multiplied by the population size P, multiplied by the number of genes (based on number of items) N, multiplied by the time taken to mutate a single gene, O(1). This is because mutation is the costliest function in the algorithm, in which each gene has a chance to mutate, occurring once to every chromosome per generation. This leaves us with O(GPN), leaving a theoretically polynomial time complexity, as increasing one variable will expand the time taken by increasing another variable. One benefit of this is that the algorithm may be configured to spend only a given general amount of time, at the cost of accuracy. (Arpit Bhayani, 2022, *run-time complexity*)

Typically, genetic algorithms are used more often in more complex problems, such as with multiple dimensions where other factors such as Volume may be a limiting factor, in which more traditional solutions take more time. The comparison below shall explore the potential for this method in a simpler setup. (Arpit Bhayani, 2022, *Multi-dimensional optimisation problems*)

Methodology

Using Python 3, I created all the algorithms to be tested. I followed some guidance from the internet but put everything together myself. These are the brute-force, dynamic programming (Memoisation and Tabulation), and genetic approximation algorithms. They shall be assessed in a variety of cases to cover strengths and weaknesses for each algorithm.

In each test, the relevant algorithms will run based on items generated with the python random module. The time taken for the algorithms will be recorded with the datetime module and shall be shown in a graph created with the matplotlib module, with other relevant statistics shown in their own graphs. Each test will be repeated 400 times unless otherwise specified, with the averages taken, to improve reliability. The item weights and values are to be both randomly selected between 1 and 10 for each item, each problem, providing a simple base for study. All of this is runs using self-made python code created in the IDLE, visible in the appendix.

For the genetic algorithm, certain parameters were selected. Population size was set at 50 to allow a relatively fast process, as well as to prevent fit chromosomes from being drowned out by chromosomes that break the weight limit, which tended to happen with the current

set-up on higher population sizes. Mutation rate was set at 0.01 per chromosome, for reasons described above. The number of generations was set at 150, which seemed to be a good middle ground by which the algorithm has tended to achieve convergence.

Test 1 – item counts (25, 50, 75, 100), low max weight (10), all algorithms

This test is intended to view the algorithms in a simple setting and analyse the effect of increasing item count, as well as for comparison to other tests.

Test 2 – same item counts as before (25, 50, 75, 100), medium max weight (100), no brute-force algorithm

This test is intended to contrast with the previous one, by seeing the difference in change in max weight. The brute-force algorithm shall not be included here for time reasons, as it takes much longer to complete when there are fewer branches of the decision trees that can be pruned due to exceeding the weight limit (thus closer to a worst-case scenario).

Test 3 – high item count (20,000), high max weight (4000), no brute-force or memoisation, 10 repetitions

This test exists to highlight the effects of a higher item count, and provide an insight into the effectiveness of directly solving problems vs approximating them when handling large quantities of data in this simple form of the knapsack problem. It is a fairer test for the genetic algorithm as approximation algorithms are typically intended for dealing with problems so large that they would otherwise require too much time to solve

Fewer repetitions are used since the algorithms take a lot longer at this scale, and the sheer number of items helps to make each individual case closer to the average case. Furthermore space limitations play a role with the limited hardware available.

Results and Analysis

Test 1

Figure 3: Time taken to complete the knapsack problem with given item size in test 1

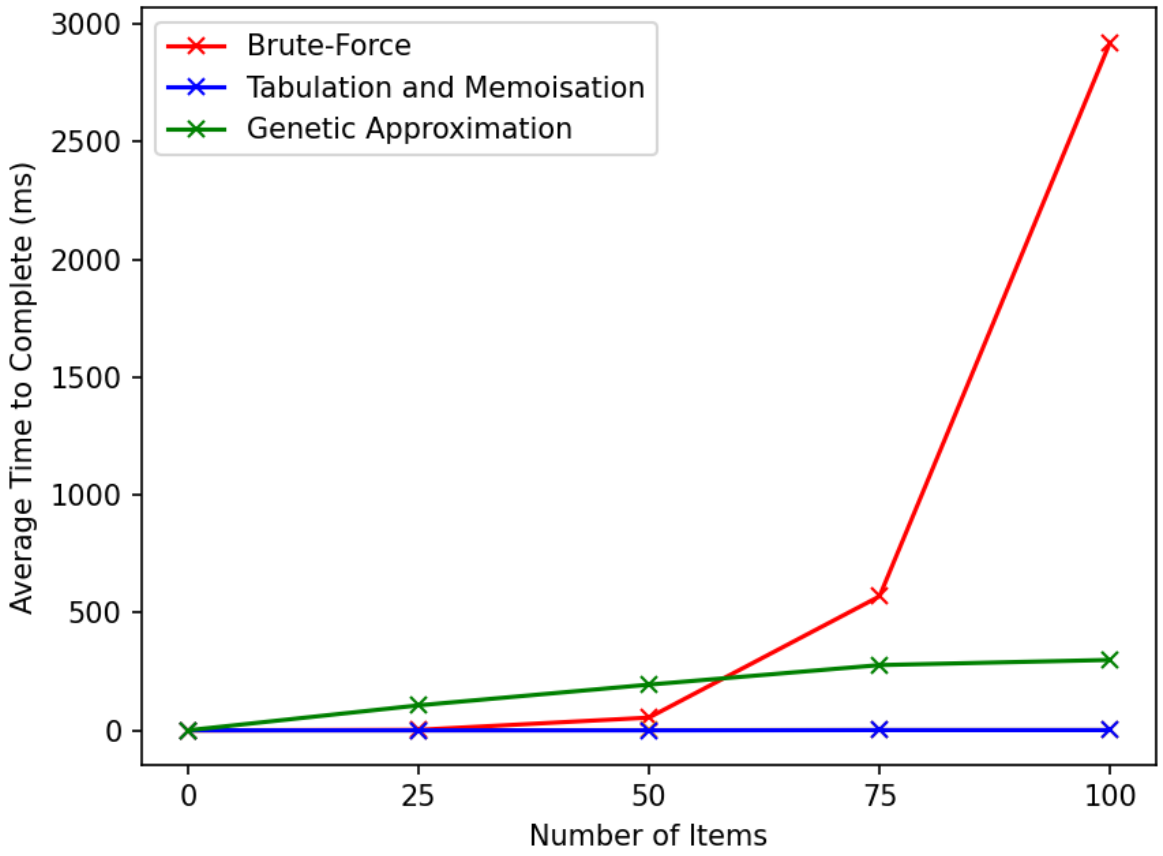


Table 1: Times taken (ms) to complete the problem with given item size for each algorithm in test 1

	25	50	75	100

Brute-Force	2	54	569	2921
Memoisation	0	1	1	1
Tabulation	0	0	1	1
Genetic	106	194	277	299

Table 2: Genetic algorithm accuracy in test 1

Item Number	25	50	75	100
Accuracy	96	72	44	40

Here the exponential nature of the brute-force approach can be seen easily. Each time the number of items increases, the time taken to find the correct solution increases by an order of magnitude. When analysing the performance of the brute-force method on individual problems, it seemed to solve them relatively quickly in some cases, and take much longer on others. For example, in the 398th problem, it took around 12 seconds to solve the problem, whereas in the very next problem, it took less than 0.3 seconds. This attests to the idea that the brute-force algorithm is unreliable in terms of speed, a result of some combinations of item weights necessitating an exponentially larger decision tree to check.

On the other hand, the alternative accurate algorithms take a negligible amount of time to solve each problem, always less than a millisecond (with the tabulation algorithm slightly faster in each case). The genetic algorithm is obviously slower than these accurate algorithms as well at this small a level, however it would typically not be used for such small problems anyway.

It is also worth noting that the genetic algorithm started to poorly perform accuracy-wise once the item number increased beyond the population size and scaled poorly throughout.

It is possible this is partly due to a mismanagement of settings, and finding a better combination of settings for each case may improve the accuracy drastically.

Test 2

Figure 4: Time taken to complete the knapsack problem with given item size in test 2

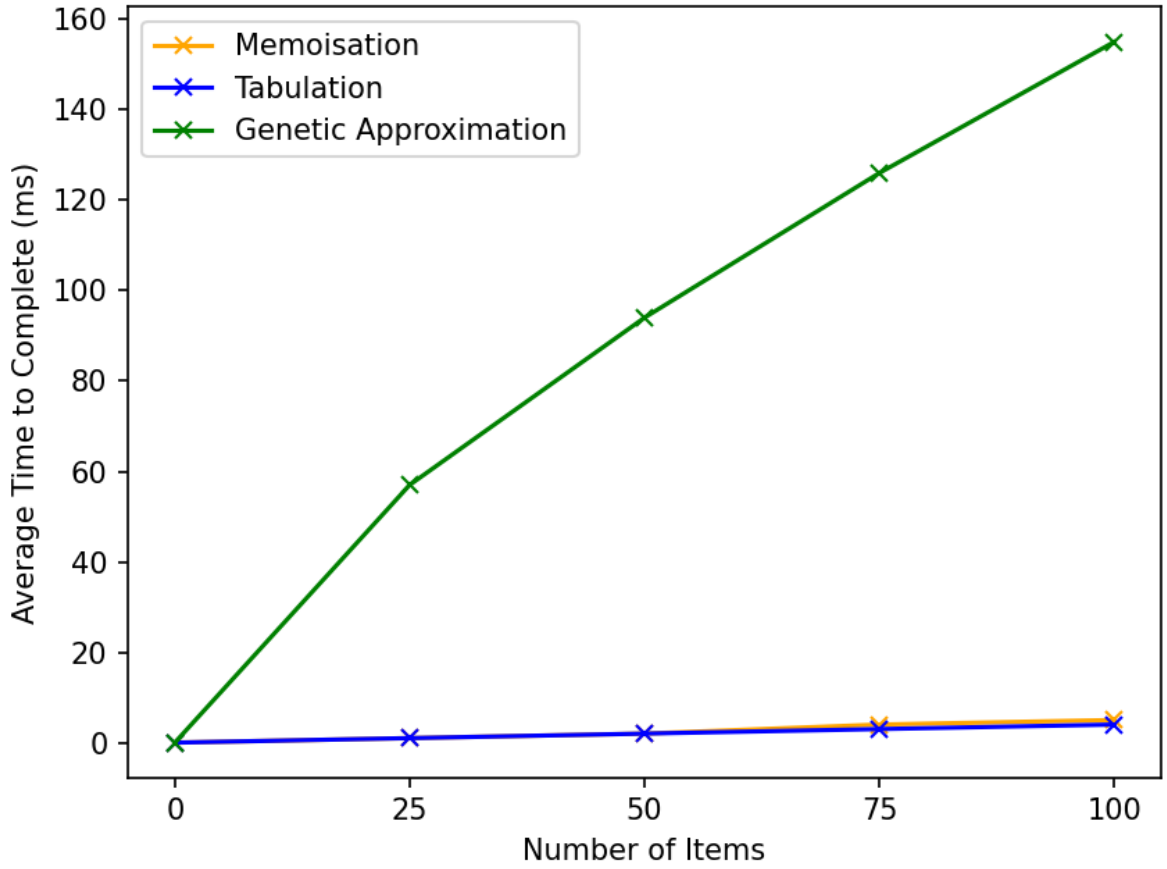


Table 3: Times taken (ms) to complete the problem with given item size for each algorithm

	25	50	75	100
Memoisation	0.91	2	4	5
Tabulation	1	2	3	4
Genetic	57	94	126	155

Table 4: Genetic algorithm accuracy in test 2

Item Number	25	50	75	100

Accuracy	97	94	92	84

In such a small problem both dynamically programmed solutions were yet again fast, however when compared to the previous speeds they were about 5 times slower. Furthermore, the algorithms scaled worse faster than when the total knapsack weight was ten times lower. This is probably due to a higher number of potential combinations to explore, due to more items being able to fit into the knapsack.

Contrasting this, the genetic algorithm performed better than previously, both in terms of speed and accuracy. This difference is due to the genetic algorithm not scaling based on the number of potential combinations, but other factors, showing a key advantage for this type of algorithm in these types of situations. All genetic algorithms so far have suffered in accuracy as the complexity of the problem increases, yet the settings remain the same. This is partly due to the likelihood of randomly finding the correct of very good combinations being high when the total number of possible combinations is low.

Test 3

Table 4: Result comparison for Test 3

	Tabulation	Genetic Approximation
Time Taken (ms)	35727	35026
Average Accuracy (%)	100	18.4

In the final test, the tabulation algorithm has scaled poorly compared to its previous results. Compared to its last result in the 2nd test, it performed about 8930 times worse (35727/4), when the inputting parameters were 8000 times bigger in raw decimal input (40 (knapsack weight capacity) x 200 (number of items)), or 1200 (6 (number of bits to store the number

40) x 200). This means that the tabulation algorithm has not scaled linearly, as can be assumed from its time complexity.

In contrast, whereas once the genetic algorithm was much slower, at this scale, the genetic algorithm is about as fast, and it is reasonable to assume it would go on to drastically beat the tabulation algorithm in terms of speed at even greater scales.

The effectiveness of the genetic algorithm here however is stopped by its poor accuracy. 18% is typically far too low to be used as an effective approximation algorithm. This could be combatted by better fitting the settings of the algorithm to suit this larger task, at the cost of speed, or by restructuring certain aspects of the program.

Conclusion

With this collection of information, the following can be deduced about the practical uses of the 4 analysed algorithms.

The brute-force algorithm scales very poorly and is outmatched by a dynamic programming solution in most cases. It therefore holds little use in solving even a simple 0-1 knapsack problem, when there are better alternatives.

The two dynamic programming approaches are a much faster accurate approach to the problem, with seemingly similar speeds, but with tabulation beating memoisation out by a small amount. Still, the dynamic approach is still not a polynomial time complexity solution, and thus will struggle when the scale of input increases beyond a certain point.

Genetic approximation algorithms hold an advantage over their more accurate competition in this simple form of the knapsack problem, as even this imperfectly optimised algorithm was able to match the speed of the tabulation algorithm at a certain scale. That being said, more time would have to be spent to improve the accuracy before it can be used as a viable algorithm for the large-scale problems it may serve a purpose for.

Ultimately the most self-evident lesson from this is that the best approach to the 0-1 knapsack problem will differ relating to how it is constructed. While no paper can cover all these circumstances, this paper may assist in tackling examples of this problem in the real world.

References

Badiru, KB. (2009). University of Oklahoma, *KNAPSACK PROBLEMS; METHODS, MODELS AND APPLICATIONS*

https://shareok.org/bitstream/handle/11244/319274/Badiru_ou_0169D_10210.pdf?sequence=1&isAllowed=y

BasuMallick, BC. (2022). *Spiceworks, What is Dynamic Programming? Working, Algorithms, and Examples*

https://www.spiceworks.com/tech/devops/articles/what-is-dynamic-programming/

Bhayani, AB. (2022). *Genetic Algorithm to solve the Knapsack Problem*

https://arpit.substack.com/p/genetic-algorithm-to-solve-the-knapsack

Díaz, R.D., Hernández-Álvarez, L., Encinas, L.H., Queiruga-Dios, A. (2021). *Chor-Rivest Knapsack Cryptosystem in a Post-quantum World.* In: Daimi, K., Arabnia, H.R., Deligiannidis, L., Hwang, MS., Tinetti, F.G. (eds) Advances in Security, Networks, and Internet of Things.

Transactions on Computational Science and Computational Intelligence. Springer, Cham.

https://doi.org/10.1007/978-3-030-71017-0_6

GeeksForGeeks. (2023). *0/1 Knapsack Problem*

https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/

Jain, AJ. (2022). *Memoization vs Tabulation in DP*

https://medium.com/@aryan.jain19/memoization-vs-tabulation-in-dp-4ff137da8044

Appendix

Brute-force

```
import random
from datetime import datetime


def bKnapsackSolver(itemWeights, itemValues, knapsackWeight):
    start = datetime.now()

    knapsackValue = 0
    n = len(itemWeights)-1

    def putIntoKnapsack(itemNum, knapsackValue, weightRemaining):
        #print(itemNum)
        if itemNum > n:#if index is out of range
            return knapsackValue
        itemWeight = itemWeights[itemNum]
        if itemWeight > weightRemaining:#if item weight exceeds weight remaining
            return putIntoKnapsack(itemNum+1, knapsackValue, weightRemaining)#if not putting in


        tempResult1 = putIntoKnapsack(itemNum+1, knapsackValue + itemValues[itemNum], weightRemaining-itemWeight)#if putting in
        tempResult2 = putIntoKnapsack(itemNum+1, knapsackValue, weightRemaining)#if not putting in
        #print(tempResult1, tempResult2)
        return max(tempResult1, tempResult2)



    maxValue = putIntoKnapsack(0, knapsackValue, knapsackWeight)
    end = datetime.now()
    timeDifference = (end - start).total_seconds() * 10**3

    return maxValue, timeDifference
```

Memoisation

```
import random
from datetime import datetime

def mKnapsackSolver(itemWeights, itemValues, knapsackWeight):
    start = datetime.now()

    knapsackValue = 0
    n = len(itemWeights)

    storedValues = [[-1 for i in range(knapsackWeight+1)] for j in range(n)]

    def putIntoKnapsack(itemNum, knapsackValue, weightRemaining):
        if itemNum == n:#if index is out of range
            return knapsackValue

        if storedValues[itemNum][weightRemaining] != -1:
            return storedValues[itemNum][weightRemaining]

        itemWeight = itemWeights[itemNum]
        if itemWeight > weightRemaining:#if item weight exceeds weight remaining
            storedValues[itemNum][weightRemaining] = putIntoKnapsack(itemNum+1, knapsackValue, weightRemaining)#if not putting in
            return storedValues[itemNum][weightRemaining]

        tempResult1 = putIntoKnapsack(itemNum+1, knapsackValue, weightRemaining-itemWeight) + itemValues[itemNum]#if putting in


        tempResult2 = putIntoKnapsack(itemNum+1, knapsackValue, weightRemaining)#if not putting in

        storedValues[itemNum][weightRemaining] = max(tempResult1, tempResult2)
        return storedValues[itemNum][weightRemaining]


    maxValue = putIntoKnapsack(0, knapsackValue, knapsackWeight)
    end = datetime.now()
    timeDifference = (end - start).total_seconds() * 10**3

    return maxValue, timeDifference
```

Tabulation

```
import random
from datetime import datetime

def tKnapsackSolver(itemWeights, itemValues, knapsackWeight):
    start = datetime.now()

    knapsackValue = 0
    n = len(itemWeights)

    storedValues = [[0 for i in range(knapsackWeight+1)] for j in range(n+1)]

    for item in range(0,n+1):
        for weight in range(0, knapsackWeight+1):
            if item == 0 or weight == 0:
                storedValues[item][weight] = 0

            elif itemWeights[item-1] <= weight:
                storedValues[item][weight] = max(itemValues[item-1] + storedValues[item-1][weight-itemWeights[item-1]],storedValues[item-1][weight])
                #heighest value of either previous entry in table or if adding item
            else:
                storedValues[item][weight] = storedValues[item-1][weight]

    maxValue = storedValues[-1][-1]

    end = datetime.now()
    timeDifference = (end - start).total_seconds() * 10**3
    return maxValue, timeDifference, storedValues
```

Genetic

```
import random
from datetime import datetime
from numpy import argsort


def gKnapsackSolver(itemWeights, itemValues, knapsackWeight, populationSize, generations, mutationChance):
    start = datetime.now()

    knapsackValue = 0
    n = len(itemWeights)
    highestValues = []
    population = []

    def populationGenerate(population, size):
        population = []
        for i in range(size):
            genes = [0,1]
            chromosome = []
            for j in range(n):
                chromosome.append(random.choice(genes))
            population.append(chromosome)
        #print("Population Created")
        #print(population)
        return population

    def fitChecker(chromosome):
        #print("checking fitness")
        weight = 0
        value = 0
        for i in range(n-1):
            if chromosome[i] == 1:
                weight += itemWeights[i]
                value += itemValues[i]
        if weight <= knapsackWeight:
            return value
        return 0



     def selector(population):#tournament selection
         chosenIndexes = []

         while len(chosenIndexes) < 2:
             randomNum = random.randint(0, populationSize-1)
             if randomNum in chosenIndexes:
                 continue
             else:
                 chosenIndexes.append(randomNum)

         chromosome1 = population[chosenIndexes[0]]
         chromosome2 = population[chosenIndexes[1]]

         if fitChecker(chromosome1) < fitChecker(chromosome2):
             return chromosome2
         else:
             return chromosome1

     def crossover(parent1, parent2):
         #print("mixing genes")
         crossoverPoint = random.randint(0, n-1)
         child1 = parent1[0:crossoverPoint] + parent2[crossoverPoint:]
         child2 = parent2[0:crossoverPoint] + parent1[crossoverPoint:]

         return child1, child2

     def mutate(chromosome):
         for i in range(n):
             if random.random() < mutationChance:
                 #print("mutating")
                 if chromosome[i] == 1:
                     chromosome[i] = 0
                 else:
                     chromosome[i] = 1
         return chromosome
```

```
def getBestChromosome(population):
    #print("getting best genes")
    fitVals = []
    for chromosome in population:
        fitVals.append(fitChecker(chromosome))

    maxVal = max(fitVals)
    bestChromosome = fitVals.index(maxVal)
    return population[bestChromosome]

def getBestFitness(population):
    #print("getting best genes")
    fitVals = []
    for chromosome in population:
        fitVals.append(fitChecker(chromosome))

    maxVal = max(fitVals)
    return maxVal

def populationBestToWorst(population):
    fitVals = []
    sortedPopulation = []
    for chromosome in population:
        fitVals.append(fitChecker(chromosome))

    indexes = argsort(fitVals).tolist()
    indexes.reverse()

    for i in range(len(population)):
        sortedPopulation.append(population[indexes[i]])

    return sortedPopulation

def smallestItem():
    tempWeights = itemWeights
    weightIndexes = argsort(tempWeights).tolist()
    return weightIndexes[0]
```

```
    population = populationGenerate(population, populationSize)
    bestChromosome = [0]*n
    bestOfBest = [0]*n

    for j in range(generations):
        bestOfBest = bestOfBest.copy()
        newPopulation = []
        #print()
        #print("Generation", j+1)

##        if fitChecker(getBestChromosome(population)) == 0:#to stop infinite no fitnesses
##            population = populationGenerate(population, populationSize-1)
##            basicChromosome = [0]*n
##            basicChromosome[smallestItem()] = 1
##            population.append(basicChromosome)
##            print("No fitness found, new chromosome", basicChromosome)

        parents = []#Crossover
        for i in range(int(populationSize/2)):
            parents.append(selector(population))

        children = []
        for i in range(int(len(parents)/2)):
            if parents[i] == parents[i+1]:
                break

            child1, child2 = crossover(parents[i], parents[i+1])
            children.append(child1)
            children.append(child2)

        for child in children:
            newPopulation.append(child)

        for i in range(populationSize-len(children)):#reproduction
            newPopulation.append(population[i])

        for i in range(populationSize):
            population[i] = mutate(population[i])



        bestChromosome = getBestChromosome(population)
        if fitChecker(bestChromosome) == 0:#to guarantee some level of fitness
            newPopulation = population[:-1]
            basicChromosome = [0]*n
            basicChromosome[smallestItem()] = 1
            newPopulation.append(basicChromosome)
            #print("No fitness found, new chromosome", basicChromosome, "with fitness",fitChecker(basicChromosome))
            bestChromosome = basicChromosome

        #print()
        population = newPopulation[:-1]
        highestFoundValue = fitChecker(bestChromosome)

        if highestFoundValue > fitChecker(bestOfBest):
            #print("found new best", highestFoundValue, "higher than", fitChecker(bestOfBest))
            bestOfBest = bestChromosome

        highestValues.append(fitChecker(bestOfBest))
        #print("Population is:", population)

        #population = populationBestToWorst(population)#USEFUL FOR VISUALISATION, DELETE IN REAL
        population.append(bestOfBest)
        #for i in range(len(population)):#check all fitnesses
        #    print(fitChecker(population[i]))

        #print("Best chromosome was", bestChromosome)
        #print("Highest fitness was", highestFoundValue)
        #print("BestOfBest is", fitChecker(bestOfBest))

    end = datetime.now()
    timeDifference = (end - start).total_seconds() * 10**3
    return bestChromosome, highestValues, timeDifference
```