

RESEARCH PROJECT SUBMISSION

PROJECT NAME

**Program: CESS**

***Course Code: CSE 224***

***Course Name: design and analysis of algorithms***

***Examination Committee***

**Dr. Gamal A. Ebrahim**

**Dr. Mohamed A. Taher**

**Dr. Mahmoud I. Khalil**

**Ain Shams University**

**Faculty of Engineering**

**Spring 2020 Semester**

**Student Personal Information**

|  |  |
| --- | --- |
| **Student ID:** | 17P8042 |
| **Student Full Name:** | Youssef Emad Saad |

**Plagiarism Statement**

I certify that this assignment / report is my own work, based on my personal study and/or research and that I have acknowledged all material and sources used in its preparation, whether they are books, articles, reports, lecture notes, and any other kind of document, electronic or personal communication. I also certify that this assignment / report has not been previously been submitted for assessment for another course. I certify that I have not copied in part or whole or otherwise plagiarized the work of other students and / or persons.

**Signature/Student Name:**

**9/6/2020**

**Date:**

**Youssef Emad Saad**

**Submission Contents**

**01: Introduction**

**02: Optimization Approaches**

**03: Results in comparison**

**04: Brute force, greedy, divide and conquer algorithms**

**05: Time complexities**

Optimization of NP-Problems

Youssef Emad Saad 17P8042

CESS

Faculty of Engineering, Ain Shams University *(Affiliation)*

Youssefemac@hotmail.com

*Abstract*—The present text explores the well known subject, NP problem optimization. NP-Problem has been considered one of the greatest paradoxes. Until this day, we don’t know a solution for an NP-Problem in an efficient time. Solving an NP-Problem will result in curing cancer for example (as protein folding is an NP-Problem as well). However, it may also decrypt any bank card number. In the following papers, we will explore 4 approaches to optimize solving an NP-Problem, we will explore how each work, their theoretical foundation, when to use each of them, problem domain, limitations, global convergence of the approach to optimal solution, and we are going to compare their results through written codes in Python and C++, and explain how the results occurred. We will also use Greedy algorithm, Brute force algorithm and divide and conquer algorithm to compare the results with the 4 optimization approaches, and we will later on discuss the time complexities (Big O, Big Ω, Big Θ) for all of them.

Keywords—P vs NP, optimization problem, knapsack, dynamic programming, genetic algorithm, backtracking, branch and bound, brute force, divide and conquer, greedy

# Introduction (*Heading 1*)

P vs NP problem is one of the unsolved problems in computer science, its also from the seven Problems made by the Clay Mathematics institute, solving any of these problems has a prize of $1,000,000. The problem was first introduced in 1971 by Stephen Cookin his paper “The complexity of theorem proving procedures”, Now what is P vs NP? First we need to explain what is P and what is NP. The Class P, or just p, is a class that contains a set of problems, these problems have a known solution or algorithm that can solve them in a Polynomial (or efficient) time. It is also possible to verify the solution in a polynomial time. The Class NP, or just NP, is a class that contains a set of problems, where a given solution or answer to this problem can be verified in a polynomial time. So NP problems (until now) cannot be solved in polynomial time, but can only be verified. Both P and NP can be verified in a polynomial time, this is why P is a subset of NP, meaning that any P problem is automatically NP. But the opposite is not true (at least for now), Some problem are NP but not P (like the problem we are going to discuss in this paper). So until now its not proven whether P is equal to NP or not (but its widely believed that it is not equal), if someone is able to prove that any of the NP set problem is equal to P, this is going to prove that any NP problem may have a solution that takes requires polynomial time complexity. Proving such thing would result in a revolution in science and mathematics, as it may cure cancer (protein folding is an NP problem) and it also may result in decrypting bank cards security numbers.

So P class problem contains of decision problems, where they can be solved on a deterministic machine in a polynomial time. NP contains also decision problems, but solutions can only be verified on a deterministic machine. So is P=NP? In a poll made in 2002 61% believed not, in 2012 this percentage increased to 83%, in 2019 the percentage peaked at 99%.

Another set of problems is the NP complete, where any other NP-problem can be reduced to polynomial time and its solution can be verified in polynomial time. So a problem is said to be NP complete if any NP problem can b transferred into it. Or for the sake of simplicity, an NP complete problem is at least as hard as any NP problem.

Another set of problems is the NP-hard problems, NP-hard problem is at least as hard as the hardest problem in NP problems class set, so all NP problems may be reduced into them. If a problem is called NP hard and NP. Its in that case NP-Complete

NP-complete problem examples are Boolean satisfiability problem (or SAT), Knapsack problem, Hamiltonian path problem, Travelling salesman problem, Subset Sum problem and so on.

So how do we deal with NP problems? We use optimization techniques so that instead of brute forcing every single solution, we can just exclude the solutions that we are sure that isn’t going to work. Or minimize the number of possibilities. We can also approximate the solution. Instead of finding the exact correct and accurate solution, we may just find a close solution that may also work. We may use evolutionary approaches (like genetic which we will discuss later) which involves creating a set of solutions and simulate them in an environment with some constraints, and see which solution will perform optimally the best between them. Another strategy is to restrict the inputs structure (for example to planar graphs).

The NP-Problem that we are going to discuss in this paper is Knapsack, we will also use 4 different optimization approaches and try to compare the results of each of them, how each performed and when to use each.

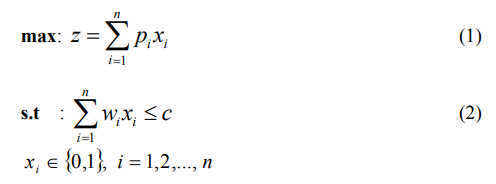
Knapsack is one of the combinational optimization problems, where the solutions is a set of numbers (or with respect to computer science, we may say a set of objects). The combinational optimization is considered to be a subset from the mathematical optimization, where the goal is to find an optimal object from a finite set of objects. (the object itself may contain other objects).

So what is the knapsack problem? We have a knapsack, and a set of items. Each item of these items has 2 attributes. The first attribute is the weight of the item and the second attribute is the value of that particular item. The goal is to put the items in the knapsack such that the value is maximized and the total weight of items did not exceed the weight limit of the knapsack.

There are many variations of the knapsack, there is the fractional knapsack, where you can take a percentage of the items, however this can be solved by greedy method in a polynomial time, so that isn’t an NP problem. There is also the multivalued knapsack where each item may have a different value if stacked with a specific item. Also if the weights are the same as the profit we get the subset sum problem.

In fact, the knapsack can be converted to a family of similar problems in the same tier: Assignment problem, closure problem, constraint satisfaction problem, integer programming, minimum relevant variables in linear system, nurse scheduling problem, traveling salesman problem, vehicle rescheduling problem, weapon assignment problem.

The knapsack mathematical model:



So after exploring what does an NP problem mean, and the knapsack problem itself, lets dive into 4 approaches of dealing with this problem.

# Genetic Approach

## How it works

The genetic approach is a type of algorithm that belongs to the evolutionary algorithms class, where this class is a subset of evolutionary computation which helps in Artificial intelligence. This algorithm is about simulating an environment where there is a population that can mate, mutate, proceed to the next generation, and die. We will explain these operations further but first of all lets discuss the genetic approach in general.

This approach is used to produce a high quality solution for the optimization problems, where the solution is almost perfect, of course its perfection depends on the constraints that we will use in the environment simulation.

So the environment gets created, we have a starting population. This population is a basic population which will later on evolve to the solution we need. So creatures (knapsacks in that case) can crossover (mate) and create a new creature with combined genome. There is also a chance for mutation (using the mutation function) a randomly selected offspring may mutate, creating a new sequence in the genome. Then the fitness function comes to take place. The fitness function determines whether the offspring adapted well to the environment or not. This is the most important part of the algorithm, as it judges the solution and whether its optimal or not, of course it might require many generations to reach this solution.

The genetic algorithm has many applications in artificial intelligence, for example in a video game, if the computer is trying to play a game, its going to try some random approaches and will keep developing them using a fitness function (which is written by the programmer) it will then use the most suitable approach after many generations, crossovers, mutations.

## The Theoretical Foundation

## The theoretical foundation of genetic algorithms were first mathematically explained in 1975. Each gene may have 3 possible values, zero or one or don’t care. Like in nature the fittest members of the population reproduce more. And so the probability of their genes to propagate to the next generation is higher than a member with genes far away from the optimal (does not meet the fitness function requirments). The fitness is evaluated from 0 to 1 (as a percentage) and the more fit an individual is, the higher he will perform and the more probability his genes will exist for further generations. Of course as the generations proceed to exponentially grow, genes that were created at the beginning cease to exist.

## When to use the Genetic Approach

Genetic algorithms are used to solve constrained and unconstrained optimization problems, based on selecting an optimal offspring (or individual). It modifies the population continuously until it reaches the fit members that adapt well with the environment created (fitness function).

However, if the search space is not really constrained tightly, and the fitness function isn’t efficient in that case genetic algorithms may not be the best choice.

In some cases genetic algorithms (especially the fitness function) may get extremely complicated. Like for example designing a gas pipeline, we may need to plug in many equations and many thermodynamics constraints inside the fitness function in order to be able to get the optimal or perfect gas pipeline shape.

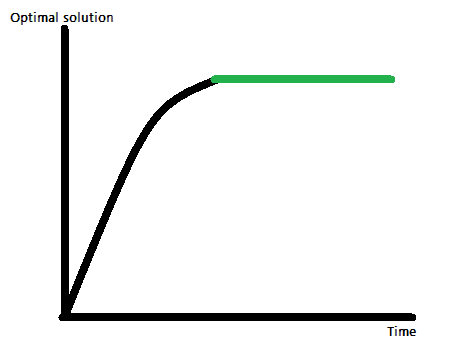
Also genetic is used when we don’t really know a mathematical or clear logical solution. But we know how the solution looks like (basically NP problems). Or even when we know the mathematical solution but its complicated. So its easier to create a simulation and let the solution.

## Global Convergence of the approach to Optimal Solution

The genetic algorithm global convergence is very special compared to the other 3 approaches that we will discuss. The reason for this is because it reaches an approximately optimal solution and not a completely 100% mathematical optimal solution, because this algorithm isn’t a deterministic mathematical equation that solves a problem just like the calculator solves 1+1, it simply doesn’t know how but it knows what the answer should look like, so the shape of the answer is represented in the fitness function. However the more this algorithm is implemented correctly the more close to the optimal the solution is.

Increasing the number of generations also is a factor affecting the global convergence of the genetic algorithm to the optimal solution, such that the more we leave the creatures (knapsacks) in the environment, the more these knapsacks will have to adapt based on their rating of the fitness function (as if they don’t, they will die/ don’t mate).

Implementation of the fitness function accurately and correctly is also a very significant factor because the more constraints you put, the more accurately you will describe the individuals inside the environment. The more individuals that will contain the close to optimal solution. However that still doesn’t mean that it reaches the optimal solution. It tends to reach the optimal solution (may be 99.9999% which is so accurate and acceptable in most cases)



Fig(1)

Fig(1). Describes how genetic algorithm reaches the optimal solution in a logarithmic manner, at the beginning it makes crucial improvements, and then it some slight improvements, afterwards it makes no improvements further and the individuals reach the optimal solution.

## Problem Domain

Applying the genetic approach to the knapsack, we have the population which is represented by knapsacks. Each knapsack has a set of items, such that each item has its own weight and value (normally just like the 01 knapsack). Of course there are no fractions of items. So we start the population with a set of random knapsacks. There is some functions in the genetic algorithm that we will explain with respect to the knapsack problem. Crossover function (mating or breading) which takes 2 knapsacks and create a 3rd knapsack, with a combination of their items (genes). Which was implemented in the Python code attached with this document by taking the first half of the list of the male and the 2nd half of the list of female and combining them together. Of course this list contains zeros and ones. So we have a new individual knapsack.

There is also the mutate function. This function may change the genome of the knapsack randomly by altering a 0 into a 1 or a 1 into a 0 randomly. The main point of this is to count on chance or luck to reach an optimal solution. If the mutation successfully created a more powerful individual or not will be determined by the fitness function

The fitness function has some constraints by which to evaluate each individual knapsack, and give it a rating. A knapsack is better as its value is more. The weight of the knapsack shouldn’t exceed a certain limit. And the more value and less weight the more chance for this individual to survive, because of the fitness function rating that is given for each of the individuals.

So every generation, knapsacks survives according to their fitness in the environment (the constraints), we may at the end look at the fittest member and then we have our answer, or we have a close to the optimal answer. Of course as we increase the generations the answer becomes more and more optimal, but its inefficient at some point due to rate of diminishing returns, where increasing generations at this point doesn’t give us much benefit as it used to be as individuals are almost as fit as the fitness function required them to be.

## Limitations

The limitations for the genetic algorithm however is that the fitness function getting called repeatedly. And in order to find an optimal solution for a complex problem or a complex environment, the fitness function could be very complicated. It could take days to run. And sacrificing the fitness may sacrifice the solution optimality, meaning the more complicated the fitness function is, the more optimal or near fit the solutions or the environment is going to become. Because this only allows a certain kind of creatures (or items) to survive in the environment.

Also if the mutation number is large, the genetic wont scale really well with the complexity of the algorithm due to the increase of the searching space size. This makes it even harder to use this technique generally.

Genetic algorithms don’t generally solve 0/1 decision problems perfectly well as there is no way to converge on solution. So the convergence is slower than other optimization approaches only with respect to the optimization problems.

# Dynamic Programming Approach

## How it works

Dynamic programming is a mathematical optimization method that can also be used in computer programming. Its basically making the algorithm more efficient and less time consuming by storing some of the results of the recursive function calls for example, by doing this we sacrifice the memory in favor of time. This is very useful especially when the algorithm has many repetitive function calls, where the result can be stored so that instead of recalling the function once again, we can make use of the result stored.

We can implement this by doing 3 steps, first of all we can find a recursive solution for solving the given problem. Second of all we can store the solution for the given function call (instead of returning it immediately). We normally in this problem will store it in a 2D array (list of lists in python) such that we can return the result of a repeated sub-problem solution whenever it is called back once again. This process of storing is also called memoization.

So dynamic programming is really suitable for multi-stage / sequential decision tree-process. It is also very efficient for linear and nonlinear problems, or even deterministic problems.

For some problems, Dynamic programming has become the only technique to solve them such as inventory, control theory. Some problems also (averagely) has dynamic programming as the most efficient solution like the problem we are discussing in this paper (although branch and bound in some cases can win over dynamic programming in its best case, sometimes branch and bound may be equal to brute force complexity in its worst case, however we will talk about this later in detail)

## The Theoretical Foundation

Dynamic programming was first founded by Richard Bellman 1950s, in the dynamic programming, we break down a big problem into smaller pieces or smaller sub problems by recursion.

For us to be able to solve a problem using DP or dynamic programming, there are 2 conditions that need to be satisfied, the problem must contain and optimal substructure, a problem is set to have an optimal substructure when we can solve it optimally by breaking it to sub-problems (or smaller problems)

So if those 2 conditions are satisfied, we may look a connection between both the structure of the optimal solution of the original problem, and solution of the sub-problems.

## When to use the Dynamic Programming Approach

Dynamic programming is used when we have a problem such that it can be split into similar sub-problems as we mentioned earlier. So mostly this approach is used in optimization problems, instead of using exhaustive search, we can just save the results of earlier searches so that we wont have to run it again. However in a greedy algorithm, the local optimization is addressed, but dynamic programming is aiming for a total optimization of the main problem. Also in divide and conquer, solutions are mixed together to achieve a solution for the main problem. On the contrary, dynamic programming or dynamic algorithms in general use memoization to save the output of already solved sub problems. Some other problems that are widely popular with Dynamic programming approach are Fibonacci number series, subset sum problem (and other knapsack variations), knapsack problem, tower of Hanoi, Dijkstra’s algorithm (shortest path), project scheduling...etc.

## Global Convergence of the approach to Optimal Solution Dynamic programming creates a complete table (or tree) for all of the possibilites in the knapsack problem. This is why the dynamic programming is capable of reaching an optimal solution accurately, unlike the genetic algorithm which may reach an accurate form of the optimal solution depending on how well written the fitness function and the number of generations there are, however we will compare the results later. But we mentioned this now just to explain that the dynamic programming is basically a faster version of brute force due to its capabiltiy of storing the answers of different sub-problems. However since it checks all of the answers and puts it in a table, the exact result as brute force can be reached and also faster, due to less function calls.

## Problem Domain

So in this problem domain, we want to store each function calls in the form of 2D arrays. We want to maximize the profit in the knapsack, we will use the tabular method (2D array).

We first create a 2D array (it is named memory in my python code) such that the columns is the same number of the knapsack size, and rows = number of items + 1. We then fill the first object’s row, then the second’s + the first row, then the 3rd + the 2nd + the 1st …etc.

We only fill the object when the capacity of the knapsack is equal or more to the object’s weight, so for example if the capacity is 3, we will zero all of the columns until we reach the column number 3 (fourth column).

If the weight isn’t sufficient for all of the objects, we choose only the objects that will provide the max profit and put it in the table, the optimal solution is found on the bottom right box of the table. Why is that?

Because we start filling till we reach maximum value at the end. Which is the last value assigned in the memory.

## Limitations

A) The greatest limitation of the Dynamic programming approach is the number of partial solutions that we need to retain.

B) Dividing the problem into sub-problems and storing every single solution which consumes the memory.

C) There is no way to form a general dynamic programming algorithm which can be generic and be used for all the problems, meaning that we have to write a specific dynamic programming algorithm for every single problem.

D) Unlike branch and bound, it has to compute all of the possibilities, although it stores the results so that it doesn’t have to calculate it again, but of course its more efficient if there is a way to predict that this solution isn’t beneficial or wont lead us to an optimal solution.

# Backtracking Approach

## How it works

The Back tracking approach is used to try to develop an answer incrementally, by rejecting answers that wont satisfy the constraints or the boundaries of the problem. It is generally used to find all solutions first to the problem that builds to the solution. Rejecting the problem is called backtracking which is the name of the algorithm itself, because it “back tracks” when it find that this answer isn’t going to lead to the optimal solution or if it messes up the constraints, which causes it to break out from the recursive call and thus eliminate further trials throughout other permutations of this given answer.

The most common example of the back tracking or if we may say, the most cliché example for the backtracking is the eight queens puzzle problem.

Backtracking generally solves constraints satisfaction problems, like sudoku, crosswords…etc. and it is used to find all of the available solutions to a problem. If a bad decision is taken, back tracking happens.

Backtracking is considered to be one of the best ways to solve problems such as knapsack and generally the combinatorial optimization problems.

The backtracking enumerates pieces of objects that may be put together to give a solution for a given problem. This is done step by step incrementally, such that it is represented through nodes of a tree like in the branch and bound that we will discuss later. So the tree is traversed in a recursive manner, from the root. Where this is called depth first order, or DFS (depth first Search). At a given node, we check if it can reach a completed solution. If it can’t the whole tree is ignored, if yes it may check for all of the sub trees of this node, And so on. So this search is only a part of the tree.

## The Theoretical Foundation

Backtracking name was first made by the mathematician D. H. Lehmer in the 1950s. then it was used in a string processing language in 1962. Where mathematically speaking it is a method to solve equations. We can solve an equation in maths by backtracking by the following steps : we first write the equation, create a flowchart and construct an expression, we use the backtracking to solve for x. after that we copy the equation and simplify it by collecting the like terms together.

So this is how backtracking was created in theory for mathematics before implementing it in computer programming

## When to use the Backtracking Approach

Backtracking may be applied to any problem which has in the core of it the “partial candidate solution” where we can perform a rapid test to see if it can reach an optimal solution.

## Global Convergence of the approach to Optimal Solution

Backtracking is able to reach the optimal solution because as we said earlier. It traverses through all of the solutions possible although instead of exhaustive searching, it backtracks whenever the path is clearly not going to give us an optimal result.

The solution that the backtracking provides is accurate in the knapsack case, as it reaches for every single combinations (unless it backtracks) without exhaustive search. But it finds the correct answer.

## Problem Domain

To solve the knapsack using backtracking, we need to acknowledge first that this problem has 2n  number of possibilities and so this is the number of ways to assign zero or one to a given tree node directions (or truth tables), branching in the left direction represents putting the item in the knapsack, while branching in the other direction represents not putting this item in the knapsack. Backtracking here is realizing by recursion that this solution isn’t suitable, so when we say backtrack, we are going to reject the solution and go backwards (also reject all of the function calls that would be done further) which is very efficient so that we wont need to check all of the item combination in the knapsack.

## Limitations

Some of the limitations of the backtracking is the fact that a given node may be expanded multiple times from other starting nodes (redundancy of data).

Also the fact that its very inefficient if there is a lot of branching.

It also takes a lot of space for the stack of the recursive function calls, where function calls are expensive and require space for each call

One of the limitations is also thrashing, which is failing a couple of times for the same reason.

The last limitation is the fact that it doesn’t detect the fact that this tree wont lead to a desired solution early on, it does so after some function calls which wastes time and consumes the memory.

# Branch And Bound Approach

## How it works

The Branch And Bound approach is an algorithm that can be used for computer optimization problems, also in mathematical optimization, Also for discrete optimization. The branch and bound algorithm basically creates a tree, with the complete full set in the root (all items included in the knapsack) and then it branches the tree and explores it based on including or excluding items in the set. Branches of the tree are considered to be subsets of the solution.

The reason why branch and bound approach is powerful is because it checks the upper and lower of each node and it may kill the node (and not explore further) which is efficient as it eliminates solution that is surely not optimal.

The branch and bound works on estimating the lower and upper bounds of all nodes of the tree. If there are no bounds, the algorithm is basically exhaustive search and its no better than brute force.

Sometimes in the worst case, the branch and bound also becomes brute force, when we search for all bounds without killing nodes from the beginning until we reach the final node.

The branch and bound is basically a minimization algorithm, which tries to find the minimum cost of a given function. So it basically splits the searching into nodes, then try to minimize the function based on these nodes, This is what we call branching. If we were to branch only, this would be brute-force, however this is when the bounding helps to maintain the efficiency of the algorithm.

So what does the bounding do? The bounding makes sure that we aren’t searching in nodes that we can predict they wont provide us with the minimum cost (or in knapsack case maximum profit).

Branching produces 2 other nodes from the parent node, one by which we include and the other exclude a given item, for every node we calculate the u and the c, by bounding, we calculate the lower bound of a value of a node.

So the branch and bound algorithm may be considered as a top-down search in a tree of nodes created by branching. To describe more of the general pseudocode of the algorithm we may provide the following steps:

First we provide an answer x*h* to the problem and keep it. If not we may put it initially as infinite. This will indicate how optimal is any node that we will explore

Second we create a queue to keep the nodes of the tree (unless killed)

Third we loop until the nodes of the queue are empty; we take the node, if this node calculation smaller than our bound then assign it as the new B, but if that’s not the case then we branch to make other nodes, if the bounding function calculates a bigger value than our B then don’t do anything, it might be later discarded as it wont lead to an optimal solution, else we keep it on the queue.

Branch and bound can traverse in 2 possible ways, DFS or BFS (depth first search or breadth first search). It searches all of the state space tree to get to the optimal answer.

## The Theoretical Foundation

Branch and bound was first created in 1960 by Alison Doig and Ailsa Land. It was initially designed for discrete programming, later on it became a common tool to be used for solving NP-problems.

In theory, The goal is to maximize or minimize the solution of a given function f(x), between other candidate solutions S, S is the feasible region, where feasible region in mathematical optimization is the set of all possible values that may satisfy the problem’s constraints, it may be either desired to find the minimum (by default it’s a minimization approach) or find the maximum by saying g(x) = -f(x). so branch and bound first branches the problem into a tree, then it minimizes each node in order to minimize the general function f(x) (or may even maximize it based on the problem). Then it records the bounds of the minimum in a queue to eliminate any path that may not be optimal.

## When to use the Branch and Bound Approach

Branch and bound is used to solve optimization problems that’s goal is to make a 0 1 decision to maximize or minimize a certain function. The branch and bound technique is used to solve many problems such as 0/1 knapsack, 8 puzzle problem, job assignment problem, N Queen problem, travelling salesman problem.

Any minimizing problem (finding the minimum of a certain problem) or even maximum by using the g(x) = -f(x) (maximizing problem) may be optimized using this approach to solve it more efficiently than brute forcing. Its also better than backtracking because backtracking may not exclude the tree that wont contain an optimal solution unlike branch and bound.

Branch and bound is also used in mathematical problems such as discrete mathematics and integer programming.

## Global Convergence of the approach to Optimal Solution

Branch and Bound is capable of reaching an optimal solution, due to trying all of the solutions, although it doesn’t continue to search further if it knows from the beginning if the solution isn’t going anywhere and that the optimal solution is lying somewhere else.

## Problem Domain

In this problem, we want to get the maximum profit inside a knapsack (variable to maximize, or minimize if we revert the signs which was done in the Python code attached with this document), we also don’t want the summation of the items weight to exceed a certain limit (the weight of the knapsack).

So these are the constrains, the question is how can we implement the branch and bound in the knapsack problem domain?

We start by initializing the upper bound by infinity. We then calculate the C and U, the C can be calculated by including fractions of the items, the U however is calculated by not including fractions (both are represented by the included and notincluded functions in the Python code). The U (or upper bound) is calculated by not adding any fractions, later on we update the upper bound if we got a bigger upper bound. Next we branch into nodes, either x=1 if we include the node, or x = 0 if we don’t, we then calculate U and C the same way we did, we check if the U is smaller then update the U, if we update the U we check if any other node has a C bigger than the current U, then kill the node, this is why we don’t have to explore nodes that we know wont provide a solution. Which is why branch and bound is effective in this problem.

## Limitations

The main problem or setback in this approach is the fact that we use trees to branch the nodes or the knapsack items. This causes the algorithm to be very time consuming, such that there may be a lot of nodes in the tree.

Another setback is that the algorithm forms the tree and the branches, and checks for the upper limit, it may take a while until it kills all the nodes that aren’t going to produce the optimal solution. So the problem is creating these nodes takes time and there is no way to solve this problem in that specific approach. Also another problem is the space consumption of these nodes because they aren’t going to produce any optimal solution and they are just consuming the memory.

The third setback is the fact that not only trees grow in a linear way, they grow in an exponential way without really any improvements in the optimal solution that we are looking for.

# Comparison Between Their Results

## Results

Each approach has optimized the problem in a different way, and every approach may be useful depending on the problem constraints. The dynamic programming performs based on the weight of the knapsack and the number of items, the complexity of the dynamic programming approach in the worst case is O(nW) where n is the number of items and W is the weight of the knapsack. The genetic algorithm performs depending on the algorithm used, for the algorithm I wrote, it performed at O(gnm), where G is the generation number, m is the number of population, N is the size of the individual knapsack (number of items inputted in the problem). I also tried to make the generation number equals to 1 to make it faster, which works for smaller knapsacks and I will explain why in part B. branch and bound worst case is same as brute force O(2n). backtracking’s worst case is also same as brute force and branch and bound’s worst case.

Genetic algorithm performed better than Dynamic programming. Both are guaranteed to have a predictable time, unlike branch and bound and backtracking, as we may find the optimal solution as quick as n. rarely the branch and bound can outperforms both the genetic and dynamic, or maybe even in that case, the genetic is close to it. Backtracking is the same as branch and bound as a time complexity, although it performed slightly faster.

## Explanation

Lets first explain the genetic algorithm, the genetic algorithm result was linear, why is that? because let’s discuss how it was executed, the genetic algorithm loops on every chromosome (or bit) for an individual (which is equal to the variable N). it also loops on all the individuals (which is equal to m), and it does that for a number of generations g. so the complexity is O(gnm). But the g and the m can both be set manually depending on how fast the algorithm reaches the optimal solution, so they are constants. So, the complexity of the algorithm depends on the number of items inputted, in a linear manner. We may control the number of generations and the number of population based on how we want to get the optimal solution. The more we increase the number of generations and the number of population the more, but this may not be efficient as we may reach the optimal solution early on and all of the improvements later on are useless. So lets assume after running tests that we identify the perfect number of generations and population size. The genetic algorithm in that case will outperform any other algorithm for the knapsack.

What about the dynamic programming? The dynamic programming in all of the cases is going to create a 2D array which is the number of items on the Y axis and the weight of the knapsack on the X axis, Dynamic Programming will always create this nW size 2D array (or list of lists) without any further optimization. The complexity will always be nW. also a huge setback is the fact that the memory usage is consumed vastly in bigger sizes.

For the branch and bound, backtracking, the worst case is 2n and the reason for this is that it may create a node for all of the possibilities until it reaches the optimal at the end. In the best case however it may propagate the tree and get the answer as quickly as n. but this is less likely to happen as the array size gets bigger and bigger. The biggest problem with the branch and bound is not only the slow time but also the fact that it consumes a lot of space.

In my algorithm for branch and bound, I implemented the tree as a list of objects (nodes) such that every time a node is killed, or a node is branched, its removed for efficiency. This is very useful because as the input size increases exponentially, any small improvements will increase the performance.

# Brute Force

The idea of brute force is to try all of the solutions. This is the 2nd most inefficient way to solve a problem, the most inefficient way to solve a problem is to randomly try solutions, but that’s not the case in brute force, because in brute force we will try solutions iteratively. Brute force may also be called exhaustive search, and its very easy to implement, but very inefficient in terms of time and space complexities.

For the brute force algorithm, there is several ways to implement it, we can use a branch and bound implementation without the bounding function. Where its going to recursively check for all of the possible nodes without any evaluation or efficiency. However, the way I implemented it in the python code attached, is generating a truth table, by creating a for loop, and changing the iterative variable into binary (“0:b”.format(variable)), then checking if this combination forms the maximum profit with weight less than or equal the knapsack weight, by looping through all of the combinations which are 2n.

Now the time complexity will always be 2n, which is worse than the 4 other approaches (but it’s the same as branch and bound, backtracking in their worst case), however in the worst case the only good thing about it compared to branch and bound, backtracking, is the fact that they both use the memory to form a tree. brute force doesn’t store anything, it just loops through different combinations, unlike branch and bound and backtracking. So brute force will win in the worst case ( same time complexity and less memory consumption). However the brute force will never win against the Dynamic programming approach or the genetic algorithm.

# Greedy Algorithm

The greedy algorithm is an algorithm that takes the best decision in the current or local stage. It doesn’t care or take the global or the macro level into its consideration, it only takes the local or the micro level into its consideration.

The solution is created stage by stage, iteratively, until it reaches the end. In the case of optimization problems it doesn’t guarantee an optimal solution, but however guarantees a quick and efficient running time.

Although the greedy algorithm works very well with the fractional knapsack problem, it solves it optimally in an efficient polynomial time, it doesn’t work very well with the 01 knapsack. Because in the 01 knapsack you cant divide items, so you might take items using the greedy procedure and then at the end there is a space in the knapsack that’s empty and you realize that on the global or macro you didn’t take the optimal decisions from the beginning. This isn’t the case in the fractional knapsack, in the fractional knapsack, if we solve it by the greedy algorithm we can create an array or list which is formed by dividing the values of the items over their weights, and then we sort this array or list in a descending order, and we start taking the items, the first item in that array represents the item with the highest value compared to its weight (as its divided by its weight). At the end if we find an item that cant fit the knapsack, we divide it to fit the maximum weight possible and get the value equivalent for that given weight, this solves it optimally.

So if we will apply the greedy approach on the 01 knapsack, we will do the same steps: create a list, divide the item’s value by their weight, sort them in a descending order, and start taking the items, however some space will remain at the end without the optimal usage of the maximum available weight, this will cause the greedy algorithm to be very quick but not provide a correct answer at all. It might provide a correct answer and it might be used for reference, but this isn’t what we call an optimal solution.

Now the greedy performed almost as quick as the genetic (but the genetic was quicker), and the greedy was quicker than the dynamic programming, however the greedy doesn’t provide a correct solution all times. The greedy is always much quicker than branch and bound, and backtracking. But all of the other approaches even genetic provide a more accurate solution than greedy.

Overall, greedy cant be used for solving NP optimization because of its inaccurate answers.

# Divide and Conquer Algorithm

Divide and conquer is an algorithm that works on recursing and branching multiple times. It splits the problem into two sub problems every time a function call occurs, it then solves the sub problems. The sub problems become easier than the main problem. So we solve the sub problem and then we sum the solutions together until we get a solution for the bigger problem. This technique is extremely useful and efficient for some problems like sorting (merge and quicksort). Quicksort is the fastest algorithm generally for sorting.

In this problem we will recursively divide the knapsack subset items, we will solve the problem several times until we get the solution we want, how are we going to do that?

First we divide it by keeping the knapsack total weight as it is, we divide the subsets into smaller subsets, until we reach only 1 item, we decide whether to put it or not. Then we add it up until we get 2 items for example, we take the decision again and keep adding it up until we reach the full number of knapsack.

Now we can implement this in 2 ways. One of them wont provide an accurate solution but will be O(NlogN) concerning the time complexity. So how can we do it? We can do it similar to the merge sort strategy, where we will sort them based on the greedy method. And add them together one by one. This will ensure a very quick time, with some memory consumption, basically exponential but still efficient, however the result isn’t accurate at all.

Another approach is to add the answers together but as we add the answer we check for optimal backpack weight, of course due to divide and conquer this will take less time than if we were to check it as a total knapsack problem, however overall this isn’t efficient at the end because checking twice is still (2n) worst case time complexity.

In the following table, we have all of the algorithms and their best cases, average case and worst case time complexity in one table. Each one of them represents if the optimal solution is put in the test case so that the algorithm detects it quickly or not.

1. Complexities of algorithms

| Algorithm name | Cases | | |
| --- | --- | --- | --- |
| Best Case | Average Case | Worst Case |
| Genetic Algorithm | Ω( gnm ) | Θ( gnm ) | O( gnm ) |
| Dynamic Programming | Ω( nW ) | Θ( nW ) | O( nW ) |
| Backtracking | Ω( n ) | Θ( 2n ) | O( 2n ) |
| Branch and Bound | Ω( n ) | Θ( 2n ) | O( 2n ) |
| Brute Force Algorithm | Ω( 2n ) | Θ( 2n ) | O( 2n ) |
| Greedy Algorithm | Ω( n ) | Θ( n ) | O( n ) |
| Divide And Conquer Algorithm | Ω( 2n ) | Θ( 2n ) | O( 2n ) |

Fig.3. Complexities of all algorithms compared.

So as a conclusion, we can see that dynamic programming will always have the same time complexity, due to creating the 2D array in all cases regardless of any other factors. Just the n number of items and W weight of the knapsack. The backtracking and the branch and bound are most of the time exponential, unless by coincidence it finds the solution optimally and rejects the other solutions quickly. Brute force will always check for all of the combination no matter what. Greedy algorithm will always have the same complexity as it divides the value based on the weight, sorts the results in a descending order, and take the items until the knapsack cant take more items. Divide and conquer will split the problem into sub problems (trees) and evaluate each, which will cause the same complexity in all cases. However the genetic outperformed all of them. Because we can set the population and generation number optimally for the best time efficiency possible, also the fitness function indicates how it will perform.

The genetic complexity is linear. And the solution it provides is very acceptable, even though its not providing it through mathematics, but its providing it through simulation. And writing the algorithm correctly will produce the desired results.

##### References

1. Martello, S. and Toth P.1990. Knapsack Problems, Algorithms and computer implementations. John Willey and Sons.
2. Ameen Shaheen and Azzam Sleit, Comparing between different approaches to solve the 0/1 Knapsack Problem, university of jordan.2016
3. Charles E. Leiseron, Ronald L. Rivest and Clifford Stein, Introduction to Algorithms, 3rd Edition, MIT Press, 1990
4. David pisinger, Algorithms for Knapsack Problems, 1995
5. Peter J. Kolesar, A Branch and Bound Algorithm for the Knapsack Problem, 1967