

RESEARCH PROJECT SUBMISSION

**Program:**

***Course Code: CSE224***

***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:** | 17p6027 |
| **Student Full Name:** | Hla Ahmed Mohamed Mansour |

**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:**

**Hla Ahmed Mohamed**

**Submission Contents**

Contents

[I. Introduction 5](#_Toc42469690)

[A. Optimization algebraic and geometric background 5](#_Toc42469691)

[B. Duality 6](#_Toc42469692)

[1) Properties of duality 7](#_Toc42469693)

[2) Importance of duality 7](#_Toc42469694)

[II. Dynamic programming 7](#_Toc42469695)

[A. Problem domain 8](#_Toc42469696)

[B. Theoretical foundation of dynamic programming 8](#_Toc42469697)

[C. When to use dynamic programming 8](#_Toc42469698)

[1) Applications 9](#_Toc42469699)

[D. How does it work 9](#_Toc42469700)

[1) Trade-offs between iterative and recursive implementation of dynamic programming: 10](#_Toc42469701)

[E. Global convergence 10](#_Toc42469702)

[F. Global convergence of dynamic programming 11](#_Toc42469703)

[G. Limitations 11](#_Toc42469704)

[III. Genetic algorithm 12](#_Toc42469705)

[A. Problem domain 12](#_Toc42469706)

[B. When to use it 12](#_Toc42469707)

[1) Applications 12](#_Toc42469708)

[C. Theoretical foundation 13](#_Toc42469709)

[D. How does it work 13](#_Toc42469710)

[E. Global convergence of genetic algorithm 15](#_Toc42469711)

[F. Limitations 15](#_Toc42469712)

[IV. Simulated annealing 15](#_Toc42469713)

[A. Problem domain 15](#_Toc42469714)

[B. When to use it 16](#_Toc42469715)

[1) Applications 16](#_Toc42469716)

[C. How does it work 16](#_Toc42469717)

[D. Theoretical foundation 17](#_Toc42469718)

[E. Global convergence of simulated annealing 17](#_Toc42469719)

[F. Limitations 18](#_Toc42469720)

[V. Branch and bound 18](#_Toc42469721)

[A. How does branch and bound work? 18](#_Toc42469722)

[B. when do we use branch and bound? 20](#_Toc42469724)

[1) LP Relaxation 21](#_Toc42469725)

[2) Applications of branch and bound 21](#_Toc42469726)

[C. Global convergence of branch and bound 21](#_Toc42469727)

[D. Limitations of branch and bound 22](#_Toc42469728)

[E. Problem domain of branch and bound 22](#_Toc42469729)

[F. Theoretical foundation of branch and bound 23](#_Toc42469730)

[VI. 0-1 Knapsack problem 24](#_Toc42469731)

[A. Testing the different solution methods 25](#_Toc42469732)

[VII. References 27](#_Toc42469737)

List of tables and figures

Fig (1) Branch and bound 19

Table(1) Execution time vs problem size 25

Fig(2) Execution time vs problem size graph 1 26

Table(2) Execution time vs problem size 26

Fig(3) Execution time vs problem size graph 2 27

# Introduction

In this report we are going to talk about four optimization approaches, which are dynamic programming, genetic algorithms, branch and bound and simulated annealing.

We are going to use the previous four optimization approaches to solve 0-1 knapsack problem which is a NP optimization problem, we will compare between the performance of the mentioned approaches and, also, we are going to apply brute force, greedy algorithm and divide and conquer to solve 0-1 knapsack and compare them to the four optimization approaches that we studied in this report.

## Optimization algebraic and geometric background

we need first to explain the idea of finding an optimal solution to a maximization or minimization problem.

As we said before in optimization problems we have inequalities that serve as constraints, these inequalities are called half space.

Half space is one of the parts resulting when a plane splits a 3D Euclidean plan.

Half space is a convex set, convex set is a number of points and it has the following property, if we draw a line segment between any two points then all the points on this line segment belong to the set.

So, our inequalities are convex sets, the intersection between convex sets is also a convex set

This can be proved easily as intersection means that the resultant set exists in all the original sets that we had in the beginning, every convex combination is then in each set, so, the intersection is convex.

the feasible region of the optimization problem is the intersection of all half spaces, the most important parts in this feasible region is the vertices.

Vertices are very important because at least one of the points where the objective value that we are trying to find exists is located at a vertex.

We can prove that geometrically, for example if we have a 3D plane each constraint we have resembles a hyperplane, the intersection of several hyperplane is polytope where every point in a polytope is a convex combination of its vertices.

Our objective function is a hyperplane that intersects the polytope if we want to minimize it we will drag the hyperplane down until it intersects the polytope at one of its vertices and there lies the optimal solution.

So, the geometrical solution is to enumerate the vertices and substitute with each vertex in the objective function and then we see if those points are integer and if not, we approximate them to figure out if the approximation is an optimal solution.

And then we wanted to computerize the solution so, we had to develop algorithmic methods to solve these optimization problems, the first algorithmic solution was naïve and took a long time to find the optimal solution.

selecting basic variables and performing Gaussian elimination, then test whether the solution is feasible and then select BFS with best cost.

We are going to explain the previously stated steps, solving a system of equations means that we express our basic variables in terms of other non-basic variables and then we will assign zeros to all non-basic variables, so, the basic variables will take the value of the constant numbers that were being added to the non-basic variables in the equation on the right-hand side of the problem, in order to know whether a solution is feasible or not we will check the value of the basic variables if they are non-negative then we have a feasible solution.

The last step is to select the basic feasible solution with the best cost, we must take into consideration that all the previously stated steps dealt with equations and we already know that optimization problems have inequalities as constraints so we must at the very beginning convert the inequalities to equations using slack variables and surplus variables.

but that naive strategy is going to take a long time, because we have a large number of basic solutions.

Denote the number of basic solutions as x

(1)

So, that leads us to the importance of the algorithms that will be explained next.

There is an important theory that we will need to understand because we can make use of it in the following explained approaches, this theory is duality.

## Duality

the primal is a minimization problem while the dual is a maximization problem.

So, the primal is moving from a high place down and the dual is moving from a low place up and the thing is the dual is always lower than the primal until they intersect, so, no crossover occurs between the primal and the dual.

If the primal has a feasible solution, then the dual is maximized till it intersects the primal at this feasible solution so it cannot be unbounded.

If the primal is unbounded, then the dual is infeasible and vice versa.

An example on primal and dual

Primal

Minimize 3x1 + 2x2 + 4x3 (2)

Subject to

2x1 + x2 + ≥ 2

2x1 - x2 +x3 ≥ 5

Dual

Maximize 2y1 + 5y2 (3)

Subject to

2y1 +2y2 ≤ 3

y1 - y2 ≤ 2

y2 ≤ 4

### Properties of duality

The dual of the dual is the primal

When doing the primal simplex, we start with the primal problem and keep on optimizing till we reach the optimal solution, at this point the dual becomes feasible.

### Importance of duality

When we have an optimal solution but we want to add a new constraint, when we add this constraint to the primal it will become not feasible but the dual is still feasible since we only added a new variable to it, we will optimize the dual, so that we make the feasibility of the primal happen.

# Dynamic programming

Dynamic programming is suited for digital computers, and we can always obtain the global optimum using dynamic programming.

Our optimization methods have the word programming in them, we should not be deceived with this word as it doesn’t mean computer programming, it just means optimization.

## Problem domain

The problems solved by dynamic programming have some common features which are:

* the problem involves multi-stage processes
* At each stage, we can use a small number of parameters to describe the state of the process.
* the decisions that we take at any stage transform the set of parameters into a similar one.

Most of the problems solved by dynamic programming are of an optimization or combinatorial type.

The problem that has three properties, which are:

* optimal substructure property, where we can get an optimal solution for a problem by getting the optimal solutions of the subproblems.
* Also, the problem should have overlapping subproblems, which means that finding solutions by solving subproblems again and again.
* the third property is that the problem should be decomposed recursively to a number of problems of the same type.

Max (x1..., xN) (4)

≤ b, xn ≥ 0, n = 1, 2..., N

## Theoretical foundation of dynamic programming

The style of exposition that Bellman used in setting out the conceptual dimension of dynamic programming is astonishing because of the simplicity of presentation. Bellman condensed the theoretical foundation of dynamic programming into a short verbal outline and he never used technical analysis for the theoretical foundation of dynamic programming, Bellman had a conviction that the cluster of ideas underlying dynamic programming has an intuitively compelling validity, immediately intelligible and is very simple, he talked about decision process that goes through many stages, a principle called principle of optimality, and finally he talked about an equation called functional equation.

Principle of optimality states that when we have a problem of multi stages, then all these stages must be solved and result in an optimal strategy, whatever the initial state and decisions are.

So, the basic idea of the theory of dynamic programming is viewing an optimal policy as the thing that determines the required decision in terms of the current state of the system.

## When to use dynamic programming

We use dynamic programming when our problem contains some calculations that are being repeated several times, and the problem can be divided to subproblems, for example if we bought a new bag and the vendor needs to pay you back in some change, we are going to see how to collect cents and coins we need to get to the amount that needs to be paid, instead of going from right to left and decompose the amount of many to coins and cents, the easier way is to go from left to right and that will be much easier, actually the algorithm that is used to solve the change problem is called DP change.

So dynamic programming beats recursion and greedy algorithms in solving these problems, as it provides an easier solution and takes less time than the recursive and greedy algorithms to solve this type of problems.

### Applications

We can solve Fibonacci sequence using full recursion solution, although easy it has a horrible performance, because the growth in time spent is exponential to the value of the function's argument.

Instead we can use a dynamic programming version, we will use an array to keep the values of the already calculated positions to reduce the number of recursive calls.

Performance of Fibonacci sequence problem:

Linear in case of dynamic programming and exponential in case of full recursion.

## How does it work

There is a strategy that we follow when we solve problems using dynamic programming optimization approach.

The great thing about dynamic programming is that it doesn’t go through the same calculations to find a solution for a problem more than once, so instead of wasting time recalculating things, we store the solutions that have been calculated in a storage unit that we visit later when we need the value of a precalculated problem to solve another problem.

Another important thing is that we break our large problem into smaller problems, we begin by solving these smaller problems and at the end we merge the stored value resulting from solving those small problems to get the solution for the big problem.

Also, dynamic programming uses some rules to enhance the time criteria for solving the problem, from those rules we have the principle of optimality which simply means that if we want to get an optimal solution for a given problem we can do that by getting an optimal solution for the subproblems that the big problem is composed of, another rule to enhance the time criteria is that we divide the problem into subproblems of polynomial number, and this is called polynomial break-up.

Dynamic programming can be done using memoization and a top-down approach or following a bottom up approach with the help of a concept called tabulation, in top- down, we use a 2D matrix, array or map and before solving a subproblem we take a look at this storage unit and search for the required value before computing it, if we didn’t find the wanted value we calculate it and store it, and if we found it we use it without wasting time on re-computations.

We can use bottom-up to avoid recursion, we simply solve all the sub-problems, store their results and then merge them to get the solution to our problem.

When we apply dynamic programming technique we need to think about whether it can get the optimal solution, and choose whether we want to apply it recursively or iteratively, and we need to clearly express the recurrence relation to relate the subproblems to the main problem.

If we decided to go with the recursive version, it will be easier and, in most cases, faster but we will face the danger of overflowing the stack.

So, we can summarize the steps of dynamic programming as following:

* Characterize optimal solution structure
* Write a recursive cost function
* Computing recursive values using memoization.
* Use computed information to build an optimal solution.

### Trade-offs between iterative and recursive implementation of dynamic programming:

We have two implementation types, one is recursive and the other is iterative, we are going to compare between them in terms of asymptotic time complexity, memory usage, execution speed, stack overflow and we are going to decide which is easier to implement:

* The two implementation approaches have the same asymptotic time complexity assuming that with recursion we do memoization.
* Concerning memory usage, in the recursive approach we have a recursive stack and sparse memoization, while in the iterative approach we have full memoization.
* Recursive implementation has a faster execution speed, and the execution speed depends on the input, while in iterative approach we do the same work regardless of the input, so, it is often slower.
* Most people find that the iterative approach for dynamic programming is more difficult to implement compared to the recursive implementation approach.

## Global convergence

We use global convergence analysis to find out whether an algorithm that is initiated at far from its solution point will at the end converge to this solution point, thus it is an important aspect of the theory of iterative algorithms.

In iterative algorithms we have a start point from which the optimization begins, an algorithm is said to be globally convergent if for an arbitrary starting point the algorithm always gives points converging to the solution.

If an algorithm A is applied on some point x that belongs to space X will result in a new point that belongs to X

xk+1 = A(xk) (5)

that is the basic idea but when we generalize it we get the A is applied on point x belongs to space X results in a subset of points, and from this subset we choose an arbitrary point xk+1

xk+1 ∈ A(xk) (6)

## Global convergence of dynamic programming

Dynamic programming is globally convergent and guaranteed to find the global optimum but it has downsides that we are going to explain in limitations, these downsides include the curse of dimensionality and menace of the expanding grid, coupled with interpolation problems.

Due to these downsides, our dynamic programming is limited to solve very low dimension optimal control problems.

So, we went for iterative dynamic programming, in which dynamic programming can be used to solve very high-dimensional optimal control problems.

The problem is that iterative dynamic programming is not guaranteed to give us the global optimum, that’s why there are several methods that were developed to enhance the global convergence of iterative dynamic programming.

From the many approaches to enhance the global convergence of iterative dynamic programming we have one approach that applies Sobol's quasi-random sequence generator, and another approach that applies multi pass computation.

Numerical examples clarify the enhancement of the possibility of obtaining the global optimum by the use of multi pass iterative dynamic programming computation with allowable control values and grid points of small number.

## Limitations

From the limitations of dynamic programming is that

* curse of dimensionality which indicates that the volume of computation required to solve the dynamic programming functional equation often increases very rapidly and sometimes exponentially with the size of a problem, so the amount of computation can sometimes make solving the equation a practical impossibility.
* menace of the expanding grid.
* it requires large storage units to store the values of the calculated subproblems so, it is not space efficient.
* Also, sometimes values are stored in the storage unit and these values are not going to be used later.
* It is nontrivial to evaluate the subproblems we have in the most efficient order.
* When we use the recursive and not the iterative form the stack memory increases and may overflow.

# Genetic algorithm

John Holland was the one built the base for genetic algorithms, and laid its most important principles, the standard or canonical of genetic algorithms is based on making the best use out of selection, crossover and mutation, which they are evolutionary operators and we are going to explain them in the section of how genetic algorithm works.

Using this algorithm, we make sure that the population’s fittest individuals will crossbreed reproduce, and their offspring will get genes and characteristics from each parent, and of course as time passes, the offspring will replace older members of the population, we have an assumption which states that the best individuals will give a better offspring because of the good genes that they possess.

Due to the previously stated assumption, the average fitness of the population is expected to improve as several generations passes, and that’s why we can sample the most promising search space areas, at which we highly expect to find the global optimum.

## Problem domain

Problem domains that finds genetic algorithms approach useful are the problems that have a very complicated fitness landscape that requires mixing to methods of child production which are mutation and cross over, the genetic algorithms approach is more powerful than hill climbing algorithm because the latter can get stuck at a local maxima or minima, while genetic algorithms approach can be used to move the population and prevent it from being stuck at a local maxima or minima

Problems that we can use genetic algorithms approach to solve are problems that have timetables like scheduling problems, genetic algorithms approach is often used as global optimization problem solving approach, genetic algorithms are applied to engineering and to many packages of software scheduling.

Some problems are not preferred to be solved by the genetic algorithms approach because it will add more complexity to the problem, an example on these problems is bit strings.

## When to use it

Genetic algorithm approach is the best option when the fitness evaluation method is clear, and they are very helpful when the algorithm is placed decently, but it is better not to use this approach when the constraints of the problem search space are not clearly defined and we should not use this approach when the process of evaluation is expensive from the computation point of view.

### Applications

Genetic algorithm approach is used a lot in the field of machine learning, where using this technique machines and computers can be able to notice and catch patterns, make a smart connection between different values and based on noticing patterns and making connections, machines can be able to make smart decisions and choices, so when all the possible inputs for a certain problem are very large to be covered with examples the machine can learn form the inputted cases and manufacture a useful output for cases that have not been entered manually to the machine.

## Theoretical foundation

First, we have a theorem called schema theorem, which explains the genetic algorithms mathematically, the schema is defined as a pattern of genes values that are binary coded, where each value is either 0 or 1 or a don’t care.

In nature, the healthiest and most fit members are more likely to reproduce more often, that’s why the following generations carry genes that are mostly contributed by the healthy and fit descendants, measuring on this idea we get that the fittest members have the best schema.

We can optimize the search space by representing the reproduction rate as a function of the fitness of a certain member relative to the other members in the same environment.

In each generation, the old schemata are destroyed and a new one is created for the current generation due to the effect of the selection, crossover and mutation of genetic operators.

If we were to pass the same schemata, we would be cloning the old generations over and over again, that’s a big disadvantage because due to the fact that we are cloning without applying any modifications, we won’t get any improvements.

That’s why genetic algorithms use crossover and mutation so that we can recombine schemata to generate a new one, and thus improve our solutions.

The fundamental advantage of genetic algorithm is that it can evaluate large number of schemata by only processing a small number of strings, and this leverage is called implicit parallelism and it is the reason why genetic algorithms are functioning that well.

The principle of implicit parallelism can be illustrated by representing it geometrically using hyperplanes.

M (b, t + 1) = [1-L (b)/ (L -1) J [1- P-mutation (b)] S (b, t) M (b, t) (7)

The above formula is used to calculate the number of schema b instances that will be in the next generation after the effect of selection, crossover and mutation.

## How does it work

The following steps explain how it works:

1. The first step is to create an initial population that is totally random.
2. Then it keeps on creating many sequences of new population by using the current generation and in order to do so, our algorithm works as following:
   1. We assign a score to each member of the current population, and a property called the fitness value is used in the calculations of these scores and we gave a name to these scores which is raw fitness.
   2. Calculate expectation values by scaling the raw fitness scores, because expectation values have a more usable range, based on the previously calculated expectation values, we select the parents, we can select a member as a parent more than once and in that case this member will donate his genes to more than one, Stochastic uniform, is the standard way which lays out a line where each parent is corresponding to a part of this line of a length that is proportional to that parent scaled value, this algorithm moves in steps that are of equal pace, and we choose a parent from each current step.
   3. We pass the members that have lower fitness from the current population to the new one, and we call them elite.
   4. Now, we create children, there are two ways of children creation, either by combining the entries of parent’s pair, or by making random changes to a parent, the first method is called cross over, and the second is called mutation.
      1. cross over children are created by selecting random genes and entries from each parent, and if we have linear constraints, then the created child is a randomly average weighted combination of both parents, in evolutionary programming cross over usually plays a minor rule, but genetic algorithm is a special case as cross over has a very important role in creating children.
      2. Mutation children are created by changing the genes of each parent in a random wat, when we have a bounded problem or a problem with linear constraints, we keep the children feasible, in evolutionary programming mutation also usually plays a minor rule, but genetic algorithm is a special case as mutation has a very important role in creating children.
   5. The last step is to create the new generation by replacing the current population with their children.
3. When we meet one of the following stopping conditions, the algorithm stops, these conditions are:
   1. when we reach the time limit.
   2. When we reach the prespecified number of generations.
   3. When the current population best point has a fitness value smaller than or equal to the fitness limit.
   4. When we have stall generations, which means that the average relative change over the generations is low and smaller than the tolerance of our function.
   5. Also, when the objective function doesn’t improve for a time equal to a prespecified value called stall time limit.

When we have non-linear constraints, we determine the feasibility of our problem.

## Global convergence of genetic algorithm

Genetic algorithm tries to find the best solution that is possible within the available time, so the algorithm will converge but the fact that the best solution that was found is not guaranteed to be the true optimal, and if we wanted to determine the true optimality there is no way but to perform a complete enumeration but that is impractical from time point of view.

So, under the condition of having enough time and If we correctly implemented the genetic algorithm, the population will converge at a global.

When all the genes have converged we can say that the population has converged, and we say that a gene has converged when 95% of the population have the same value for the converged gene, and when population converges as 95% of them have the same values for each gene of the whole set of genes we will notice that the average of fitness will converge to the fitness of the best individual of the population.

## Limitations

Genetic algorithm approach has same limitations such as:

* the random selection of the population that we should start with, we can overcome this limitation by basing the population that we will start with on a partial sample of solutions, and then run the technique to construct the rest of population, this process will help us reduce the time required to reach the optimal solution.
* The use of solid-fitness functions, the strategy followed in genetic algorithm techniques for assigning fitness values to different inputs is by giving the input a low fitness value if it fails to satisfy the tests, we can improve the assignment strategy by assigning high fitness values to the inputs that passes the test, as the latter strategy will help more with finding the optimal solution.
* Despite, global search being more robust compared with local searches, it will be a struggle to find our optimal solution in a large landscape.

# Simulated annealing

The idea behind the name is that the process of melting metals to change their physical structure happens at a very high temperature and then we decrease the temperature gradually till we reach the structure needed, so this approach is an imitation for the process of metal annealing.

## Problem domain

Simulated annealing is used to solve optimization problems that doesn’t have exact methods and deterministic methods are not a practical solution because the implementation of its computations is too complicated.

From the advantages of simulated annealing is that it doesn’t return a false point as the optimum, it can escape from local minima and local maxima, also problems solved by simulated annealing doesn’t require a mathematical model.

The problem implemented using simulated annealing must have a suitable random number generators, because in simulated annealing use the generation of random numbers to solve the problems, random number generators are usually of low quality and are slow, so the domain of problem that is going to be solved using simulated annealing should clarify the amount of random numbers required to solve the problem in order to determine the length and speed of the random number generator that we are going to use, also some problem domains define the required quality of the used random number generator.

It is sometimes difficult to solve a real problem and map it to simulated annealing domain, it needs experience with handling the algorithm.

## When to use it

Simulated annealing is a very important optimization tool that is used for solving both continuous and discrete optimization problems.

It is used to solve problems of managing operations and production as job shop scheduling and single machine, it is also a good solution to difficult discrete optimization problems, it is also used to solve engineering problems that conventional algorithms have a hard time trying to solve, but if the system of the problem is very complex, this solution will be slow.

We also apply simulated annealing to find an optimal solution for discrete problems that have a permutation property, an example of that type of problems is traveling sales man problem.

Also, we apply it to continuous global optimization problems, where there are two classes, in the first class we imitate the physical annealing of metals process, and in the second class we apply stochastic differential equations to get the solutions for optimization problems that are not discrete but of a continuous nature.

### Applications

Simulated annealing is used to solve travelling salesman problem, this problem is used to identify the shortest path that passes by each city and returns to the one that the journey started at, this a popular np combinatorial optimization problem.

## How does it work

Simulated annealing algorithm works as following:

* If our goal was to minimize the solution to our optimization problem and find its global minima then we will start with a variable analogous to temperature in the metal example, and we are going to decrease that variable till we reach a good enough solutions at a low value for that variable, and if we were to find a global maxima then we would start with a variable that is of a very low value and we are going to increase the value of that variable and find a good enough maxima at a high value.
* As we move from large values to lower ones on the x-coordinate, we will have to accept solutions that are worse than our current solution, and the idea behind that is because we don’t want to get caught in a solution that turns out to be a local solution and not a global one, so accepting neighbor solutions especially at high values of x-coordinate is a must.
* The idea behind accepting a new solution, or sticking to the one we have is governed by an equation, based on which we take a new solution or we return the current solution we have as a good enough solution.

P = e(S2-S1)/T (8)

Where S2 is the neighbor solution, S1 is the current solution and T is the value of the x-coordinate at each solution, if the result P is greater than a number that is randomly generated then we will accept the neighbor solution, else we keep our solution and we notice that the division by the variable, shows that as we are at a high value of the variable the possibility of accepting a new solution is high, which makes sense because the optimal minimum solution exists at a small value of that variable.

## Theoretical foundation

There are two theorems that are fundamental to simulated annealing, the first is called statistical equilibrium, statistical equilibrium is based on a hypothesis called ergodicity hypothesis which states that we can imagine a system of particles as a set with statistical properties that can be observed easily, if this system of particles is at a state of equilibrium we can get some quantities that are useful for example: entropy and energy distribution, the second step is to combine the states that we found in state of equilibrium with probability density and we will find that the probability density depends on the energy and we can calculate this probability using Boltzmann law.

Δu = u – uo = eσA [T4 – T04] (9)

The second theorem is related to convergence so, we are going to talk about it in the next section.

## Global convergence of simulated annealing

The second theorem is that the algorithm for simulated annealing has a property, this property is the stochastic convergence to the global optimum, this is resembled by an infinite diagram for temperature decay, and this decay goes on with infinite small steps, but of course this is theoretical because in real life we have a limited time of execution, so the plan is to get as close as possible to the optimum within the available time, so we have a concept called Markov chain, Markov chain is a state sequence, and this sequence has a property that states that the probability of getting to the next state depends on the previous state, so we are going to resemble the state that we get to at the iteration number with X(k), the transition probability for a pair of states is given by the following relation.

Pij(k)=probability[X(k)=j|X(k-1) = i] (10)

Pij is called transition matrix, in simulated annealing, the transition of Markov-chain is corresponding to a state space move.

In order to make sure that simulated annealing converges towards the global optimum, we have to use a suitable schedule for cooling, when we have a complex landscape for our function, it will be difficult for simulated annealing to escape from local solutions and get the global optimum, we can solve this problem by using simulated tempering which means raising the temperature, or increasing the value of the variable analogous to temperature, but still the convergence will be slow in case of functions with a complicated environment.

## Limitations

The problems that we may encounter when we solve a problem by simulated annealing are:

* If it is a minimization problem, and the variable analogous to temperature in metal annealing process sis initialized at a very high value, we will keep accepting solutions and we might jump to high values and accept them, so too much perturbation is not beneficial.
* If the variable decreases very fast, as in the metal annealing process the temperature should decrease slowly so that we can be able to enhance our solution, and get closer and closer to the optimal solution, so there are two types for cooling or reducing the value of temperature analogous variable, if this variables decreases in an exponential manner this means that we will move from the high values fast, and then we will traverse slowly at the low temperature parts where the optimal solution most likely exists so we will traverse slowly and won’t mess it, that’s why the exponential cooling is suitable for simulated annealing, but if the cooling analogous to the value reduction is linear, this means that we will spend the same amount of time at each value, so this is preferred when there are lots of neighboring low valleys but it will take long time to find the minimum optimal
* If we don’t choose our parameters carefully, there is a big chance that the algorithm ends without finding an optimal maxima or minima.

# Branch and bound

## How does branch and bound work?

In branch and bound, we make a combination between relaxation techniques and partial enumeration strategy.

We form classes of solutions and we investigate whether they contain optimal solutions or not.

We draw a branch and bound tree

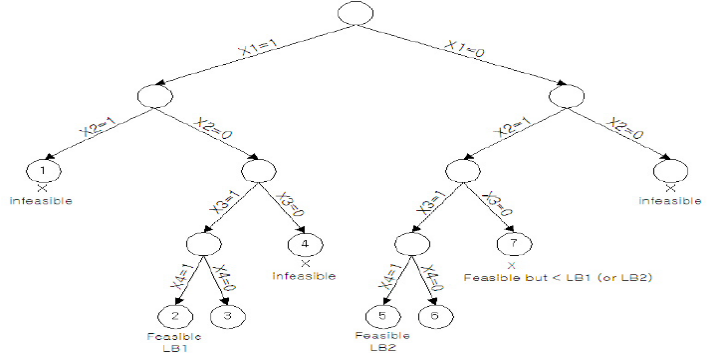


Fig (1) Branch and bound

We start at the root node and we perform relaxation, which gives us an upper bound in case of maximization problems and lower bounds in case of minimization.

The possible outcomes of the relaxation at the root:

1. We find that the problem is infeasible.
2. We are lucky and we find an optimal solution
3. We get a fractional value of a binary variable, so we branch to 0 or 1.

At intermediate nodes we find incumbent solutions, which are the best-found feasible solutions till now.

We terminate when we face the following cases:

* Infeasible solution
* The value can’t be improved on the incumbent
* We found our optimal solution and completed the branching

There are several strategies for selecting the next node because our goal is to reduce the total time consumed till we find the optimal solution, from these strategies we have best first and depth first.

We will start with best first, our strategy here is to reduce the time taken to find the solution by reducing the size of the solution tree, we do that by selecting which sub-problem to begin with, where we start with the best sub-problem that gives the lowest lower bound in case of minimization and the highest high bound in case of maximization, but that doesn’t always work because usually the best solution is usually achieved when we dive deep into the tree, also this will lead to high usage of memory.

The second strategy is depth first, which works by always selecting the deepest node in the tree, so we keep diving deep into the tree and then we backtrack, it minimizes the memory used, but this will be a waste of time if we initiate the solution with a lower bound that is not good enough.

## when do we use branch and bound?

Branch and bound is used to solve combinatorial and mathematical optimization problems that can be of linear constraints, linear constraints mixed with integer constraints for some variables as the case in mixed integer linear programming and nonlinear constraints.

There is a way to know whether branch and bound is a useful method to solve a problem, we use branch and bound when the problem involves a decision that should be taken regularly at different time intervals, or there are a number of resources that need to be assigned to some investments or other gain purposes, we combine it with LP relaxation when we have an integer constraint.

If we want to solve NP problem and this problem has linear constraints but the variables have to be integer then we will be in a subset of linear programming called mixed integer linear programming, because Linear programming solves P problems only.

Mixed integer linear programming is NP hard and it has several methods to solve NP problems, so we use branch and bound for solving mixed integer programming problems and, in this case, we can combine the branch and bound with other methods.

One of these methods is doing a process called LP relaxation which ignores the integer constraint and solve the problem as a linear programming problem, if the result was found to be of an integer value then we are lucky and we found the optimal solution for the initial mixed integer linear programming problem.

So, there is a relation between branch and bound and mixed integer linear programming as branch and bound combined with LP relaxation enable us to solve NP problems like the knapsack problems when integer constraints exist.

Mixed integer is the same thing as linear programming but with one more detail, which is that some variables must be integers, while linear programming allows variables to be real numbers and have fractional values, in integer linear programming no variable is allowed to have a fraction result, the value must be a whole value.

linear programming was invented by Dantzig in year 1947, and it has been a growing area for many years.

A fundamental tool and a beautiful connection between visual geometrical intuition and algebraic expression for a given problem.

The domain of the problem is not affected by whether we are using MILP or using linear programming, only the whole idea about the fact that it is not acceptable for a certain variable to have a fractional value.

If our problem has a feasible solution, which means a solution that satisfies all constraints, then a BFS exists, also if we have an optimal solution, which means it achieves the desired maximum or minimum, then the existence of a basic optimal solution is a must, it is impossible for an that is infeasible or doesn’t have a bound to have a solution that is optimal.

### LP Relaxation

We can prove infeasibility using linear relaxation, where if the problem after relaxation is infeasible this means that the mixed integer problem is infeasible, LP relaxations can produce optimal solutions that we can round to good feasible solution for the optimization problem.

One of the ways to solve problems that have linear constraints and also integral constraints is by using optimistic relaxation, in which we remove the integral constraints and start solving our problem using simplex method, if we get integer results then the problem is solved if we don’t we might round the results and see if they give us an optimal solution.

Simplex algorithm is a connection between algebraic and geometric view

Simplex algorithm is a local search algorithm and it is guaranteed to find the global optimum because of the convexity, optimal solution for our objective function is at a vertex, a vertex is a BFS which is an abbreviation for Basic Feasible Solution, you can move from one BFS to a neighboring one and you can detect whether it is optimal, from any BFS you can move to a BFS with a better cost

We need a BFS to begin with and to be able to achieve the last point in the outline, so, we create a fake BFS to solve the real BFS, which means that we add some variables called artificial variables to our equations and we find the BFS for them and from this BFS we start moving from one BFS to another, this is done by entering a non-basic variable called entering variable and in return kicking out one of the basic variables and we call it leaving variable.

So, we swap a basic and a non-basic variable and after that we perform Gaussian elimination.

The previous operation is called pivoting.

### Applications of branch and bound

Branch and bound is used to get the minimum stability of a system whose parameters are not fixed, also to the distance between the stable situation that we are currently in and being unstable, also it is used in finding the optimal solution for NP problems like 0-1 knapsack problem and traveling sales man problem, this problem is used to identify the shortest path that passes by each city and returns to the one that the journey started at, this a popular np combinatorial optimization problem.

## Global convergence of branch and bound

When branch and bound algorithm has a certain selection method that doesn’t change, it will converge, but in case of the spatial branch and bound algorithms that are used to solve mixed integer nonlinear programming, there is no guarantee that after a finite number of steps we will converge, when we combine branch and bound with LP we will find that, there is a part in LP relaxation solved using simplex, we find that the simplex algorithm finds the solution after an unknown but finite number of steps, the number of steps maybe exponential to the program size, as the resultant sequence is limited and finite, while using interior point methods we converge as close as possible to the global optimum.

## Limitations of branch and bound

Branch and bound can consume too much time in a path that ends with not finding and optimal solution, this tends to happen when we branch and the nodes that we have seem to be promising so we keep on branching and applying relaxation to find our bounds and dive deep into the tree and in the end we find that the branch that we have been working on doesn’t lead to an optimal solution, although as the size of the problem increases, branch and bound may end by having execution time that is exponential to the size of our problem, also, it is difficult to make the branch and bound algorithm parallel in execution, that’s due to the need to balance the weight and load.

## Problem domain of branch and bound

As we explained before branch and bound is used in solving mixed integer programming problems which are difficult to solve so we relax them and solve them using branch and bound.

The problem domain is an objective function to be minimized or maximized and is defined by linear equations and inequalities and has at least one variable that has to be integer.

The basic form of a mixed linear programming problem is a certain number of variables for example k variables and a certain number of linear constraints let’s call it h those k variables have to be non-negative and all the constraints come in the form of equations or inequalities.

Minimize Ax1 + Bx2 (11)

subject to

x1 −x2 ≤ 1

x1, x2 ≥ 0

x1, x2 to Z

We relax this problem and remove the integer constraints and then solve it using branch and bound and hope for the best.

Minimize Ax1 + Bx2  (12)

subject to

x1 −x2 ≤ 1

x1, x2 ≥ 0

## Theoretical foundation of branch and bound

Branch and bound has a fundamental theorem called Dantzig theorem which states that if we have a NP problem for example 0-1 Knapsack problem and the statement of the problem is as following:

Kj = 1 at j = 1 up to S

Kj = 0 at j= s+2 up to N

Ks+1  (13)

Then the upper bound of the problem will be equal to

(14)

If we assumed that P = (x, y): Ax + Gy ≤ b where P is projection, and A, G are rational matrices and b is a rational vector.

If we have S = (x, y) ∈ P: x integral, and S is not equal to phi, also by using Minkowski-Weyl theory

Which states that P is equal to the summation of conv(V) and cone(R) where R is an integral matrix and V is a rational matrix, and if we calculated the truncation of P and gave it the symbol T, we will find that it is a rational polytope, because it is a bounded projection of a polyhedron.

If we have T1 which has values of the ordered pair (x, y) that belongs to T and all the values of x is integer, conv(T1) gives us a rational polytope.

If we have an ordered pair (x1, y1) belongs to S, also x1 is an integral, and the summation of λi is equal to 1 where i is from 1 to p and all the values of λ satisfying the following relation.

(x1, y1) = + + (15)

Thus S = T1 + R where R is set of integral conic combinations.

Conv(S) is equal to the summation of conv(T1) and cone(R) and it is a rational polyhedron that has P recession. From that we get that conv(S) is equal to a set of (x, y) ordered pairs that satisfies the relation A’ x + G’y ≤ b’.

# 0-1 Knapsack problem

In this problem, our goal is to fill our bag with the most valuable items that can fit in the bag, and get the best outcome and the best money value, it is called 0-1 because we either take the item or not.

So, our goal is to maximize the value without passing the capacity constraint.

Greedy algorithms can always find the optimal solution for fractional knapsack, that is not the case in 0-1 knapsack, greedy algorithm sometimes fail to find the optimal solution, so as an improvement for the implementation of greedy algorithm, in the code of greedy algorithm we sorted items according to the ratio between their value and weight like we do in fractional knapsack, but the difference here is that we don’t take a fraction of an item, we either take it as a whole or leave it.

In fractional knapsack, each item can be divided in to smaller parts so the implementation of the greedy algorithm of fractional knapsack is still a bit different from the implementation of our code for 0-1 knapsack, where we move on the sorted array of items that is sorted in a descending order according to the ratio between the value and weight and we take a whole item, when it is it’s turn in the list and also adding it wouldn’t pass the predefined capacity of our knapsack.

Genetic algorithm also doesn’t always get the optimal solution, but the possibility of genetic algorithm to find the solution is higher than that of greedy algorithm.

Dynamic programming always finds the optimal solution, in the submitted code we used recursion and memoization to find the solution for our 0-1 knapsack problem.

Simulated annealing also converges towards the optimal solution, it might find a solution near the optimal solution, but it can escape from local maxima and local minima.

In branch and bound, we start at a node and we perform bound relaxation, after that we branch and at each node we perform bound relaxation, and if our goal was to maximize we pick the highest upper bound and stop branching.

* Genetic algorithm has the worst-case time complexity of O(N).
* Simulated annealing has the worst-case time complexity of O(N).
* Greedy algorithm has a worst-case time complexity of O(NlogN).
* Dynamic programming with a worst-case time complexity O(W\*N), where W is the capacity of the knapsack and has also Ө(W\*N) and Ω(W\*N) and also a space complexity of O(W\*N).
* branch and bound has a pretty bad worst-case time complexity of O(2n).
* the worst time complexity of divide and conquer is also O(2n).
* the worst time complexity of brute force is also O(2n), best case complexity Ө(2n), and average case complexity of Ω(2n).

## Testing the different solution methods

All codes were tested on the same device and were written using java, all on the same IDE which is netbeans.

We stored the current time before execution and after execution, the difference between them was returned as the execution time.

The capacity of the knapsack in all codes was 50\*number of items, the price and weight were filled in a loop, weight is 10\*item-ID and price is 50\*item-ID, so all the numbers are obtained under the same circumstances for fair comparison between the different methods.

The following table compares the different methods at small sizes

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Size** | **dynamic** | **divide&conq** | **Bruteforce** | **Greedy** | **B&B** | **Genetic** | **SA** |
| 10 | 0.047 | 0.002 | 2.05 | 0.049 | 0.05 | 0.019 | 0.012 |
| 20 | 0.052 | 0.03 | 7.186 | 0.049 | 0.054 | 0.025 | 0.02 |
| 30 | 0.058 | 7.754 | NA | 0.049 | 0.057 | 0.035 | 0.029 |
| 40 | 0.063 | NA | NA | 0.05 | 0.062 | 0.049 | 0.047 |
| 50 | 0.062 | NA | NA | 0.054 | 0.063 | 0.057 | 0.052 |
| 60 | 0.064 | NA | NA | 0.054 | 0.066 | 0.07 | 0.058 |
| 70 | 0.067 | NA | NA | 0.054 | 0.068 | 0.079 | 0.073 |
| 80 | 0.067 | NA | NA | 0.06 | 0.079 | 0.085 | 0.091 |
| 90 | 0.068 | NA | NA | 0.06 | 0.082 | 0.094 | 0.125 |
| 100 | 0.072 | NA | NA | 0.06 | 0.094 | 0.107 | 0.223 |

Table(1) Execution time vs problem size

Fig(2) Execution time vs problem size graph 1

At large values we will find that the execution time of branch and bound increases exponentially, and at large sizes genetic algorithm and simulated annealing and dynamic programming are better, genetic algorithm and dynamic programming are the best options on the long run, but dynamic programming requires large storage, so we prefer genetic algorithms.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Size** | **Genetic** | **SA** | **B&B** | **Dynamic** |
| 100 | 0.107 | 0.223 | 0.094 | 0.072 |
| 200 | 0.315 | 0.359 | 0.504 | 0.087 |
| 300 | 0.523 | 0.737 | 3.865 | 0.159 |
| 400 | 0.883 | 1.428 | 22.591 | 0.281 |
| 500 | 1.137 | 2.064 | NA | 0.445 |
| 600 | 1.876 | 3.444 | NA | 0.93 |
| 700 | 2.352 | 5.713 | NA | 1.317 |
| 800 | 3.198 | 6.313 | NA | 1.919 |
| 900 | 3.415 | 8.227 | NA | 2.835 |
| 1000 | 4.312 | 10.522 | NA | 3.4 |

Table(2) Execution time vs problem size

Fig(3) Execution time vs problem size graph 2

# References

1. Moshe sniedovich, “Dynamic programming foundations and principles,”, 2nd ed., pp. 357-377, 2011.
2. Rein luus,”Monographs and surveys in pure and applied mathematics: iterative dynamic programming”, pp. 96, 2000
3. Darrall Henderson, Sheldon H. Jacobson, Alan W. Johnson,” the theory and practice of simulated annealing”, vol 57, pp. 291-301, 2003
4. Dr. Gerhard Niemeyer, ”efficient production planing and scheduling: an integrated approach with genetic algorithms and simulation”, pp. 37-51 , 1996
5. Alberto Guida,” a branch and bound algorithm for the global optimization and its improvements”,pp. 5-12,pp. 19-24 ,2015