KWAME NKRUMAH UNIVERSITY OF SCIENCE AND TECHNOLOGY

KUMASI GHANA

COLLEGE OF SCIENCE

SCHOOL OF GRADUATE STUDIES

OPTIMIZING MEMORY USING KNAPSACK ALGORITHM

By

EVANS BAIDOO (BEd. Information Technology)

A Thesis submitted in partial fulfilment of the

Requirement for the degree

MASTER OF PHILOSOPHY (Information Technology)

July 2015

DECLARATION

I, Evans Baidoo, hereby declare that this thesis, "Optimising Memory Using Knapsack Algorithm", consist entirely of my own research work produced from research undertaken under supervision and that no part of it has been published or presented for another degree elsewhere, except for the permissible excepts/references from other sources, which have been duly acknowledge.

Evans Baidoo

(Student Number PG1140813) **……..**

…….. (Signature)

(Date)

Certified by:

…….. ……..

Mr. Dominic Asamoah **(Signature)** **(Date)**

(Supervisor)

Certified by:

…….. ……..

Dr. J. B. Hayfron-Acquah **(Signature)** **(Date)**

(HOD Computer Science)

DEDICATION

This research work is dedicated to my lovely mother Madam Diana Amissah and the Amissah family. Your prayers, encouragement and support has brought me this far and will always be remembered and cherished.

May God richly bless you and may you live long to enjoy the fruits of your labour.

ACKNOWLEDGEMENT

I wish to express my profound gratitude to the almighty God for His tender mercy and loving kindness and for given me the strength up to this level. He indeed has been faithful to honour all his promises.

I thank my supervisor, Mr D. Asamoah, for many insightful discussions that help develop the ideas in the thesis. I express deep gratitude to him for his advice and support and also to Mr Emmanuel Ofori Oppong for being the source of constant motivation and inspiration throughout my thesis and his highly perceptive comments.

An extended gratitude to my internal examiner, Dr. Ahmed Missah, for his highly perceptive comments and suggestions.

I would like to thank my colleague KGEE ICT masters, Felix, Maxwell and Aboagye, for their useful suggestions, propositions and leverage.

May the good Lord shine his face upon you all.

To Derrick Lamptey, I say thank you for your support in the software development of this thesis and to Stephen Oppong for his many support in diverse ways.

Finally, to my family who have supported me in all aspect of my education and friends, Bismark, Boakye Yiadom and Larko Osei, whose advice and encouragement have strengthened me when it was most necessary. I say thank you to you all.

ABSTRACT

Knapsack problem model is a general resource allocation model in which a single resource is assigned to a number of alternatives with the aim of maximizing the total return. Knapsack problem has been widely studied in computer science for years. There exist several variants of the problem. The study was about how to select contending data/processes to be loaded to memory to enhance maximization of memory utilization and efficiency. The instance is modeled as 0 – 1 single knapsack problem.

In this thesis a Dynamic Programming (DP) algorithm is proposed for the 0/1 one dimensional knapsack problem. Problem-specific knowledge is incorporated in the algorithm description and evaluation of parameters, in order to look into the performance of finite-time implementation of Dynamic Programming.

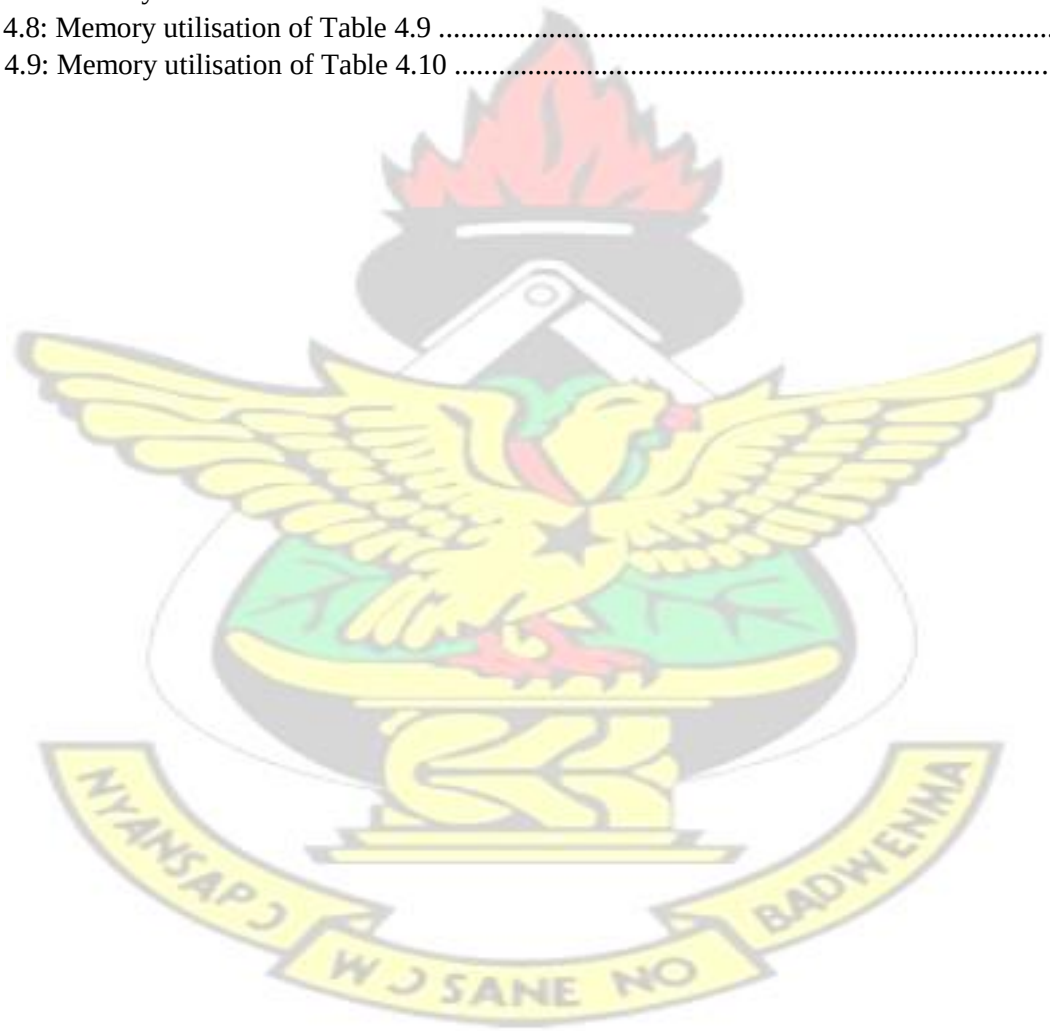
Computer implementation considerations played an important role in its development. We test the presented method with a set of benchmark data and compare the obtained results with other existing heuristics. The proposed method appears to give good results when solving these problems unravelling all instance considered in this study to optimality in "reasonable" computation times. Computational results also shows that the more the number of items or processes, the higher the time to produce optimal results.

Conclusively, when the curse of dimensionality can be dismissed, dynamic programming can be a useful procedure for large sequencing problems.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES



CHAPTER ONE

1.0 Introduction

Earlier computers had a single-level scheme for memory. Computer evolution has moved from gigantic mainframes to small stylish desktop computers and to low-power, ultraportable handheld devices with in a relatively short period of time. As the generations keep passing by, computers making up of processors, memories and peripherals turn out to be smaller and faster with memory prices going up and down. However, there has not been a single main memory that was both fast enough and large enough even though computers were becoming faster and programs were getting bigger, particularly multiple processes that were simultaneously carried out under the same computer system. Though putting more random access memory (RAM) in the computer is nearly at all times a good investment, but it is not really advisable to spend extra money to get full benefit from the memory you already have, if there is an effective algorithm to ensure effective memory utilisation.

Hard drives sizes are becoming bigger (in four years it moved out from 4Gb to 40Gb as a 'reasonable' hard drive) but memory at all this while is by far highly-priced than storage, so operating system desires to utilize free disk space as virtual memory.

The operation of computers is such that, the more programs execute, the slower the computer goes. Not only is the virtual memory on the hard drive a hundred times sluggish than real memory, but shambling one lump of information from physical memory into the swap file that hoard virtual memory on the hard drive to create storage for the program, a different request is seeking for takes time too. Avoiding this problem will only means needed applications should be allowed to run. It should be noted that one of the most critical resources that a computer wishes to handle is its memory. All software executes in some form of storage and all data is stored in some form of memory.

Modern computer memory management is for some causes a crucial element of assembling current large applications. First, in large applications, space can be a problem and some technology is efficiently needed to return unused space to the program. Secondly, inexpert implementations can result in extremely unproductive programs since memory management takes a momentous portion of total program execution time and finally, memory errors becomes rampant, such that it is extremely difficult to find programs when accessing freed memory cells. It is much secured to build more unfailing memory management into design even though complicated tools exist for revealing a variety of memory faults. It is for this basis that efficient schemes are needed to manage allocating and freeing of memory by programs.

Optimizing current memory management strategies strength is performed by altering the space allocated to each task. To achieve high levels of multiprogramming while avoiding thrashing such policies vary the load (i.e., the number of active tasks). Additionally, in a system that runs out of capacity probably because the system is undersized, several options are available. This option includes either upgrading the processor (if possible), reduce available functionality, or optimize.

A great deal of realistic problems where some predefined conditions are respected such that the sum of the values of the selected entities is maximized can be represented by a set of entities, each having an associated value, from which one or more subsets has to be selected. The most ordinary situation is obtained by establishing that the sum of the entity sizes in each subset does not exceed some prefixed bound by associating a weight/size to each entity. Knapsack problems are generally what these problems are called. The theory of the knapsack problem is such that given a set of items, each with weight and value, whiles keeping the overall weight smaller than the limit of the entire pack, settle on which items to pick to maximize the value. Putting forward a practical example, suppose a thief invaded a house and inside the house he identified the objects as follows:

A jug worth $70 and weighs 3 pounds

A silver nugget worth $45 and weighs 6 pounds

A work of art worth $55 and weighs 4 pounds

A reflector worth $30 and weighs 5 pounds Which

objects should be picked?

Considering this set as a small problem, it is evident that the appropriate response is the jug and the work of art, for a sum total value of $125, however if there exist a tall list of objects calculating the answer would be a herculean task. In such an instance it is obvious that the best possible choice of objects to pick may indicate close to the optimal solution of the knapsack problem.

The allocation schemes to handle assigning and releasing of memory efficiently over a time period by a process or task is an interesting research topic on itself, but this research work will not go into this problem. The problem that will be considered in this thesis is that of accepting or rejecting the Process (an occurrence of an executed program) as they come in from the process queue to compete for memory space when a user request to run a program. The goal is to maximise the number of processes in a limited memory space.

1.1.1 OS - Memory Management

The purposefulness of an operating system which holds or controls primary memory is Memory management. Memory management oversees each and every memory locality either it is free or it is given to some process. When a process is created it verifies how much memory is to be allocated to it. It respectively brings up to date the status and tracks when some memory gets freed or unallocated and as well as decides the memory to be given to a process and at what time.

Memory management gives insurance by utilizing two registers, a base register and a limit point register. Whiles the limit register identifies the size of the range, the base register stores the smallest legal physical memory address. For instance, if the limit register is 1109000 and the base register stores 200000 then, the program through 411999 it can legally access all addresses from 200000. The following is the way information and data to memory addresses is made

- compile time binding is employed to produce the absolute code when it is known at compile time the location of the process
- The compiler produces re-locatable code after it has not recognized at compile time where the process will be stored in memory. This activity occurs during Load time
- At run time binding must be delayed to be done if the process can be shifted during its implementation from one memory segment to another. This is at the Execution stage. Thus under any given condition, data or instructions can be addressed to various memory cell locations.

The primary memory of the PC is productively oversees by the operating system signifying the obligation of overseeing processes The memory manager is the part of the operating system which takes charge of this duty. The entire system performance is made crucial by the performance of the memory manager since in order to execute every process must have some amount of primary memory.

As explained by Infosys Technologies Education Solutions Limited in their book in 1997, "The memory manager is in charge of distributing basic memory to processes and for helping the software engineer in stacking and putting away the essential's primary memory.Managing the sharing of primary memory and minimizing memory access time are the basic goals of the memory manager."

A system which runs multiple processes at the same time has a real challenge of efficiently managing memory. The memory manager assigns a fraction of primary memory to every

process for its system use since primary memory can be space-multiplexed. Nevertheless, the memory manager must establish how to allot and de-allot available memory when new processes are started and when completed processes finish execution at same time keeping track of which processes are running in which memory locations.

For their special use dynamic memory allocation is executed by current operating systems. They may also carry out as required the same procedure for their applications, or permit the applications to allocate and de-allocate memory to programs that may clinch programming interface functions (APIs such as malloc). In order to better the performance of the projected system, real time operating system uses unlike memory allocation strategies and algorithm.

Two basic types of memory managements get into attention in Real Time Operating System (RTOS), the stack and the heap (Diwase et al 2012). Where memory is given out for mechanical variables within functions is considered the stack. A stack performs it operation based on Last In First Out (LIFO) basis where a fresh space is assigned and de-assigned at only one "end", called the Top of the stack. During program implementation the memory given out in the stack area is used and reused. The heap on the other hand presents more established storage of data in favour of a program. For the duration of a program, memory given out in the heap stays in existence. Thus, static variables and global variables (storage class external) are assigned on the heap. The program until it makes use of memory allocated in the heap area, if started at zero at program start will remains unchanged. Therefore the heap area does not need to hold garbage.

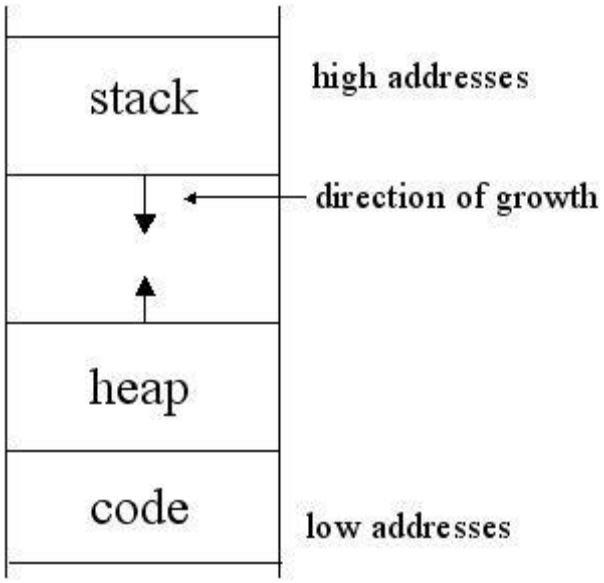


Figure 1.0: Stack versus Heap allocation
Source: http://www.cs.cornell.edu/

The stack and heap both contain memory that computer program can use. The figure 1.0 above shows that they grow in opposite directions. In between the stack and the heap is a free memory which can be assigned to the stack or the heap.

Asma'a Lafi (2013) explained that task control blocks are used in context switching throughout stack management where as memory excluding memory used for program code is concerned with heap management and it is used for dynamic allocation of data space for system jobs. Real time operating system concurs static and dynamic memory management concurs

1.1.2 Virtual Memory

Frequently discussion of virtual memory is identified as a way to extend the random access memory by means of hard drive as slower, additional, system memory. Through this means systems allows the computer to use extra memory than is available physically by using hard disk space. Technically, virtual memory permits data to be shuffled from the paging file to memory as needed and again to make room for new data, data from memory is moved back to the paging file. In particular, the computer system streams onto the hard drive which is

used as "virtual" memory once it runs out of memory. Present operating systems refer to this as swap space. Swap space comes into attention when the system request for extra memory resources and the physical memory (RAM) amount needed is full. By this, dormant pages in memory are switched to the swap space (Wienand, 2013).

"Virtual" as it is called is only because it is not physical memory. It does not mean it is fake simply because initially, the thought of using disk to expand RAM is what is meant as virtual memory. A problem attributed to disk is that it's slow to access. If disk is accessed in fractions of seconds then registers can be accessed in 1 nanosecond and cache in 5 ns and RAM in about 100 ns. It may be 1,000,000 times slower to access disk than a register however the disk has the benefit of easy accessibility to a lot of disc space for alittle price (Lin 2013).

Genuine, or physical, memory exists on RAM chips inside the PC. Virtual memory, as prior demonstrated, does not actually exist on a memory chip. It is an enhancement strategy and is executed by the working framework keeping in mind the end goal to give an application program the feeling that it has more memory than really exists. According to Lin (2003), Virtual memory is an old idea. Earlier, PCs had reserve as well as virtual memory. For quite a while, virtual memory just showed up on centralized servers. PCs in the 1980s did not utilize virtual memory.

Virtual memory is used as a means of memory protection with current systems. Each program utilizes a scope of location called the address/location space. The presumption of working system designers is that any client program can not be trusted. Client programs will attempt to devastate themselves, other client programs, and the operating system itself. That appears like such a negative perspective, in any case, it is the manner by which operating systems are composed. Lin (2003) stressed that, it is not obliged that programs must be deliberately hurtful yet can be coincidentally (modifying the information of a pointer guiding to garbage

memory).Virtual memory help prevent programs from interfering with other programs in addition to ensuring programs cooperation and sharing of memory.

An advantage of virtual Memory system is by permitting a greater number of procedures to keep running than the permitted memory size. This is accomplished by just including parts of procedures that are obliged to keep running in memory and the remaining on disk. Without a doubt the base piece of a procedure that must dependably be in memory is called its working set. More often than not, a program does not have to have its whole binary record in memory to run when it is performing an errand that just uses some piece of its file. This means, say, a 16MB system could joyfully keep running on a machine with just 4MB of memory.

The aggregate size of all procedure location spaces is not constrained by the measure of the physical memory when every procedure is dispensed a virtual location space that is not an immediate toss of main memory. Every procedure's location space, which for this situation is its virtual location space, can be as large as the whole address space in physical memory in the event that it needs. The location space of a processor alludes the scope of conceivable locations that it can utilize when stacking and putting away to memory. The location space is constrained by the registers' width, on the grounds that to stack a location a heap instruction is issued with the location to stack from put away area in a register. For instance, registers that are 32 bits wide can hold addresses in a register range from 0x00000000 to 0xFFFFFFF.

2^32 is equivalent to 4GB so a 32 bit processor can load or store to up to 4GB of memory (Bottomupcs.com, 2015). In the most recent couple of years, PC processors are as a rule each of the 64-bit processors, which as the name infers has registers 64 bits wide giving a centrality that there is less or no compelling reason to spare interim variables to memory when the compiler is under register pressure incompletely in light of the fact that each program aggregated in 64-bit mode needs 8-byte pointers, which can increase code and information size, and in this way affect both instruction and data cache intensity.

Virtual Memory can be actualized in one of two ways: Paging and Segmentation. Segmented Virtual Memory takes into account the assurance and migration of processes' memory sections and in addition the sharing of libraries between processes. Paged Virtual Memory moreover permits the Virtual Memory's size to surpass the greatest size of the physical memory since it puts just parts of a program that are needed for an errand in memory whilst the remaining is stored on disk (Cyberiapc.com, 2015). From this perspective, it can noticed that exchange off from memory to disk of unmoving or blocked processes can make space for different processes that might want to execute and subsequently, is a capable method for using memory.

According to Nutt (1997), "virtual memory strategies allow a process to use the CPU when only part of its address space is loaded in the primary memory". In this approach, every process's location space is apportioned into parts that can be stacked into primary memory when they are required and composed back to optional memory otherwise. "Another finished after-effect of this approach is that the system can run programs which are really bigger than the essential memory of the system, consequently the thought of "virtual memory."

Brookshear (1997) explains how this is accomplished: "Suppose, for example, that a main memory of 64 megabytes is required but only 32 megabytes is actually available. To create the illusion of the larger memory space, the memory manager would divide the required space into units called pages and store the contents of these pages in mass storage. A typical page size is no more than four kilobytes. As different pages are actually required in main memory, the memory manager would exchange them for pages that are no longer required, and thus the other software units could execute as though there were actually 64 megabytes of main memory in the machine". The memory manager stays informed concerning every one of the pages that are filled into the primary memory all together for this system to work.

1.1.3 Process Concept

A process is a case of a program in execution. At the point when a PC is running, numerous programs are simultaneously sharing the CPU. Every running program, in addition to the data structures expected to oversee it, is known as a process. Numerous advanced process ideas are communicated as jobs, (e.g. job scheduling), and the two terms are frequently utilized reciprocally. For every process there is a Process Control Block, PCB, which stores processspecific data (specific information may change from system to system).

- Process State -, waiting, running etc.,
- parent process ID and Process ID
- Program Counter and CPU registers - These should be spared and restored when swapping processes all through the CPU
- CPU scheduling data - Such as priority information and pointers to scheduling queues.
- Memory management data - E.g. segment tables or page tables.
- Accounting data - account numbers, kernel and user CPU time consumed, limits, etc.
- I/O status information -, Open file tables, devices allocated etc.

In modern systems a solitary process can be permitted to have various strings of execution, which execute simultaneously. All started processes are put away in a job queue as they vie for space to be stacked into the memory for CPU scheduling. The job scheduler essential goal is to present an adjusted blend of jobs, for example, I/O bound and processor bound. This is to guarantee that the CPU is kept occupied at all times and to convey "acceptable" reaction times for all programs, especially for intelligent ones. Processes may be in one of 5 states, as shown in Figure 1.1.

- New - The process is in the phase of being made.
- Ready - The process has every one of the resources accessible that it needs to run, however the CPU is not at present taking a shot at this present process' instructions.
- Running - The CPU is taking a shot at this present process' instructions
- Waiting - The procedure cannot keep running right now, in light of the fact that it is waiting on some resource to be made accessible or for some event to happen. For instance the process may be waiting on keyboard information, disc access request, inter-process communications, a clock to go off, or a started process to wrap up.
- Terminated - The process has finished.

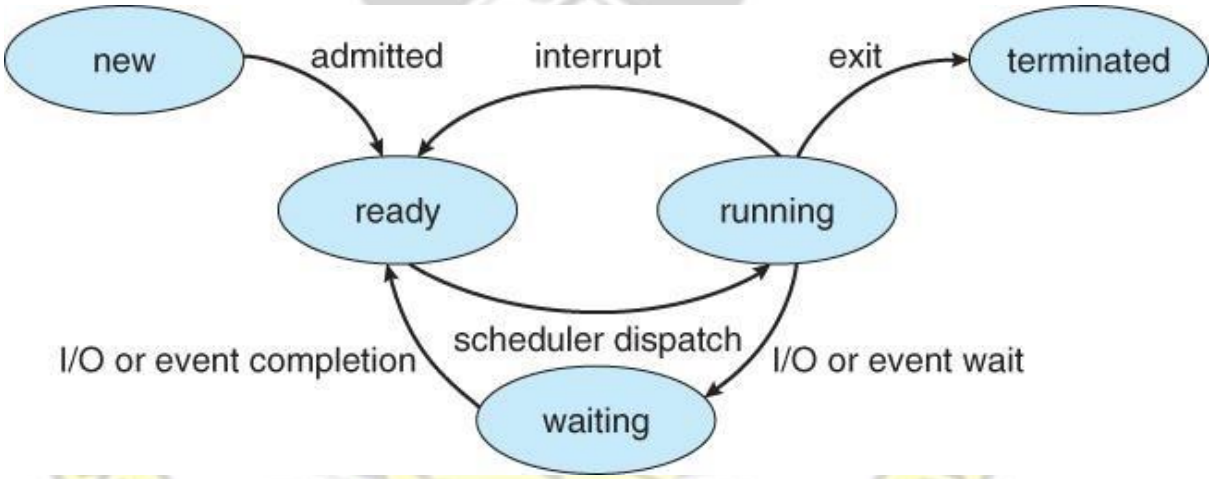


Figure 1.1: Diagram of process state

Source: Operating System Concepts, Ninth Edition

Processes have memory and their memory is divided into four sections.
The text area contains the ordered program code, read in from non-unstable capacity when the program is started.

- The data section stores worldwide and static variables allotted and introduced before executing primary memory.

- The heap is utilized for dynamic memory distribution, and is overseen through calls to new, erase, malloc, free, and so on.
- The stack is utilized for local variables. Space on the stack is held for nearby variables when they are announced (at function entrance or somewhere else, dependent on the language), and the space is freed when the variables go out of extension.

When processes are swapped out of memory and later restored, additional information must also be stored and restored. Key among them is the program counter and the value of all program registers

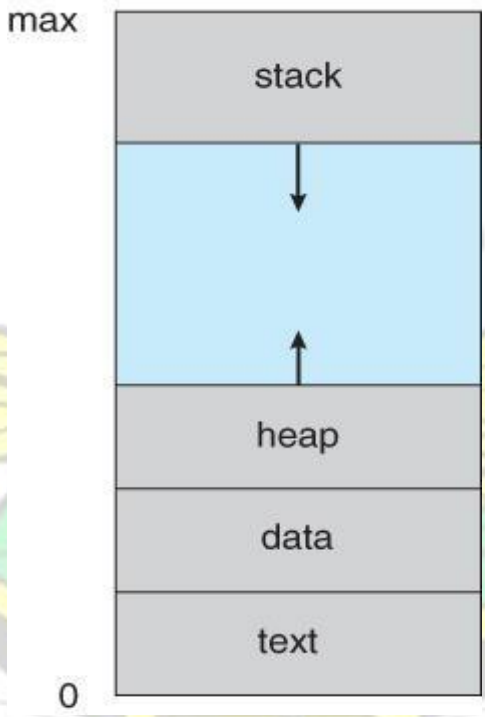


Figure 1.2: A process in memory

Source: Operating System Concepts, Ninth Edition

There are a variety of possibilities when the CPU chooses which process to run. These include:

- When process transforms from running to waiting. This could be a direct result of IO request, wait for started program to end, or wait for synchronization operation (like lock acquisition) to finish.

- When process changes from running to ready - on culmination of interrupt handler, for instance. Basic sample of interrupt handler - clock interrupt in intelligent systems. if scheduler changes processes for this situation, it has preempted the running process. Another regular case interrupt handle is the IO completion handler. ○ When process changes from waiting to ready state (on termination of IO or acquisition of a lock, for instance). ○ When a process ends.

1.2 Problem Statement

In a computer system, a process as soon as created wants to run. There is a number of N created processes all contending for memory space to run. All process want to fill the main memory that can hold an aggregate weight of W. If all processes are allowed to run, it will lead to system crushes, system running low memory, system underperformance, system overheat and difficulty in accessing data. We want to fill the main memory that can hold an aggregate weight of W with some blend of data/process from N possible list of data/process each with data size w_i and priority value v_i so that optimal utilisation of memory of the data/processes filled into the System main memory is achieved hence maximized.

Among the data/processes contending for memory,

Which of them should be allowed to run and which should not?
Does the allowable data make optimal use of memory without memory losses?

The problem considered is a combinatorial optimisation problem which represents a variety of knapsack-type problems. The problem identified can be modelled as a 0/1 knapsack problem. A mathematically formulation of the 0-1 Knapsack Problem (KP) is stated in the ensuing integer linear function.

Maximizes ____ $\sum_{j=1}^{n} P_j x_j$ (1.1)

Subject to $\sum_{j=1}^{n} (w_j x_j) \leq C$ ____(1.2)

x_j = 0 or 1, j = 1... N

Where, P_j refers to the value, or worth of item j, x_j refers to the item j, w_j refers to the relative-weight of item *j*, with respect to the knapsack and C refers to the capacity, or weightconstraint of the knapsack. There exist $j = 1 \ldots n$ items, and there is only one knapsack.

1.3 Objective

The main objective of the study is to model a real-life computer problem of data loading into memory for CPU scheduling as a 0-1 knapsack problem, and determine an effective algorithm to solving this problem to achieve optimum memory efficiency.

1.4 Justification

PC memory constitutes countless number of flip flops at the physical level. Every flip flop comprises of a couple of transistors, and has the capacity store one bit. Individual flip flops are addressable by a remarkable identifier so it can be perused and overwrite them. Accordingly, thoughtfully, as Humans, we can think about the majority of our PC's memory as only one monster cluster of bits. Since as humans, we are not that great at doing the greater part of our reasoning and number-crunching in bits, we assemble them into bigger gatherings, which together can be utilized to represent numbers. 8 bits are called 1 byte; past bytes, there are words (which are infrequently 16, often times 32 bits).

The computer's memory more often is abstractly regarded as one large (size 232 or so) array of bytes. This memory holds a lot of things which comprises the programs' code, including the operating systems, other data used by all programs and also all the variables.

Memory management is taken care of by the working together of the operating system and compiler, but as a smart computer user, it is imperative to see what is going on under the hood

and to determine if there are more better way to prevent the buying and replacing of hardware when new programs which require higher memory are created often resulting in systems not only running out of memory but also the frequent crushing of data.

Assuming process/data are items and memory is a big knapsack and we would like to stack this knapsack that can store a sum weight of C with a number of combinations of items from N possible list of items each with weight W_i and value V_i to ensure that the items value filled into the knapsack is increased.

The identified problem has an included confinement that every item ought to be allotted space apart from the reality that it has a linear objective function with a single linear limitation which totals the values of the items in a queue. Assuming N is the items total number, then there exist 2^N subsets of the item collection. “So an exhaustive search for a solution to this problem generally takes exponential running time. Therefore, the obvious brute-force approach is infeasible. However, specialized algorithms can, in most cases, solve a problem with n =100,000 in a few seconds on a mini/micro computer” (Pisinger, 2003). A few correct and approximate algorithms, for example, dynamic programming and polynomial estimation created by Fayard-Plateau and Balas-Zemel, Horowitz-Sahni's greedy algorithm, meta-heuristics calculation, among others can be utilized to tackle the knapsack also called the rucksack problem. The knapsack problem is NP-complete to tackle precisely, therefore it is normal that no known calculation can be both right and quick (polynomialtime) on all cases, and numerous cases that emerge by and by can in any case be explained precisely.

In view of these, studies of knapsack problems and their algorithms has been an area of much interest in the contribution to academic knowledge due to its occurrence in most daily activities, hence the reason for investigating the problem using Dynamic programming.

Knapsack problems particularly in the last decade have been rigorously studied, dragging both theorists and patricians possibly due to its substance to integer programs. “The theoretical

interest arises mainly from their simple structure which, on the other hand allows exploitation of a number of combinatorial properties and, on the other, more complex optimization problems to be solved through a series of knapsack-type sub-problems. From the practical point of view, these problems can model many industrial situations: capital budgeting, cargo loading, cutting stock." (Salkim and Derkluyer Knapsack problems and survey). Additionally, Knapsack problems are used in all spheres of our daily activities: in financial decision making, bid prices, routing of vehicles etc. A binary integer program can be viewed as a knapsack constraint if only it has a single constraint.

1.5 Scope and Limitations of the Study

This study is within the confines of a Data loading into the memory for CPU scheduling i.e. accepting or rejecting a Data as they come in from the process queue to compete for memory space.

The survey will consider a single 0-1 knapsack problem, where a single container is packed with best possible subset of items. We will denote the capacity of the container C. The computer solution developed in Java programming language will be adapted to work out the single 0-1 knapsack problem.

The common situation where a container of n capacities C_i (i = 1,. . . , n) are presented, often referred to as multiple knapsack problems is not considered.

1.6 Organization of Thesis

The study is prepared in five chapters.

Chapter 1 presents background of the Knapsack Problems, the Operating system and memory management issues and the problem statement, objective, methodology and justification for optimising memory using knapsack algorithm.

Chapter 2 make available relevant literature review of Knapsack problems applications, memory allocation dynamics and the solution approaches being proposed and adopted in

literature.

Chapter 3 is the methodology of this research thesis. It is devoted to the algorithms for the solution method. Here the dynamic programming algorithm is introduced and explained

Chapter 4 provides the implementation of data and outcome of study of simulated data.

Chapter 5 presents the conclusion and future work.

1.7 Summary

In the preceding chapter, a discussion of the formulation of the knapsack problem to the solution of data allocation of computer programs, computer memory hierarchies and unit issues, the background of the case study area (management of computer memory), and the justification of the thesis. In the next chapter, we shall put forward the literature pertinent in the area of 0-1 knapsack problems and memory allocation dynamics.

CHAPTER TWO

LITERATURE REVIEW

2.0 Introduction

An integral part of our current technological advancement is memory; whether it is a mobile phone, or the electronics in an auto-mobile, an IPod or even a router. These developments have been touching and altering modern lives like never before. Memory management for several reasons have become a critical area in modern large applications.

Knapsack problem largely considered as a discrete programming problem has become one of the most studied problem. The motive for such attention essentially draws from three facts as stated by Gil-Lafuente et al. “ (a) it can be viewed as the simplest integer linear programming

problem; (b) it emerges as a sub-problem in many more complex problems; (c) it may signify a great number of practical situations" (Gil-Lafuente et al., 2013).

In this chapter, a review of literature on Memory management processes and Knapsack problems and applications is presented.

2.1 Memory management, Processes and its allocations

The computer system primary memory management is key. After all, "all software runs in memory and all data is stored in some form of memory. Memory management has been studied extensively in the traditional operating systems field" (Silberschatz & Peterson, 1989). Memory management includes giving approaches to dispense bits of memory to programs at their request, and liberating it for reuse when not really required.

Chen et al. (2010), identified possibilities of managing memory in smart home gateways. They asserts that "due to different architecture, memory management for software bundles executed in home gateways differs from traditional memory management techniques because traditional memory management techniques generally assume that memory regions used by different applications are independent of each other while some bundles may depend on other bundles in a gateway". By way of contribution, they introduce a service dependency heuristic algorithm that is close to the optimal solution based on Knapsack problem but performs significantly better than traditional memory management algorithms and also in a general computing environment identified the difference between memory management in home gateway and traditional memory management problem

Algorithms in support of memory management bearing garbage collection or explicit deallocation (as in C's malloc/free) have gained enough studies. Known for decades, regionbased memory management as an alternative approach has not been considered until lately. In region-based memory management objects can not be liberated separately; each allotted object is positioned in a program-specified region and regions are removed with all

their contained objects. In his work, Gay (2001) identified two varieties of memory safety: “temporal safety (no accesses to freed objects) and spatial safety (no accesses beyond the bounds of objects)” to resolve traditional region-based systems for instance arenas in which removing a region might retain dangling pointers that are later accessed increasing memory usage. Using a dialect of C with regions that ensures temporal safety dynamically Gay plan, execute and assess RC.

Restricting their consideration to virtual machines (VMs), in a position paper, Singer and Jones (2011), looked at how economic hypothesis can be connected to memory management. They watched the correspondence between the economic idea of a customer and a request of a virtual machine running a single program in a segregated heap. “Economic resource consumption corresponds to the virtual machine requesting and receiving increased amounts of heap memory from the underlying operating system. As more memory is allocated to a virtual machine's heap, there is additional benefit (economic utility) from the extra resource”. They additionally examine production and cost capacities, which may help with effective memory allocation between different virtual machines that are going after an altered amount of collective system memory.

Modern microprocessors have reached limits in instruction level parallelism and on-chip power capacity. On-node concurrency levels have also increased dramatically. “Moore’s law is still active and fine but the rising transistor count is now used to build extra processing units instead of faster single-threaded cores” (Giménez et al., 2014). To support on-node parallelism, the memory architecture has also become more complex. This has severely complicated the task of extracting peak performance. Optimizing memory access is critical for execution and power effectiveness. To create report of exact expenses of memory accesses at particular locations, CPU producers have created sampling-based "Performance units (PMUs)". On the other hand,

this information contains an inordinate measure of irrelevant or uninteresting data and is too low-level to be seriously understood.

Adopting a semantic approach, Giménez et al (2014), made a study on-node memory access performance. To give data which may contain the connection important to viably decipher information, they built up a system and an instrument to assemble fine-grained memory access execution data for particular data objects and areas with low overhead and credit semantic data to the examined memory access. The tool performs sampling and attribution and discovers and diagnoses performance problems in real-world applications. As noted by Gim´enez et al (2014), this technique provides "useful insight into the memory behaviour of applications and allows programmers to understand the performance ramifications of key design decisions: domain decomposition, multi-threading, and data motion within distributed memory systems".

It is an acknowledged fact that, packet switching architecture - distributed shared memory (DSM) – has drawn much notice as of late, basically because of "its ability to triumph over the inherent memory-bandwidth limitation of output-queued switches" (Li and Elhanany, 2005). DSM execution has been contemplated from a hypothetical perspective by investigating the conditions under which it can copy an output-queued switch. "At the core of the DSM design is a memory management algorithm that determines the memory units to which arriving packets are forwarded. However, the complexity of such algorithms found to date is O(N), where N denotes the number of ports in the system, thereby inherently limiting the scalability of the scheme". Li and Elhanany (2005) put forward a fresh pipelined memory management algorithm for DSM switches which not just shows how handling and memoryaccess speedup elements yield a very versatile DSM switch structural design but additionally offering diminished timing density at the expense of settled latency.

As program advancement is sifting its way into a more extensive programming community and more conventional state-based dialects from the practical programming community, "the view

that a more disciplined approach to memory management becomes a very important aspect (Coquand et al., 1999). For example, web programming languages such as Java and Python include garbage collection as part of the language, and there are various packages for performing memory management in C and C++". This therefore calls for a formal understanding of memory management. Goguen et al. (2002) developed an incremental tracing algorithm to mange memory. They focused on accepting incremental tracing, a principally error-prone area of garbage collection. "Tracing is the first two phases in garbage collection and consists of a systematic search through memory distinguishing those memory cells that are in use by the program from those that are not". Goguen et al. (2002) stressed that, they adopted the incremental algorithm since a few programs can be utilizing memory as the memory management system is hunting down the unused cells or de-allotting those that have been found. Embedded systems performance is verified by several factors of which the process in which memory is managed is an important one of them. Jingwei et al. (2012) perform a research on "optimizing software of memory management on ARM design." They identified and discussed three software optimization methods of memory management based on ARM embedded system and embedded Linux. This includes among other experiments, "setting memory access permissions in system initialization to improve the reliability of the embedded system, using FCSE (Fast Context Switch Extension) to improve the usage efficiency of memory and finally, the allocation of memory space by integer times of 4 KB to make more processes concurrency possible on embedded system which hardware configuration is insufficient." Jingwei et al. (2012) concluded that optimizing memory management with any or all of the above stated methods improve to a larger extent the efficiency of operation of embedded Linux system and ARM.

Kornaros et al. (2003) describe a fully programmable memory management system optimizing queue handling at multi gigabit rates. They argue that "the architecture of a memory manager can provide up to 10Gbs of aggregate throughput while handling 512K queues". They stated

that “two of the main bottlenecks when designing a network embedded system are very often the memory bandwidth and its capacity. This is mainly due to the extremely high speed of the state-of-the-art network links and to the fact that in order to support advanced quality of service (QoS), per-flow queuing is desirable.” In their estimation, their introduced system support a full instruction set and trust it can be utilized as a hardware component as a part of any appropriate embedded system, especially network SoCs (system on chip) innovation that actualize per flow queuing partly because of the way that when outlining this scheme a few optimisation techniques were assessed and the most cost and performance efficient ones utilized (Kornaros et al., 2003). These strategies minimize both the memory transmission capacity and the memory limit required, which is viewed as a fundamental point of preference of the proposed scheme.

In implementing of multimedia applications, it must be created to put together a high memory bandwidth, low power, high speed, and large data storage capacity. The support of real-time memory de/allocation, processing and retrieving of information is allowed by its run-time memory management. Leeman et al. (2005) based their findings on a proficient technique of improving and optimising of dynamic memory management for embedded systems in multimedia applications. They evaluate the execution of a recently made system-level investigation technique to improve the memory management of normal media applications in a broadly utilized D recreation image system. Their strategy depends on an examination of the quantity of memory accesses, standardized memory footprint1 and energy approximations for the system contemplated bringing about a change of standardized memory foot print and the evaluated energy dispersal over traditional static memory usage in an Optimized form of the driver application. In their final delivery, Leeman et al. (2005) asserted that, their created system was able to “scale perfectly the memory consumed in a system for a wide range of input parameters whereas the statically optimized versions are unable to do this.”

The vital necessity of a memory management system is to give approaches to dynamically assign parts of memory to programs at their request, and liberating it for reuse when no more significant. This is crucial to the PC system. “Several methods have been devised that increase the effectiveness of memory management. Virtual memory systems separate the memory addresses used by a data from actual physical addresses, allowing separation of processes and increasing the effectively available amount of RAM using paging or swapping to secondary storage”. Vasundhara Rathod and Pramila Chawan (2013) presented a study of memory management systems of an operating system and the implementation of clock with adaptive replacement. They compared the memory management subsystems of BSD 4.4, Linux 2.6 and Windows and then developed a new algorithm clock with adaptive replacement using temporal filtering for resourceful usage of memory. They concluded that “the quality of the virtual memory manager can have an extensive effect on overall system performance.”

An important operation of present computer system is computer memory since it keeps program instructions or data on a permanent or short-term basis for use in a computer. Nevertheless, there is an escalating gap created involving the pace of microprocessors and the pace of memory. Zhu and Qiao (2012) presented a survey on computer system memory management and optimization techniques. They reviewed together with the hardware designs of the memory organization like the memory hierarchical structure and cache design, diverse memory management and optimization procedures to ease the gap; virtual memory techniques from an old bare-machine process to segmentation and paging approaches and the memory management procedures changing from replacement algorithms to optimization techniques.

In the event of a page error virtual memory system needs productive page substitution algorithms to choose which pages to expel from memory. “Over the past years numerous algorithms have been recommended for page replacement. Each algorithm tries to decrease the page fault rate while sustaining minimum overhead. As newer memory access patterns were

discovered, research mainly focused on preparing newer approaches to page replacement which could adjust to changing workloads" (Chavan et al., 2011).

Chavan et al. (2011), attempted to study the latest methodologies such as "CLOCK-Pro, Low Inter-reference Recency Set(LIRS), CLOCK with Adaptive Replacement (CAR) and Adaptive Replacement Cache (ARC)" and tried to sum up key page replacement algorithms suggested till present thereby taking a look at conventional algorithms such as "CLOCK, Least Recently Used(LRU) and Belady's MIN".

2.2 Knapsack problems and applications

The knapsack problem (KP) is a traditional combinatorial issue used to show numerous modern circumstances. It has been concentrated seriously in the previous decade pulling in both scholars and experts. The scholars' interest emerges principally from their basic structure which both permits abuse of various combinational properties and allows more perplexing optimiation problems to be illuminated through a progression of Knapsack type.

"Since Balas and Zemel a dozen years ago introduced the so-called core problem as an efficient way of solving the Knapsack Problem, all the most successful algorithms have been based on this idea. All knapsack Problems belong to the family of *NP-hard* problems, meaning that it is very unlikely that we ever can devise polynomial algorithms for these problems" (Pisinger, 1994).

Year	Author	Solution Proposed
1950s	Richard Bellman	Produced the first algorithm - dynamic programming theory - to exactly explain the 0/1 knapsack problem.
1957	George B. Dantzig	. He gave an exquisite and productive strategy to obtain the answer for the continuous relaxation issue, and henceforth an upper bound on z which was utilized as a part of all studies on KP in the accompanying a quarter century

1960s	Gilmore and Gomory	Among other knapsack-type problems he explored the dynamic programming approach to the knapsack problem
1967	Katherine Kolesar	Experimented with the first branch and bound algorithm of the knapsack problem.
1970s	Horowitz and Sahni	The branch and bound methodology was further created, turned out to be the main approach fit for taking care of issues with a great amount of variables.
1973	Ingargiola and Korsh	Presented the initial reduction formula, a preprocessing algorithm which significantly reduces the number of variables
1974	Johnson	Gave the first polynomial time approximation design to solve the problem of the subset-sum; Sahni extended the result to the 0/1 knapsack problem.
1975	Ibarra and Kim	They introduce the first completely polynomial time approximation design
1977	Martellon and Toth	Proposed the first upper bound taking over the charge of the continuous relaxation.
The key products of the eighties concern the resolution of mass problems, for which variables cataloguing (required by all the most effective algorithms) takes a very high percentage of the running time.		
1980	Balas and Zemel	Introduced another way to deal with the issue by sorting, much of the time, just a little subset of the variables (the core problem). They demonstrated that there is a high likelihood for discovering an ideal solution in the core, in this way abstaining from considering the remaining objects.

Table 1.0: Core history of Knapsack Problems and Solutions

The Knapsack problem has been concentrated on for over a century with prior work dating as far back as 1897. “It is not known how the name Knapsack originated though the problem was

referred to as such in early work of mathematician Tobias Dantzig suggesting that the name could have existed in folklore before mathematical problem has been fully defined" (Kellerer, 2004).

Confronted with instability on the model parameters, robustness investigation is a suitable way to deal with dependable solution. Kalai and Vanderpooten (2006) examined the hearty knapsack problem utilizing a maximum-min conditon, and proposed another robustness method, called lexicographic α-vigor. The authors demonstrated that "the complication of the lexicographic α-robust problem does not augment compared with the max-min version and presented a pseudo-polynomial algorithm in the case of a bounded number of scenarios".

Benisch et al. (2005) experimented the problem of selecting biased costs for clients with probabilistic valuations and a merchant. They demonstrated that under specific suspicions this problem can be summary to the "continuous knapsack problem" (CKP). They introduced another quick epsilon-optimal calculation for unravelling CKP occurrences with asymmetric concave reward capacities. They likewise demonstrated that their calculation can be stretched out past the CKP setting to handle pricing problems with covering merchandise (e.g. products with normal parts or basic asset necessities), as opposed to indistinct merchandise. They gave a structure to taking in dispersions over client valuations from past information that are exact and perfect with their CKP calculation, and approved their strategies with trials on evaluating prices got from the "Trading Agent Competition in Supply Chain Management (TAC SCM)". Benisch et al. (2005) results confirmed that their method converges to an epsilon-optimal result more rapidly in practice than an alteration of an earlier recommended greedy heuristic.

"Ant Colony optimization (ACO) algorithm" is a novel replicated transformative algorithm, which gives another strategy to combinatorial optimization problems. It is a probabilistic strategy for tackling computational issues which can be lessen to discovering great ways through charts. ACO point is to hunt down an ideal way in a chart in light of the conduct of ants looking for a way between their state and a supply of food. Shang et al (2006) solved the

Knapsack problem using this algorithm. ACO was enhanced in determination procedure and data change so that it can't undoubtedly keep running into the local optimum and can meet at the global optimum. Their try-out illustrates the strength and the probable control of this kind of meta-heuristic algorithm. Kosuch (2010) presented an "Ant Colony Optimization (ACO) algorithm" for the Two-Stage Knapsack case with discretely dispersed weights and limit, using a meta-heuristic approach. Two heuristic utility measures were proposed and compared. Additionally, the author keeping in mind the end goal to acquire a paradigm for the construction termination presented another thought of non-utility measures. The author argued why for the proposed measures, "it is more efficient to place pheromone on arcs instead of vertices or edges of the complete search graph. Numerical tests show that the author's algorithm is able to produce, in much shorter computing time, solutions of similar quality than CPLEX after two hour. Moreover, with increasing number of scenarios the percentage of runs where his algorithm is able to produce better solutions than CPLEX (after

2 hours) increases." Boryczka (2006) offered a new optimization method for the Multiple Knapsack Problem based on Ant colony metaphor. "The MKP is the problem of assigning a subset of n items to m distinct knapsacks, such that the total profit sum of the selected items is maximized, without exceeding the capacity of each of the knapsacks. The problem has several difficulties in adaptation as well as the trail representation of the solutions of MKP or a dynamically changed heuristic function applied in this approach." attainable outcomes showed the strength ACO approach for solving the Multiple Knapsack Problem.

Branch and Bound is a class of accurate algorithm for different optimization problems particularly whole number programming problems and "combinational optimization problems" (COP). It isolates the solution space into little sub-problems that can be understood autonomously (branching). Bounding disposes of sub issues that cannot contain the ideal solution, thereby diminishing the solution space size. Branch and Bound was initially proposed via Land and Diog in 1960 for tackling whole number programming.

The branch and bound algorithm when applied to the Knapsack model which is single constrained reduces the total numbers of sub-problems. This approach enhances the performance of the algorithm by generating and adding new objective function and constraints to the Knapsack model. (Munapo, 2008). The author stated further that "majority of algorithms for solving Knapsack problem typically, use implicit enumeration approaches. Different bounds base on the remaining capacity of the knapsack and items not yet included at certain iterations have been proposed for use in these algorithms." Comparable routines may be utilized for a nested Knapsack problem as long as there is a set up method for testing whether an element entered into a Knapsack at one stage can likewise be entered at the next stages.

Florios et al. (2009) tackled an example of the "multi-objective multi-constraint (or multidimensional) knapsack problem (MOMCKP)", with three target capacities and three limitations. The creators requested for an accurate and approximate algorithm which is a legitimately altered form of the "multi-criteria branch and bound (MCBB) algorithm", further tweaked by suitable heuristics. Three branching heuristics and a more universally useful composite branching and construction heuristic were worked out. Moreover, the same issues were unravelled utilizing standard "multi-objective evolutionary algorithms (MOEA)", to be specific, the SPEA-2 and the NSGA-II. The outcomes from the definite case demonstrate that the branching heuristics incredibly enhance the execution of the MCBB algorithm, which turns out to be quicker than the versatile ε - constraint. With respect to execution of the MOEA algorithms in the particular problems, SPEA-2 beats NSGA-II in the level of estimate of the Pareto front, as measured by the scope metric (particularly for the biggest occurrence). In the middle of the 1970s "several good algorithms for Knapsack Problem (KP) were developed (Horowitz and Sahni 1974), (Nauss 1976), and (Martello and Toth 1977). The starting point of each of these algorithms was to order the variables according to nonincreasing profit-to-weight (pj/wj) ratio, which was the basis for solving the Linear KP." From this explanation, appropriate upper and lower bounds were derived, making it possible to apply some rational analysis to fix

several variables at their optimal limit. Finally the KP in the remaining variables was solved by branch and bound techniques. This approach was applied by Essandoh (2012) to model site development for solid waste disposal in SekondiTakoradi metropolis as a 0-1 knapsack problem. His method could be adopted for any land site management problem to obtain an optimum refuse disposal management. Most prominent adopters to his study were district assembly for refuse disposal management and waste management companies.

Given a knapsack of limit, Z, and n dissimilar items, Caceres and Nishibe (2005) algorithm resolved the single Knapsack problem using local computation time with communication rounds. With dynamic programming, their algorithm solved locally pieces of the Knapsack problem. The algorithm was implemented in Beowulf and the obtained time showed good speed-up and scalability (Robert and Thompson 1978).

Jan et al. (2006) with a constriction for server-based adaptive web systems took to Web content adaptation with a bandwidth. The problem can be stated as follows: "Given a Web page P consisting of n component items $d_1, d_2, \ldots, d_n$ and each of the component items d_i having J_i versions $d_{i1}, d_{i2}, \ldots, d_{iJi}$, for each component item d_i select one of its versions to compose the Web page such that the fidelity function is maximized subject to the bandwidth constraint". They invented this problem as a "linear multi-choice knapsack problem (LMCKP)" and changed the LMCKP into a knapsack problem (KP) and afterward introduced a dynamic programming approach to calculate the KP. A numerical sample demonstrates the method and showed its viability.

A general indication of the latest methods for taking care of hard Knapsack Problems, with unique stressing on the expansion of cardinality limitations, dynamic programming, and simple divisibility where computational results, looking at all late algorithm, were exhibited has been given (Martello et al., 2000).

"An approach, based on dynamic programming, can be used for solving the 0/1 multiobjective knapsack problem. The main idea of the approach relies on the use of several complementary

dominance relations to discard partial solutions that cannot lead to new nondominated criterion vectors" (Bazgan et al., 2007). This way, they acquired a productive technique that outflanks the current schemes both in terms of CPU time and size of tackled situations. Broad numerical trials on different sorts of examples of multidimensional task were accounted for. A correlation with other precise strategies was likewise performed. "The data association problem consists of associating pieces of information emanating from different sources in order to obtain a better description of the situation under study. This problem arises, in particular, when, considering several sensors aimed at associating the measures corresponding to the same target" (Hugot et al., 2006). This problem widely in literature is frequently expressed as a 16 multidimensional task problem where a state model is upgraded. While this methodology appears to be tasteful in effortless circumstances where the danger of confounding targets is generally low, it is a great deal harder to get a right description in denser circumstances. Hugot et al (2006) proposed to address this subject in a several criteria framework by means of a second integral model, in view of the recognizable proof of the objectives. Because of the problem specificities, a trouble-free and effective methodology can be utilized to produce non-dominated solutions. Moreover, they showed that "the accuracy of the proposed solutions is greatly increased when considering a second criterion. A bi-criteria interactive procedure is also introduced to assist an operator in solving conflicting situations".

Silva et al. (2008) managed the issue of incorrectness of the solutions produced by metaheuristic methodologies for combinatorial optimization bi-criteria knapsack problems. A version of the "stochastic knapsack problem with normally distributed weights, the two-stage stochastic knapsack problem" was studied by Kosuch and Lisser (2009). Contrary to the single-stage knapsack problem, "items can be added to or removed from the knapsack at the moment the actual weights become known (second stage). In addition, a chance-constraint is introduced in the first stage in order to restrict the percentage of cases where the items chosen lead to an overload in the second stage". According to the authors, "there is no method known to exactly

evaluate the objective function for a given first-stage solution, and therefore proposed methods to calculate the upper and lower bounds. These bounds are used in a branch and bound framework in order to search the first-stage solution space. Special interest is given to the case where the items have similar weight means with numerical results presented and analyzed".

As more complex as it has become, Zhong and Young (2009) explained the exploits of an integer programming tool, "Multiple Choice Knapsack Problem (MCKP)", to present ideal results to transportation programming - a procedure of selecting projects for financing given spending plan and different constraints-problems in situations where substitute variants of projects are considered. Optimization methods for utilization in the transportation programming procedure were analyzed and after that the procedure of building and taking care of the optimization problems examined. The ideas about the utilization of MCKP were exhibited and a real-world transportation programming illustration at different budget levels were given. They gave convenient arrangements in transportation programming practice as well as outlined how the utilization of MCKP addresses the current complexities.

Lin and Yao (2001) investigated a Knapsack problem where every one of the weight coefficients are "fuzzy numbers". The work was based on the assumption that "each weight coefficient is imprecise due to the use of decimal truncation or rough estimation of the coefficient by the decision maker. To deal with this kind of imprecise data, fuzzy sets provide a powerful tool to model and solve this problem." Their work was expected to wiped out the first Knapsack problem into a more universal problem that would be valuable in viable circumstances. As a result, their study showed that "the fuzzy Knapsack problem is an extension of the crisp Knapsack problem in a special case of the fuzzy knapsack problem".

A number of stochastic and mathematical techniques have been produced in a bid to solve "multi-objective problems". The techniques worked in view of scientific models while much of the time these models are radically simplified images of real world issues. A study conducted by Gholamian et al. (2007) as an alternative to mathematical models exploit a hybrid intelligent

system *i*. The chief core of the scheme is fuzzy rule base which links from a solution space (*X*) to decision space (*Z*). "The system is designed on non-inferior region and gives a big picture of this region in the pattern of fuzzy rules. Since some solutions may be infeasible; then specified feed forward neural network is used to obtain non-inferior solutions in an exterior movement". In addition, "numerical examples of well-known NP-hard problems (i.e. multi-objective travelling salesman problem and multi-objective knapsack problem)" were given to illuminate the exactness of created system (Boyd and Cunningham, 1988).

There are studies of income maximization in the dynamic and stochastic knapsack problem (Gallego et al. 1994), and (Gershkov et al. 2009), "where a given capacity needs to be allocated by a given deadline to sequentially arriving agents. Each agent is described by a two-dimensional type that reflects his capacity requirement and his willingness to pay per unit of capacity". Dizdar et al. (2010) performed one of the studies and characterize implementable policies. They solved "the revenue maximization problem for the special case where there is private information about per-unit values, but capacity needs are observable after deriving two sets of additional conditions on the joint distribution of values and weights under which the revenue maximizing policy for the case with observable weights is implementable, and thus optimal also for the case with two-dimensional private information". In particular, they investigated "the role of concave continuation revenues for implementation". They also created a "simple policy for which per-unit prices vary with requested weight but not with time, and prove that it is asymptotically revenue maximizing when available capacity/ time to the deadline both go to infinity". This draws attention to the significance of nonlinear as compared to dynamic pricing.

The multidimensional 0/1 knapsack problem is a combinatorial optimization problem, which is NP-hard and occurs in countless fields of optimization. The multidimensional 0/1 knapsack problem is a standout amongst the most surely understood integer programming problems and has gotten wide consideration from the operational exploration community amid the most

recent four decades. Albeit late advances have made conceivable the results of medium size cases, tackling this NP- hard problem remains an exceptionally fascinating test, particularly when the quantity of limitations increments. Fréville and Plateau (2004) surveyed the main results published in the literature and focused on the "theoretical properties as well as approximate or exact solutions of this special 0–1 program". The multidimensional 0/1 knapsack problem, characterized as a knapsack with multiple resource limitations, is surely understood to be a great deal more complex than the single limitation description. Freville and Plateau (2004), designed an efficient pre-processing procedure for large-scale instances. Their algorithm "provides sharp lower and upper bounds on the optimal value, and also a tighter equivalent representation by reducing the continuous feasible set and by eliminating constraints and variables". This plan was appeared to be exceptionally compelling through a great deal of computational investigations with test problems of the literature and large scale randomly created cases.

The "Knapsack sharing problem" (KSP) is put together as an addition to the usual knapsack problem. It is an NP-Hard combinatorial optimization problem, acknowledged in several real world situations. "In the KSP, there is a knapsack of capacity, c and a set of n objects, namely {\cal N}, where each object j ,j = 1… n, is associated with a profit pj and a weight wj . The set of objects {\cal N} is composed of m different classes of objects. The aim is to determine a subset of objects to be included in the knapsack that realizes a maximum value over all classes". Yamada et al. (1998) solved the knapsack sharing problem to optimality by presenting a "branch and bound algorithm and a binary search algorithm". These algorithms are executed and computational tests are done to break down the conduct of the created algorithms. As a result, they found that "the binary search algorithm solved KSPs with up to

20,000 variables in less than a minute in their computing environment".

Important classes of combinational optimization problems are the “Multidimensional 0-1 Knapsacks and various heuristic and exact methods” have been formulated to give response to them.

The traditional knapsack problem and a variation in which an upper bound is forced on the quantity of items that can be chosen are tackled. It is established that suitable combinations of adjusting procedures yield novel and further dominant ways of rounding. Moreover, a “linear-storage polynomial time approximation scheme (PTAS) and a fully polynomial time approximation scheme (FPTAS) that compute an approximate solution, of any fixed accuracy, in linear time” is also presented. These linear complexity bounds offer a considerable progress of the best once known polynomial bounds (Huttler and Mastrolilli, 2006).

Heuristic algorithms experienced in literature that can generally be named as population heuristics include; “genetic algorithms, hybrid genetic algorithms, mimetic algorithms, scatter-search algorithms and bionomic algorithms”. Among these, Genetic Algorithms have risen as a dominant latest search paradigms (Chu and Beasley, 1998).

Genetic Algorithms (GA) are PC algorithms that hunt down fine solutions to a problem from among countless solutions. They are versatile heuristic search algorithm in view of the evolutionary thoughts of natural selection and hereditary qualities. “These computational paradigms were inspired by the mechanics of natural evolution, including survival of the fittest, reproduction, and mutation. This algorithm is an intelligent exploitation of random search used in optimisation problems” (Sinapova Lydia 2014).

Works using genetic algorithms to solve the knapsack problem with inaccurate weight coefficients has been investigated. “The work is based on the assumption that each weight coefficient is imprecise due to decimal truncation or coefficient rough estimation by the decision-maker. To deal with this kind of imprecise data, fuzzy sets provide a powerful tool to model and solve this problem”.

Lin (2008) examined the likelihood of genetic algorithms as a part of taking care of the fuzzy knapsack problem without characterizing participation capacities for each inexact weight coefficient. Lin's proposed approach replicated a fuzzy number by allocating it into several partition points. "A genetic algorithm was used to evolve the values in each partition point so that the final values represented the membership grade of a fuzzy number. The empirical results show that the proposed approach can obtain very good solutions within the given bound of each imprecise weight coefficient than the fuzzy knapsack approach. The fuzzy genetic algorithm concept approach is different, but gave better results than the traditional fuzzy approach" (Lin, 2008).

Works on how to how to demonstrate an accurate determination of parameters and search mechanisms prompted an execution of Genetic Algorithms that yield top notch solutions (Hoff et al., 1985). The methods were "tested on a portfolio of 0/1 multidimensional knapsack problems from literature and a minimum of domain-specific knowledge is used to guide the search process. The quality of the produced results rivals and in some cases surpasses the best solutions obtained by special-purpose methods that have been created to exploit the special structure of these problems".

Bortfeldt and Gehring (2001) presented a hybrid genetic algorithm (GA) for the container packing problem with boxes of unlike sizes and one container for stacking. "Generated stowage plans include several vertical layers each containing several boxes. Within the procedure, stowage plans were represented by complex data structures closely related to the problem. To generate offspring, specific genetic operators were used that are based on an integrated greedy heuristic. The process takes several practical constraints into account. Extensive test calculations including procedures from other authors vouch for the good performance of the GA, above all for problems with strongly heterogeneous boxes".

A general labelling algorithm for locating all non-dominated results of the "multiple objective integer knapsack problems" (MOIKP) was made accessible by Figuera et al. (2009). They presented algorithms for creating four network models, mostly demonstrating the MOIKP.

Their algorithm is based on fathoming the "multiple objective shortest path problems" on a principal network. "Each network is composed of layers and each network algorithm, working forward layer by layer, identifies the set of all permanent non-dominated labels for each layer. The effectiveness of the algorithms is supported with numerical results obtained for randomly generated problems for up to seven objectives while exact algorithms reported in the literature solve the multiple objective binary knapsack problem with up to three objectives. Extensions of the approach to other classes of problems including binary variables, bounded variables, multiple constraints, and time-dependent objective functions are possible". Maya and Dipti (2011) also solved the 0-1 Knapsack Problem (KP) by means of Genetic Algorithms (GAs) in a research project

Modified GA generates outcomes that are better in quality than other driving heuristic (which are for the most part, in view of tabu search) for the KP (Chu and Beasley 1998). However, GA is much slower than other heuristics. Hence, there is a trade-off often seen in or between quality of solution and computer time consumed.

GAs often calls for the creation and assessment of lots of dissimilar children. However, "GAs are capable of generating high-quality solutions to many problems within reasonable computation times". (Beasley and Chu, 1996; Chu and Beasley, 1997, 1998; Chang et al., 2000; Beasley et al., 1999). Additionally, "while performing search in large state-space or multi-modal state-space, or n-dimensional surface, a genetic algorithm offers significant benefits over many other typical search optimisation techniques like linear programming, heuristic, depth-first, breath-first".

Simoes and Costa(2001) performed an empirical study and evaluated the exploits of the "transposition A-based Genetic Algorithm (GA) and the classical GA for solving the 0/1

knapsack problem". Obtained results showed that, just like in the domain of the function optimization, transposition is always superior to crossover. The process of settling on allocations in business procedures to get the most out of profit includes: "collecting profit data for a plurality of classes in the business operation, where each class includes an allocation having a cost function and each allocation belongs to the group consisting of physical allocations and economic allocations; determining profit functions for the allocations from the profit data; formulating a Multiple Choice Knapsack Problem to maximize profit from the profit functions, the cost functions, and a cost constraint; and solving the Multiple choice Knapsack Problem to determine values for the allocations" (European Patent Application EP1350203).

Elhedli (2005) considered a class of "nonlinear Knapsack problems with applications in service systems design and facility location problems with congestion". The author gave two linearization and their individual solution methodologies. The first is calculated specifically utilizing a business solver and the second method provides a piecewise linearization that is explained by "cutting plane Method". The two dimensional Knapsack problem (2KP)

intended at loading a max-profit subset of rectangle picked from a known set into a different rectangle is addressed (Caprara and Monaci 2004). They considered "the natural relaxation of 2KP given by the one dimensional KP with item weights equal to the rectangle areas, proving the worst-case performance of the associated upper bound, and presented and compared computationally four exact algorithms based on the above relaxation, showing their effectiveness".

The objective of the "multi-dimensional knapsack problem" (MKP) is to discover a subset of items with higher value that fulfils various knapsack limitations. For several decades, many researches on methods of solving MKP, heuristic and exact together have been done. Fleszar and Hindi (2009) introduced a number of quick and compelling heuristics for MKP that depend on unravelling the LP relaxation of the problem. Enhancing systems were proposed to fortify

the consequences of these heuristics. Furthermore, the heuristics were kept running with suitable deterministic or randomly generated limitations forced on the linear relaxation that permit creating various superior results. All algorithms were tried on a generally utilized set of benchmark problem examples to demonstrate that they contrasted positively and the best-performing heuristics accessible in literature.

The constrained compartmentalised knapsack problem is an extension of the classical integer constrained knapsack problem which can be stated as the following hypothetical situation: “a climber must load his/her knapsack with a number of items. For each item a weight, a utility value and an upper bound are given. However, the items are of different classes (food, medicine, utensils, etc.) and they have to be loaded in separate compartments inside the knapsack (each compartment is itself a knapsack to be loaded by items from the same class). The compartments have flexible capacities which are lower and upper bounded. Each compartment has a fixed cost to be included inside the knapsack that depends on the class of items chosen to load it and, in addition, each new compartment introduces a fixed loss of capacity of the original knapsack. The constrained compartmentalised knapsack problem consists of determining suitable capacities of each compartment and how these compartments should be loaded, such that the total items inside all compartments does not exceed the upper bound given. The objective is to maximise the total utility value minus the cost of the compartments.” Problems of this nature appear in real world situations such as steel cutting or paper reels. Arenales and Marques (2007) modelled this problem as an “integer non-linear optimisation problem” and designed a heuristic approach and finally examined their approach by using computational experiments.

Computational networks are distributed systems comprising of heterogeneous computing reserves which are conveyed geologically and administratively. These very adaptable systems are intended to meet great computational requests of numerous clients from scientific and business orientations. In any case, there are issues identified with the processing’s resources

which is made out of a grid. Vandester (2008) study the design of a Pan-Canadian grid. The design exploits “the maturing stability of grid deployment toolkits, and introduces novel services for efficiently allocating the grid resources. The changes faced by this grid deployment motivate further exploration in optimizing grid resource allocations. By applying this model to the grid allocation option, it is possible to quantify the relative merits of the various possible scheduling decisions”. With this model, the allocation problem was prepared as a “knapsack problem”. Definition in this way takes into account quick solution times and brings about almost ideal allocations.

E-trade and web-administrations in the territory of web applications particularly in the Last decades have seen exponential development. An imperative characteristic of service metric for web applications is the response time for the client. Web application ordinarily has a multi-level structure and a request may need to cross through every one of the levels before completing its processing. For that reason, a request’s sum response time is the whole of response time at all the levels. Since the normal response time at any level relies on the quantity of servers assigned to this level, a wide range of setups (number of servers designated to every level) can give the same value of service guarantee in terms of aggregate response time. Logically, one might want to discover the design which minimizes the aggregate system cost and fulfils the aggregate response time guarantee. This was modelled as an integer optimization problem (Zhang et al., 2004).

The knapsack problem (KP) is summed up to the situation where items are somewhat requested through a set of priority relations. As in common KPs, every item is connected with benefit and weight, the knapsack has a permanent limit, and the problem is to settle on the set of items to be filled in the knapsack. Be that as it may, every item can be acknowledged just when all the preceding items have been input into the knapsack. The knapsack problem with these extra restrictions is known as “the precedence-constrained knapsack problem (PCKP)” (Samphaiboon and Yamada 2000). To explain PCKP precisely, “a pegging approach, where

the size of the original problem is reduced by applying the Lagrangian relaxation followed by a pegging test" (Yamada and You 2007). By this technique, they found themselves able to fathom PCKPs with a large number of items within a couple of minutes on a common workstation.

A promising solution approach called Meta-RaPS was presented by Moraga et al. (2005) for the 0-1 Multidimensional Knapsack Problem (0-1 MKP). "Meta-RaPS construct feasible solutions at each iteration through the utilization of a priority rule used in a randomized fashion. Four different greedy priority rules are implemented within Meta-RaPS and compared. These rules differ in the way the corresponding pseudo-utility ratios for ranking variables are computed. In addition, two simple local search techniques within Meta-RaPS' improvement stage are implemented. The Meta-RaPS approach is tested on several established test sets, and the solution values are compared to both the optimal values and the results of other 0-1 MKP solution techniques. The Meta-RaPS approach outperformed many other solution methodologies in terms of differences from the optimal value and number of optimal solutions obtained" (Moraga et al. 2005). Test results obtained by Moraga et al. (2005) confirmed that the Meta-RaPS technique is simple to comprehend and simple to

execute hence its ability to attain fine results.

Babaioff, Kleinberg and Immorlica (2007) presented a model for "the multiple-choice secretary problem in which k elements need to be selected and the goal is to maximize the combined value (sum) of the selected elements". The authors also carried out the matriod secretary problem in which the elements of a weighted matriod enter in an irregular request. As every element is monitored, the algorithm settles on an unavoidable choice to pick it or skip it, with the limitation that the selected element must constitute a free set. The goal is to augment the consolidated weight of the selected element. The authors proposed an "integer programming algorithm" for this problem.

Aggarwal and Hartline (2006) also designed tactical auctions program which the competition for revenue is high when the auctioneer is constricted to select managers with private values and with weights openly identified that fit into the knapsack.

According to Chekuri and Hanna (2005) the "multiple Knapsack problems (MKP) is a natural and well-know generalization of the single Knapsack and is defined as follows: Supposing a set m items and n bins (Knapsacks) are given such that each item i has a profit p(i) and a size s(i) and each bin j has a capacity C(i).The goal is to find a subset of items of maximum profit such that they have a feasible packing in the bins". MKP is a unique instance of the generalized assignment problem (GAP) where the benefit and a size of an element can shift taking into account particular container that is allocated to GAPS is APX-hard and a 2approximation, for it is contained in work of Shmoy and Tardos (1990). This approach tends to be a fine approximation for MKP. Shmoys and Tardos (1994) central result for MKP is a "polynomial time approximation scheme (PTAS)". Aside its innate theoretical significance as a typical generalization of the well-studied knapsack and bin packing problems it gives off an impression of being the most grounded exceptional instance of GAP that is not, APX hard They authenticate them by demonstrating that a small simplification of MKP is APX-hard. As a mater of fact, their outcome aided to segregate the border at which request of GAP become APX-hard. An exciting feature of this strategy is PTAS-"Preserving reduction from an arbitrary instance of MKP to an instance with MKP to an instance with 0(log n) distinct sizes and profit". Standard heuristics in operations research, (for example, greedy, tabu search and Simulated

Annealing) takes a shot at enhancing a lone current solution. Residents' heuristics utilize various current solutions and join them together to produce new solutions.

Hybrid approach which consolidates systematic and heuristic schemes was proposed to decrease that incorrectness in the backdrop of a scatter search method. (Gomes da Silva.,

2007). Gomes da Silva stated that “the component of this method is used to determine regions in the decision space to be systematically searched. Comparisons with small and medium size instances solved by exact methods are presented. Large size instances are also considered and the quality of the approximation evaluated taking into accounts the proximity to the upper frontier, devised by the linear relaxation, and the diversity of the solutions”. He compared the performance to other two well-known meta-heuristics. The outcomes demonstrated the viability of the proposed approach for both small/medium and large size cases. A decisive event tabu search strategy which explores both sides of the possibility limit has demonstrated viability in taking care of the multidimensional knapsack problem. This was applied to “the multidimensional knapsack problem with generalized upper bound constraints” (Li and Curry, 2005). Li and Curry (2005) showed the benefits of utilizing surrogate control information versus a Lagrangian relaxation plan as decision standards for the problem class. A limitation normalization technique was introduced to reinforce the surrogate control information and enhance the computational results. The merits of escalating the pursuit at key solutions were additionally illustrated.

Hanafi and freville, (1998) illustrated another way to deal with Tabu Search (TS) emphasising on tactical oscillation and surrogate control information that gives stability between escalation. Heuristic algorithm like Tabu Search and Genetic algorithm have also appeared in recent times for the solution of Knapsack problems. Chu et al. (1998), proposed a genetic algorithm for the multidimensional Knapsack problem.

Employing an approximate solution method based on tabu search, Hifi et al (2002) worked on the Knapsack Sharing Problem (KSP). First, “they described a simple local search in which a depth parameter and a tabu list were used. Next, they enhanced the algorithm by introducing some intensifying strategies.” The two versions of the algorithm produce acceptable result within logical computational time. Broad computational testing on problem examples taken from the literature demonstrated the adequacy of the proposed approach. Eager about making

use of a easy heuristic scheme (simple flip) for answering the knapsack problems, Oppong (2009) offered a study work on the application of usual zero-1 knapsack trouble with a single limitation to determination of television ads at significant time such as prime time news, news adjacencies, breaking news and peak times. Television (television) stations all over the world time table their programmes mixed with adverts or commercials which are the principal sources of income of broadcasting stations. The intention in scheduling classified ads is to attain wider viewers satisfaction and making maximum earnings from the classified ads or adverts. It was once shown that the outcome from the heuristic system compares favourably with the well-known meta-heuristic approaches akin to Genetic Algorithm and Simulation Annealing.

A few present approximation algorithms for the minimization variation of the predicament and a proposed scaling centered on completely polynomial time approximation scheme for the minimal knapsack trouble has been studied. Islam (2009) evaluated the performance of this algorithm with current algorithms. His experiments exhibit that, the suggested algorithm runs fast and has an excellent performance ratio in nature. He additionally conducts extensive experiments on the data furnished through Canadian Pacific Logistics options for the duration of the MITACS internship program. The writer proposed a scaling situated varepsilonapproximation scheme for the multidimensional (d -dimensional) minimal knapsack crisis and checks its efficiency with a generalization of a greedy algorithm for minimum knapsack in d- dimensions. The writer's experimentation showed that the varepsilon-approximation scheme displays excellent efficiency ratio in nature.

Probably the most effective algorithms for fixing the binary-criterion {0,1} knapsack problem are found on the core concept (i.e. based on a small number of principal variables). However this proposal should not be utilized in problems with two or more criterion. Gomes da Silva et al (2008) certified the existence of one of these set of variables in bi-criteria {0,1} knapsack circumstances. Numerical experiments were carried out on five types of {0,1} knapsack situations. The outcomes were made accessible for the supported and nonsupported solutions

as well as for the complete set of efficient solutions. An outline of an approximate and a specified process was additionally presented.

Simulated annealing is a probabilistic strategy for discovering the global minimum of a cost function that may have a numerous local minima. It lives up to expectations by imitating the physical procedure whereby a solid is gradually cooled so that when in the end its structure is "frozen", this happens at a minimum energy configuration. Proposed in Kirkpatrick, Gelett and Vecchi (1983) and Cerny (1985), simulated annealing maintain a temperature variable to create heating process. The temperature is earlier set high and after that allows to gradually "cool" as the algorithm runs. While this temperature variable is high the algorithm will be permitted, with more recurrence, to accept solutions that are more awful than the present solution. This gives the algorithm the capacity to hop out of any local optimums it discovers itself on early on in execution. As the temperature is decreased so is the possibility of tolerating more awful solution, thus permitting the algorithm gradually focusing on a zone of the search space in which ideally, a near ideal solution can be found.

Fubin and Ru (2002) put forward a “Simulated Annealing (SA) algorithm” for the 0/1 “multidimensional knapsack problem”. Problem-specific information is included in the algorithm explanation and assessment of parameters. Keeping in mind the end goal to investigate the abilities of finite-time implementation of SA, computational results demonstrated that SA performs creditable than a genetic algorithm in term of solution time, whilst requiring just a reserved loss of solution quality.

There is a variant of the typical binary knapsack problem termed the “fixed-charge knapsack problem, in which sub-sets of variables (activities) are associated with fixed costs. These costs may represent certain set-ups and/or preparations required for the associated sub-set of activities to be scheduled” (Akinc, 2006). Akinc, discussed problem extensions as well as numerous potential real-world applications. He stated that “the efficient solution of the problem is facilitated by a standard branch-and-bound algorithm based on (1) a non-iterative,

polynomial algorithm to solve the LP relaxation, (2) various heuristic procedures to obtain good candidate solutions by adjusting the LP solution, and (3) powerful rules to peg the variables". Computational experience demonstrates that the proposed branch-and-bound algorithm indicates superb potential in the solution of a wide range of large fixed-charge knapsack problems (Akinc, 2006).

Index selection for relational databases is a significant problem which has been studied quite comprehensively (Gholamian 2007). With index selection algorithms for relational databases from literature, at most one key is regarded as a candidate for every attribute of a relation. "However, it is possible that more than one different type of indexes with different storage space requirements may be present as candidates for an attribute. Also, it may not be possible to eliminate locally all but one of the candidate indexes for an attribute due to different benefits and storage space requirements associated with the candidates. Thus, the algorithms available in the literature for optimal index selection may not be used when there are multiple candidates for each attribute and there is a need for a global optimization algorithm in which at most one index can be selected from a set of candidate indexes for an attribute". Gholamian (2007) presentation on the multiple choice index selection problems showed that it is NP-hard, and offers an algorithm which provides roughly an optimal solution within a client specified error bound in a logarithmic time order.

Another critical issue in almost all sectors is the allocation of resources among different activities which has lead to a multitude of research on this topic. Work on non-linear Knapsack problems Zoltners and Sinha (1975) provided a review of a conceptual framework for sealed resources allocation modelling. They developed general model for sales resource allocation which simultaneously account for multiple sales resources, multiple items period and carry over several actual applications of the model in practice, which illustrates the practical value of their integer programming models.

The problem of resource allocation among different activities such as allocating a marketing budget among sales territories is analyzed by Luss and Gupta (1980). The authors assumed that the return function for each territory uses different parameters and derives single-pass algorithms for different concave pay off functions based on the Karush-Kuhn-Tuncker (KKT) condition in order to maximize total returns for a given amount of effort.

Carlo Vercellis (1994) argues that a "Lagrangean decomposition technique for solving multiproject planning problems with resource constraints" tends out to be useful and therefore described an alternative mode of executing every activity in the project. The disintegration can be valuable in a number of ways; from a side, "it provided bounds on the optimum, so that the quality of approximate solutions can be evaluated. Furthermore, in the context of branch and bound algorithms, it can be used for more effective fathoming of the tree nodes.

Finally, in the modelling perspective, the Lagrangean optimal multipliers can provide insights to project managers as prices for assigning the resources to different projects".

Allotment of resources under uncertain conditions is an exceptionally regular problem in some genuine situations. Managers need to choose whether or not to contract applicants, not knowing whether future competitors will be more grounded or more alluring. Machines need to choose whether to acknowledge employments without information of the significance or profitability of future occupations. Counselling organizations must choose which occupations to tackle, not knowing the income and resource connected with potential future request (Owusu-Bempah, 2013).

All the more as of late, online auctions have turned out to be an imperative resource allocation issue. Promoting auctions specifically give the main source of monetization for a an array of web administrations including web crawlers, online journals, and social networking sites. Furthermore, they are the primary source of client acquisition for a wide exhibit of small online business, of the networked world. In offering for the privilege to show up on a website page, (for example, an internet searcher), sponsors need to exchange off between huge quantities of

parameters, including essential words and viewer attributes. In this situation, a publicist may have the capacity to assess precisely the offer required to win a particular auction, and benefit either in direct income or name recognition to be picked up, yet may not think about the trade off for upcoming auctions. These problems include an online situation, where an algorithm needs to settle on choices on whether to acknowledge an offer, based exclusively on the required resource investment (or weight) and anticipated estimation of the present offer, with the aggregate weight of all chosen offer not surpassing a given spending plan. Moshe et al (2008) studied this model as a knapsack problem.

When the weights are even and equivalent to the weight limitation, the problems above tends to represent the popular secretary problem which was first introduced by Dynkin (1963).

Many theoretical studies of Knapsack problems have been intended and applied to real life problems. far more than a decade now in computer science, Knapsack problem has been widely studied.

Many that were mostly application oriented made researchers and practitioners look for better and faster solutions to cope with vast industrial and financial management problem. There are a lot of variations of the problem but the zero-one maximum knapsack in single dimension is the simplest. Loads of industrialized problems can be stated as Knapsack problem examples includes spending plan control Payload stacking, cutting stock, project election etc.

CHAPTER THREE

METHODOLOGY

3.1.0 Introduction

In this chapter the study deals with the methodology. The fundamental theory of Dynamic Programming with regards to its definition and formulation, component, objectives and the method of analysis of data to arrive at the objective will be discussed in this chapter

3.1.1 Terminologies in Dynamic Programming

Every Dynamic model consists of a set of decision variable which represents the decisions to be made; a contrast to a problem data, where values are either given or can be simply calculated from what is given.

3.1.2 Decision Variables

Decision variables portray the amounts that the decision makers might want to focus. They are the questions of a scientific programming model. Normally, it ideal qualities with an optimization strategy can be resolved. In a general model, decision variables are given algebraic assignments, for example, $x1,x2,x3,\ldots \ldots .xn$. The quantity of decision variables is n, and xj is the name of the jth variable. In some particular circumstance, it won't be strange to utilize different names, for example, xij or yk or zij. An allocation of values to all variables in a problem is known as a solution.

3.1.3 Objective Function

The objective function assesses some quantitative model of direct significance, for example, expense, benefit, utility, or yield.

The general linear objective function can be composed as

$$z = c1x1 + c2x2 + \cdots + cnxn = \sum_{j=1}^{n} c_j\, x_j$$

Here cj is the coefficient of the jth choice variable. The condition chosen can be either augmented or minimized. Here Z can be described as the immediate importance where as c_j is the weight and x_j is the profit of the item.

3.1.4 Parameters

The collection of coefficients (z, c, x_j, v_i, w_i) for all values of the lists i and j are known as the model's parameters. For the model to be totally decided all parameter values must be stated.

3.1.5 Non-Negativity Restrictions

For the most part handy problems are necessitated to be nonnegative variables; $x_j \geq 0, for$ $j=1...n.$This exceptional kind of restriction is known as a non-negativity restriction. Once in a while variables are required to be non-positive or, actually, may be unlimited (permitting any real value).

3.1.6 Knapsack Algorithm

The knapsack problem is a standout amongst the most concentrated on issues in combinatorial optimization, with some genuine applications. Thus, numerous unique cases and generalizations have been analyzed. The Knapsack problem is a traditional problem with a single limitation. Various types of Knapsack Problems happen, contingent upon the items distribution and knapsack and additionally partly because of their extensive variety of applicability. Variations includes the "0 – 1 Knapsack Problem, Bounded Knapsack Problem,

Multiple-choice Knapsack Problem, Multiple Knapsack Problem, Multi-constrained Knapsack Problem, Integer, linear and non-linear Knapsacks, deterministic and stochastic Knapsacks," multidimensional Knapsack and so on.

This thesis solve the variation of binary 0 – 1 knapsack problem

3.1.7 The 0-1 Knapsack Problem

This is a clean integer programming with a single check which forms an essential class of whole number programming. It confines the number xi of duplicates of every sort of item to zero or one and the relating aggregate is boosted without having the data size total to surpass the limit C. The 0-1 Knapsack Problem (KP) can be mathematically stated through the succeeding integer linear programming.

"Let there be *n* items, z_1 to $z_n$where z_j has a value p_j and data size w_j. x_j is the number of copies of the item z_j, which, must be zero or one. The maximum data size that we can carry in the bag is C. It is common to assume that all values and data sizes are nonnegative." To make simpler the illustration, we also presume that the items are scheduled in increasing order of data size

$$maximize \sum_{j=1}^{n} p_j x_j \quad \text{_____(3.1)}$$

$$subject\ to \sum_{j=1}^{n} (w_j x_j) \leq C \quad \text{_____(3.2)}$$

x_j = 0 or 1, j = 1,...,n

Increase the summation of the items values in the knapsack so that the addition of the data sizes must be not exactly or equivalent to the knapsack's limit.

3.1.8 Existing Heuristics of Optimising Memory

Without loss of generality, authors, programmers and proponents of PCs attempt to answer the subsequent questions: “which kind of application data should be loaded and to which kind of memory?” With a specific end goal to take care of this question, data placement could be conducted on one hand by the components of the considered memory (access speed, energy cost, huge number of miss access cases, and so forth.), by profiling benchmarks (number of times that data is accessed, data size, access frequency, etc.) and then again by the data gathered either analyzing statistically the benchmark’s code or profiling benchmarks dynamically (“number of times that data is accessed, data size, access frequency, and so on.”). Because of the restricted size of memory spaces, one needs to attempt to ideally assign data in it all together not to end up running low memory or squashes of some PC applications. In this perspective, the greater part of the creators, software engineers and advocates of computers tend to utilize one of these three following strategies.

Load data into memory by size: all smaller data/process are stacked into memory allotment space as there is space accessible else held in RAM.

Load data into memory by number of accesses: the most repeatedly accessed/used data are apportioned memory space as there is space accessible else held in RAM.

Load data into memory by number of accesses and size: this is in some way a blend of the two earlier strategies. The goal here is to join their merits. In the event that we consider the example of a structure in which just a section is the most frequently accessed/used, we consider average number of access to this structure. This maintains a strategic distance from granularity issues. Here, data are sorted by proportion (access number/size) in downward order. The data with the most elevated proportion is stacked first into memory space as there is space a accessible else it is held in RAM.

In this thesis, the third strategy is referred to as the basis for our memory optimizations. The problem that this strategies listed try to solve is a problem of combinatorial optimization

which represent the old fashioned but popular knapsack problem. Assuming data are items and memory is a huge knapsack. We wish to load this knapsack that can hold an aggregate weight of W with some items combination from a record of N probable item each with weight $w_i$and value v_i so that the items value loaded into the knapsack is augmented. A crucial investigation of this problem demonstrates that this problem has a sole linear limitation, a linear objective function which totals the items value in the knapsack, and the included constraint that every item will be in the knapsack or not - representing the legendary binary knapsack problem.

As earlier indicated, this research paper makes use of the dynamic programming algorithm to investigate the problem.

As "Knapsack Problems are NP-hard" there is no recognized exact solution technique than possibly a greedy approach or a possibly complete enumeration of the solution space. However quite a lot of effort may be saved by using one of the following techniques: These are "Branch-and-Bound and dynamic programming" methods as well as meta-heuristics approaches such as "simulated annealing, Genetic algorithm, and Tabu search" which have been employed in the case of large scale problems solution.

3.2.0 Dynamic Programming

This is an approach for responding to an unpredictable problem by reducing it into a set of simpler sub-problems It is appropriate to problems displaying the properties of overlying subproblems and optimal substructure. Dynamic Programming (DP) is an effective procedure that permits one to take care of a wide range of sorts of problems in time O(n^2) or O(n^3) for which an innocent methodology would take exponential time. Dynamic Programming is a general way to deal with a sequence of interrelated choices in an optimum way. This is a general approach to taking care of problems, much like "divide-and-conquer" aside from that unlike divide-and-conquer, the sub-problems will normally overlap. Most in a general sense,

the approach is recursive, similar to a PC schedule that calls itself, adding data to a stack every time, until certain ceasing conditions are met. Once ceased, the solution is solved by expelling data from the stack in the best possible sequence.

With a specific end goal to take care of a given problem, utilizing a dynamic programming method, we have to tackle different parts of the problem (sub-problems), and after that consolidate the results of the sub problems to achieve a general solution. Regularly when utilizing a more guileless approach, a considerable lot of the sub-problems are created and solved many times. The dynamic programming methodology tries to take care of every subproblem just once, therefore diminishing the quantity of calculations: once the answer for a given sub-problem has been registered, it is kept or "memoized": whenever the same solution is required, it is basically looked up. This methodology is particularly valuable when the quantity of rehashing sub-problems develops exponentially as a size's function of the input. Dynamic programming algorithms are used for optimization (for instance, discovering the most limited way between two ways, or the speediest approach to multiply numerous matrices). A dynamic programming algorithm will look at the earlier tackled sub-problems and will consolidate their answers for give the best answer for the given problem.

3.2.1 The basic idea of Dynamic Programming

Basic Idea (version 1): Given the problem, by one means or another separate it into a sensible number of sub-problems (where "sensible" may be somewhat like n^2) in a manner that the ideal solution for the smaller sub-problems can be utilized to give ideal solutions for the bigger ones.

Basic Idea (version 2): Assume a recursive algorithm for a few problem gives a truly terrible recurrence like T(n) = 2T (n − 1) + n. In any case, assume that a large portion of the subproblems you achieve as you go down the recursion tree are the same. At that point you would like to get a major savings if the calculation is kept so that each distinctive subproblem is processed just

once. The solution can be kept in an array or hash table. This perspective of Dynamic Programming is regularly called memoizing.

A description of a lot more important characteristics of dynamic programming is illustrated next

- *The problem can be separated into stages:* In some dynamic programming applications, the stages are related to time, hence the name dynamic programming. These are often dynamic control problems, and for reasons of efficiency, the stages are often solved backwards in time, i.e. form a point in the future back towards the present. This is because the paths that lead from the present state to the future goal state are always just a subset of all the paths leading forward from the current state. Hence it is more efficient to work backwards along this subset of paths.
- *Every stage has quantity of states:* Most usually, in solving a small problem at a stage this is the information needed.
- *The choice at a stage informs the condition at the stage into the condition for the next stage.*
- *Given the present state, the optimal choice for the outstanding stages is independent of choices made in earlier states:* This is the primary dynamic programming rule of optimality. It implies that it is alright to break the problem into smaller pieces and fathom them separately.
- *There is a recursive connection linking the value of choice at a stage and the value of the optimum choices at earlier stages*: In other words, the optimum choice at this stage utilizes the already discovered optima. In a recursive connection, a function emerges on both sides of the equation.

3.2.1 Outline of Dynamic Programming

1. Define a small piece of the entire problem and locate an ideal solution for this small part.

2. Enlarge this small part a little and locate the ideal solution for the new problem utilizing earlier found ideal solution
3. Keep on with Step 2 until you have discovered that the present problem incorporates the first problem. At the point when this problem is understood, the stopping conditions will have been met.
4. Track back the answer for the entire problem from the ideal solutions to the small problems to unravel along the way

3.3 Implementation of the 0-1 Kp Using Dynamic Programming

Recall knapsack Problem

Formal description: Given two n-tuples of positive numbers

$(v_1, v_2 \dots v_n) and\ (w_1, w_2 \dots w_n)\ and\ C > 0$ we wish to determine the subset $T \subseteq$

$\{1, 2, \dots n\}$ that

$$maximize \sum_{i \in T} v_i \quad \text{____(3.3)}$$

$$while \sum_{i \in T} w_i \leq C \quad \text{_____(3.4)}$$

The problem identified is an "optimization problem" which can be solved using Dynamic programming. The thought is to register the answers for the subsub-problem once and store the solution in a table, so they can be reclaimed (more than once) later.

3.3.1 The Idea of Developing a DP Algorithm

1. **Structure:** Distinguish the structure of an ideal solution.

   Break down the problem into smaller problems, and discover a connection between the structure of the ideal solution of the first problem and the solution of the smaller problems.
2. **Principle of Optimality**: Recursively identify the value of an ideal solution. Express the solution of the first's problem as ideal solutions for smaller problems.

3. **Bottom-up calculation**: Work out the value of the best solution in a base up style by utilizing a table structure.
4. **Construction of ideal/best solution**: create an ideal/best solution from processed data

Steps 1-3 outline the premise of a dynamic-programming response for a problem.

3.3.2 Dynamic Programming Algorithm for Knapsack:

A set of *n* items is given where i = *item* z = the storage limit.

w_i = size or data weight

v_i = profit

C = maximum capacity (size of the knapsack)

The target is to locate the subset of items of maximum sum value so much that the total of their sizes is at most C ("they all fit into the knapsack").

Step 1: Break up the problem into smaller problems.

Constructing an array V[0. . . n, 0 . . .C].

For $1 \leq i \leq n$ and $0 \leq z \leq C$, the entry V[i ,z] will keep the maximum (combined) computing value of every subset of items {1,2, . . ., i} of (combined) size largely z.

If we calculate all the entries of this array, then the array entry V[n, C] will hold the highest computing items value that can fit into storage, that is, the solution to the problem.

Step 2: Recursively describe the worth of an ideal solution in terms of solutions to smaller problems.

Initial settings: Set

$$V[0, z] = 0 \text{ for } 0 \leq z \leq C, \text{ no item}$$

$$V[0, z] = -\infty \text{ for } z < 0, \text{ illegal}$$

Recursive step: Use

$$V[i, z] = max(V[i-1, z], v_i + V[i-1, z-w_i])$$

for $1 \leq i \leq n, 0 \leq z \leq C$,

Remember: V[i, z] stores the maximum (combined) computing value of any subset of items {1,2, . . . , i} of (combined) size at most z, and item i has size, w_i (units) and has a value v_i, C (units) is the maximum storage.

To compute V[i , z] there are only two choices for item i:

Leave item i from the subset: The best that can be done with items {1, 2, . . . i - 1} and storage limit z is V[i – 1, z].

Take item i (only possible if $w_i \leq C$): This way we gain v_i benefit, but have spent w_i bytes of the storage. The best that can be done with the remaining items {1, 2, . . ., i – 1} and storage limit, z - w_i is V[i – 1, z - w_i].

End product is v_i + V[i – 1, z - w_i].

If w_i> z, then v_i + V[i – 1, z - w_i] = $-\infty$

Step 3: Using Bottom up computing V[i, z]

Bottom: V[0, z] = 0 for all $0 \leq z \leq C$.

Bottom-up computation:

$V[1, z] = max \ \ (V[i-1, z], v_i + V[i-1, z-w_i])$ *row by row.*

V[i, z]	z = 0	1	2	3	. . .	. . .	C
i = 0	0	0	0	0	…	. . .	0
1							
2							
⋮							
N							

bottom

up

Table 3.1 Bottom up computation

The dynamic programming pseudo code is as follows:

"// Input:

// Values (stored in array v)

// Data sizes (stored in array w)

// Number of distinct items (n)

// Knapsack capacity (C) // Storage limit (z) for j from 0 to C do m[0, z] := 0 end for for i from 1 to n do for z from 0 to C do if w[i] <= j then m[i, z] := max(m[i-1, z], m[i-1, z-w[i]] + v[i]) else m[i, z] := m[i-1, z] end if end for end for "

An extended variation of the pseudo code of the Dynamic programming is illustrated below.

"// Recursive algorithm: either we use the last element or we don't. Value (n, C)

// z = space left or storage limit,

// n= # items still to choose from,

// i= # items,

// w_n = size or data size of item

// Knapsack capacity (C)

{ if (n == 0) return 0; if (w_n > z) result = Value(n-1,z);

// can't use nth item else result = max{ Value(n-1, z), v_n + Value(n-1, z-w_n), }; return result;

}"

From the equation, this takes exponential time. However, there are simply O(nC) dissimilar couples of values the arguments can probably take on, hence ideal for "memoizing".

"Value(n,C)

{ if (n == 0) return 0; if (arr[n][z] != unknown) return arr[n][z]; // <- added this if (w_n > z) result = Value(n-1,z); else result = max{ Value(n-1, w) , v_n + Value(n-1, z-w_n) };

arr[n][z] = result; // <- and this

return result;

}"

Given that any known couple of arguments to Value can go through the array test just once, and in doing so generates at most two recursive calls, we have at most 2n(C +1) recursive calls summation, and the sum time is O(nC).

The above discussion results in computing the value of the optimal solution. To obtain the actually items involved in the computing of the value of the optimal solution, it is work backwards: if arr[n][z] = arr[n-1][z] then we do not use the *n*th item so we just recursively work backwards from arr[n-1][z]. Otherwise, we did use that item, so we just output the *n*th item and recursively work backwards from arr[n-1][z-w_ n].

3.3.3 A simple Test Case

Imagine a number of data in a queue, all waiting to be allocated memory space with limited memory capacity. We want to maximize the number of allocated processes ensuring that there is optimal utilisation of memory without exceeding the memory capacity. The data/processes are labelled A through G. Which data/process should be loaded into memory? In a bid to respond to this problem, information is gathered either statistically by examining the benchmark's code or dynamically by benchmarks profiling ("number of times that data is accessed, data size, access frequency," etc.). Because of the limited size of memory spaces, one has to try to optimally allot data in a bid not to end up running low memory or crushes of some computer data. The dynamic profiling benchmark features of "number of times that data is accessed and data size" is adopted and used as parameters to implement the Dynamic programming method.

Thus each data has a "value" (Number of times data is accessed) and a "data size" (Size) in bytes. For instance, assuming the values and size for this case are:

Data	1	2	3	4
Value (Number of times data is accessed), v_i	3	4	5	6

Size (Data size), w_i in kb	2	3	4	5

Table 3.2 Example of Bottom up computation

Let say the computer memory space (capacity of Knapsack, C) is 5 kilobytes. Which data(es) should be loaded into memory?

The knapsack problem can be solved in exponential time by attempting all probable subsets. With Dynamic Programming, the problem can be reduced to time O(nC).

This is done top down by beginning with a simple recursive solution and after that attempting to "memoize" it. It is started by simply working out the best possible sum value, and then the actual extract of the items needed can be seen. Given a set of *n* items, let i = *item* z = the storage limit.

w_i = size or data weight

v_i = profit

C = maximum capacity (size of the knapsack)

Data (data size, value) = {(2, 3), (3, 4), (4, 5), (5, 6)}.

Initial settings

$V[0, z] = 0$ for $0 \leq z \leq C$, (1)

$V[0, z] = -\infty$ for $z < 0$, (2)

$V[i, z] = max\ (V[i-1, z], v_i + V[i-1, z-w_i])\ row\ by\ row$ (3)

V[i, z]	z = 0	1	2	3	4	5
i = 0	0	0	0	0	0	0

Table 3.3 Solution space i = 0

This is true because of (1)

The i = 1 case {(2, 3), (3, 4), (4, 5), (5, 6)}.

Using item (2, 3), compute the values of row 3

v_i = 3, w_i = 2, i = 1;

$V[i, z] = max\ (V[i-1, z], v_i + V[i-1, z-w_i])$

V[1,*0*] = max (V[0, 0], 3+ V[0, 0-*2*] = max(0, $-\infty$) = 0

V[1,*1*] = max (V[0, 1], 3+ V[0, 1-*2*] = max(0, $-\infty$) = 0

V[1,*2*] = max (V[0, 2], 3+ V[0, 2-*2*] = max(0, 3) = 3 V[1,*3*]

= max (V[0, 3], 3+ V[0, 3-*2*] = max(0, 3) = 3

V[1,*4*] = max (V[0, 4], 3+ V[0, 4-*2*] = max(0, 3) = 3

V[1,*5*] = max (V[0, 5], 3+ V[0, 5-*2*] = max(0, 3) = 3

V[i, z]	z = 0	1	2	3	4	5
i = 0	0	0	0	0	0	0
1	0	0	3	3	3	3

Table 3.4: computation of i = 1 case solution

The *i* = 2 case {(2, 3), (3, 4), (4, 5), (5, 6)}.

Using item (3, 4), compute the values of row 4 , v_i = 4, w_i = 3, 1 = 2;

$V[i, z] = max\ (V[i-1, z], v_i + V[i-1, z-w_i])$

V[2,*0*] = max (V[1, 0], 4+ V[1, 0 - *3*] = max(0, $-\infty$) = 0

V[2,*1*] = max (V[1, 1], 4+ V[1, 1- *3*] = max(0, $-\infty$) = 0

V[2,*2*] = max (V[1, 2], 4+ V[1, 2- *3*] = max(3, $-\infty$) = 3

V[2,*3*] = max (V[1, 3], 4+ V[1, 3- *3*] = max(3, 4) = 4

V[2,*4*] = max (V[1, 4], 4+ V[1, 4- *3*] = max(3, 4) = 4

V[2,*5*] = max (V[1, 5], 4+ V[1, 5- *3*] = max(3, 4+3) = 7

V[i, z]	z = 0	1	2	3	4	5
i = 0	0	0	0	0	0	0
1	0	0	3	3	3	3

2	0	0	3	4	4	7

Table 3.5: computation of i = 2 case solution

The i = 3 case

Using item (4, 5), compute the values of row 5, v_i = 5, w_i = 4, 1 = 3;

$V[i, z] = max\ (V[i-1, z], v_i + V[i-1, z-w_i])$

V[3,*0*] = max (V[2, 0], 5+ V[2, 0 - *4*] = max(0, $-\infty$) = 0

V[3,*1*] = max (V[2, 1], 5+ V[2, 1 - *4*] = max(0, $-\infty$) = 0

V[3,*2*] = max (V[2, 2], 5+ V[2, 2- *4*] = max(3, $-\infty$) = 3

V[3,*3*] = max (V[2, 3], 5+ V[2, 3 - *4*] = max(4, $-\infty$) = 4

V[3,*4*] = max (V[2, 4], 5+ V[2, 4 - *4*] = max(4, $0 + 5$) = 5

V[3,*5*] = max (V[2, 5], 5+ V[2, 5 - *4*] = max(7, $0 + 5$) = 7

V[i, z]	z = 0	1	2	3	4	5
i = 0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7

Table 3.6 computation of i = 3 case solution

The i = 4 case

Using item (5, 6), compute the values of row 6

v_i = 6, w_i = 5, 1 = 4;

$V[i, z] = max\ (V[i-1, z], v_i + V[i-1, z-w_i])$

V[4,*0*] = max (V[3, 0], 6+ V[3, 0 - *5*] = max(0, $-\infty$) = 0

V[4,*1*] = max (V[3, 1], 6+ V[3, 1 - *5*] = max(0, $-\infty$) = 0

V[4,*2*] = max (V[3, 2], 6+ V[3, 2 - *5*] = max(3, −∞) = 3

V[4,*3*] = max (V[3, 3], 6+ V[3, 3 - *5*] = max(4, −∞) = 4

V[4,*4*] = max (V[3, 4], 6+ V[3, 4 - *5*] = max(5, −∞) = 5

V[4,*5*] = max (V[3, 5], 6+ V[3, 5 - *5*] = max(7, 6) = 7

V[i, z]	z = 0	1	2	3	4	5
i = 0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

Table 3.7 computation of i = 4 case solution

To find the particular items to be included in the knapsack that makes the optimal subset,

Let $i = n$ and $k = z$ if $V[i, k] \neq V[i\text{-}1, k]$ then, mark the ith item to be included in the knapsack

$i = i\text{-}1$, $k = k - w_i$, else $i = i\text{-}1$

From the completed solution table 3.7

Let $i = 4$, k = 5, $v_i = 6$, $w_i = 5$

V[i,k] and $V[i\text{-}1, k]$

V[4,5] = 7 and $V[4\text{-}1, 5]$ =7

Since V[i, k] = V[i – 1, k], item 4 should not be included in the knapsack.

V[i, z]	z = 0	1	2	3	4	5
i = 0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

Table 3.8: Finding the optimal subset. i = 3 case

Consider, i=3, k = 5, vi = 6 and wi = 5

V[i, k] and V[i-1, k]

V[3, 5] = 7 , V[3-1, 5] = 7

Since V[i, k] = V[i-1, k], so item 3 cannot be part of the knapsack

Consider, i = 2, k = 5, vi = 4, and wi = 3

V[i, z]	z = 0	1	2	3	4	5
i = 0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

Table 3.9: Finding the optimal subset. i = 2 case

V[*i*, *k*] and V[*i*-1, k]

V[2, *5*] = 7 and V[*2*-1, *5*] = 3.

Since V[*i*, *k*] ≠ V[*i*-1,*k*], then the item 2 should be included in the knapsack *k* – w_i = 2, Now consider, *i* = 1, *k* = 2, v_i = 3 and w_i =2

V[i, z]	z = 0	1	2	3	4	5
i = 0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

Table 3.10: Finding the optimal subset. i = 1 case

V[*i*, *k*] , *V[i*-1,*k*]

V[1, 2] = 3 and V[1-1, 2] = 0

Since V[*i*, *k*] ≠ V[*i*-1,*k*], then item 1 can be part of the knapsack.

k – w_i = 0, Again, consider *i* = 0, and *k* = 0
This will result in an invalid answer.

V[i, z]	z = 0	1	2	3	4	5
i = 0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

Table 4.11: Selected item of the optimal solution

Consequently, the optimal solution is 7 which is arrived at by choosing items with number, i = {1, 2}

From table 3.2, Item 1 has a value 3, a data size of 2 and that of item 2 has a value 4 and data size of 3. Together their value will be 7 and data size is 5 equalling the total capacity of the knapsack.

3.3.4 Computerised Solution

Knapsack Algorithm Finding the Optimal Subset

Input: v[1…n] value (Number of times data is accessed), w[1…n] storage vector or size, n number of data, C max capacity and z storage limit.

Output: V[n, C]: the maximum computing value of any subset of data {1, 2, . . ., n} of (combined) size at most C

"Function Knapsack(v, z, n, C)
1: for z = 0 to C do
2: V[0, z] ⟵ 0
3: end for
4: for i = 1 to n do
5: for z = 0 to C do
6: if w[i] ≤ *z then*
7: V[i , z] = max (V[i – 1, z], v[i] + V[i – 1, z - w[i]])
8: keep[i ,z] = true # ith item (data) is selected in V[i , z]
9: else
10: V[i , z] = V[i – 1, z]
11: keep [i ,z] = false # ith item (data) is NOT selected in V[i ,z]
12: end if
13: end for
14: end for
15: $K \leftarrow C$
16: for i = n down to 1 do
17: if keep[i ,K] = true then
18: output i

19: $K \leftarrow C - w[i]$

20: end if

21: end for 22:

return V[n,C]

end function"

The above Algorithm stores record of the subset of items that result in the optimal solution. To calculate the actual subset, extra supplementary Boolean array *keep*[i , z] which is true if we come to a decision to pick the ith data in V[i , z] and false otherwise. The value of the storage limit z, is given to Knapsack, k to store.

Ultimately it is prudent to note that dynamic programming, though mind-numbing to carry to complete by hand, is very effective contrasted with a brute force listing of every promising combination to locate the optimum one.

3.4 Importance of Dynamic Programming

Dynamic programming is an optimization technique and obtains solutions from a set of options of individual elements. It enables one to develop sub solutions of a large program whiles ensuring that the sub solutions are easier to maintain, use and debug. Additionally, it possesses overlapping that means they can be reuse. These sub solutions are optimal solutions for the problem.

Dynamic programming calculates its solution (bottom up) by combining them from smaller sub-solutions, and by attempting numerous potential outcomes and decisions before it lands at the ideal/best set of decisions. It moreover, stores its past values to keep away from multiple computations

There is a litmus test for Dynamic Programming, called The Principle of Optimality. Dynamic programming can be connected to any issue that examine the principle of optimality. Generally expressed, "partial solutions can be optimally extended with regard to the state after the partial solution instead of the partial solution itself". A problem is said to

fulfil the Principle of Optimality if the sub-solution of the finest solution of the problem are themselves ideal solution for their sub-problems. For instance, to choose whether to broaden an estimate string matching by a substitution, insertion, or deletion, we don't require to know precisely which succession of operations was performed to date. The fact is, there may be a lot of distinctive edit sequence that accomplish a cost of C on the first p characters of pattern P and t characters of string T. Future choices will be made in light of the outcomes of past choices, not the actual choices themselves.

CHAPTER FOUR

DATA IMPLEMENTATION AND ANALYSIS

4.0 Introduction

In this chapter, we shall consider a computational study of dynamic programming applied to knapsack instance. Consideration is given to the 0-1 or binary knapsack problem (KP) where given a set of n items and a knapsack with

p_j = value of item j,

In this research work, we name p_j *Number of times data is accessed of item j and Item as Data*

$w_j = weight$ of item j,

In this research work, data size of item j is memory size

$C =$ capacity of the knapsack,

The problem is to select a subset of the data/process whose total memory space does not exceed the knapsack capacity C, and whose total value, *in this case* Number of data access is at maximum.

Thus select a subset of the items so as to

$$Maximize \sum_{j=1}^{n} p_j x_j \quad \text{_____(4.1)}$$

$$subject\ to \sum_{j=1}^{n} (w_j x_j) \leq C \quad \text{_____(4.2)}$$

x_j = 0 or 1, j ∈ $N = \{1, \dots, n\}$,

Where $x_j = \begin{cases} 1 & if\ item\ j\ is\ selected; \\ 0 & Otherwise \end{cases}$

Without loss of generality it is assumed that all input data are positive integers.
The objective function (equation 4.1) is to find a subset of the possible items (i.e. the vectors of items); where the sum of the Number of data access of these items is maximised, according to constraints presented in equation 4.2. Equation 4.2 states, that the sum of the memory size (relative-weights) of the vector of items chosen is not to be greater than the capacity of the knapsack. Equation 3 refers to the notion that we wish to generate a vector of items, of size n (j = 1, . . . , n items), whereby a 0 at the i^{th} index indicates that this item is not in the chosen subset and a 1 indicates that it is.

In modern computers many processes run at once. All program that is executing on the computer, either noticed or unnoticed, consume some computer resources. But before a data/process runs it needs to be created and then given all the resources accessible that it requests to run (i.e. set to ready queue in the ready state) before the CPU work on this process's

instructions – run state. A current created program is place in the ready queue. Data/processes stays in ready queue for CPU allocation. Active processes are placed in a run queue and load them into memory for execution. Multiprogramming operating system permits a lot more process to be carried into the executable memory at one time and process that are loaded use the CPU by means of some scheduling algorithm.

For each Process that wants to be allocated space to run, it carries CPU scheduling information and Memory Management information such as Number of times data is accessed information and memory usage respectively to be allocated to page tables or segment tables. When CPU is ready to execute a Process, it picks up the data item from the run queue based on the system information it carries: in this case, the Number of times data is accessed and checks if it has available memory space for that Process.

In this research work the system information a process carries is implemented using integers within a fixed range, with 10 being the highest possible Number of times data is accessed value/figure.

4.1 Data Collection

As already indicated all created Process that wants to be allocated space to run, carries resources - CPU scheduling information and Memory Management information. These resources are gathered either statistically by examining the benchmark's code or dynamically by benchmarks profiling ("number of times that data is accessed, data size, access frequency," etc.).

This research work uses the data collected dynamically from a process – "number of times data is accessed and data size" to represent the value and weight of an item by assigning randomized personal data - to model the knapsack problem and solve the problem of Computer systems ending up running low memory by employing a dynamic programming approach.

Category A: The table below shows a computer system with a total of 15 created processes, all with their system information in figures. The computer memory can accommodate capacity of 32mb.

Processes No	data size/mb	Number of times data is accessed
1	6	6
2	4	5
3	8	8
4	6	2
5	6	9
6	7	4
7	5	7
8	7	9
9	3	6
10	10	2
11	3	9
12	6	10
13	9	9
14	5	8
15	3	6
	Total 88	

Table 4.1: Data values of Category A

Source: *Authors survey*

Category B: The table below shows system information of a total of 32 data/processes in figures. The memory capacity of this system is 512mb.

Processes No	Memory data size/mb	Number of times data is accessed
1	25	6
2	52	5
3	100	7
4	86	9
5	36	5
6	76	3
7	12	4
8	56	7
9	128	7
10	96	7
11	160	8
12	120	3
13	82	2
14	68	4
15	92	6
16	48	5
17	128	4

18	160	8
19	24	6
20	64	2
21	96	1
22	124	3
23	65	2
24	12	3
25	45	7
26	56	6
27	86	7
28	82	3
29	98	7
30	134	8
31	142	4
32	200	5
	Total 2753	

Table 4.2: Data values of Category B

Source: *Authors survey*

4.2 Data Implementation

The system's main goal is to pick from a pool of data (processes), some Process that should be loaded into memory for system execution by the CPU taking into account some notable constraints.

4.2.1 Graphical User Interface

The user interface show the parts of the Program which communicate with the user directly. It introduces a warm page with a file menu. The file menu introduces the main program, the ProcessMaster. ProcessMaster has two tabs are available: PCalculate and intermediate output. A tabbed pane was used to allow for a general interface to contain the changing panel. The user selects which component to see by choosing the tab matching to the required component.

4.2.2 PCalcuate Panel

The Program has five sections.

The Start section: user input the total number of items or process. Be it as it may for the system utilisation to be calculated, the system needs to take certain parameters as inputs from the user, and execute validity checks to ensure that the input taken is valid

- Capacity of the system: which will determine the limit of the memory space of processes
- Number of Processes: which identify the number of items that needs to be entered into the Program. It have to be entered as a non-negative number.

The Data entry section: where user input the various process memory usage and "Number of times data is accessed". The user click on the Start Algorithm button to initiate this process. It displays a table of two items:

- Memory – Enter memory usage of each Process. Have to be input as a non-negative digit
- Number of times data is accessed: Enter the Number of times data is accessed of each Process. Have to be input as a non-negative digit
- The compute button: Once these parameters have been entered and verified, a click on this button loads the output section.

The Save to File button: Data entered into the program can be saved for later reference and usage. The system allows files to be saved only in xls/xlsx format.

System input section: where user can generate random values to test the system or browse for a file which may have input data for the Program or from a saved file. Two main buttons are found here:

Browse for File: Data can be loaded directly from an excel file i.e. xlsx/xls file or a notepad file. Therefore the Program allows for a database that can be updated with an import functionality from an excel file.

Generate Random values: System can automatically generate randomly data values to check system simulation.

System output section: It initially displays a table of titles. After input of data, it displays the inputted data, its memory and assigned Number of times data is accessed as well as the optimal solution of the task given.

The Number of times data is accessed Value section: It identifies the highest total Number of times data is accessed obtained which result in optimal value to be taken by the System and the System Throughput that indicates the quantity of processes finished per unit time. This section displays the optimal solution to the input data that should be loaded into the system.

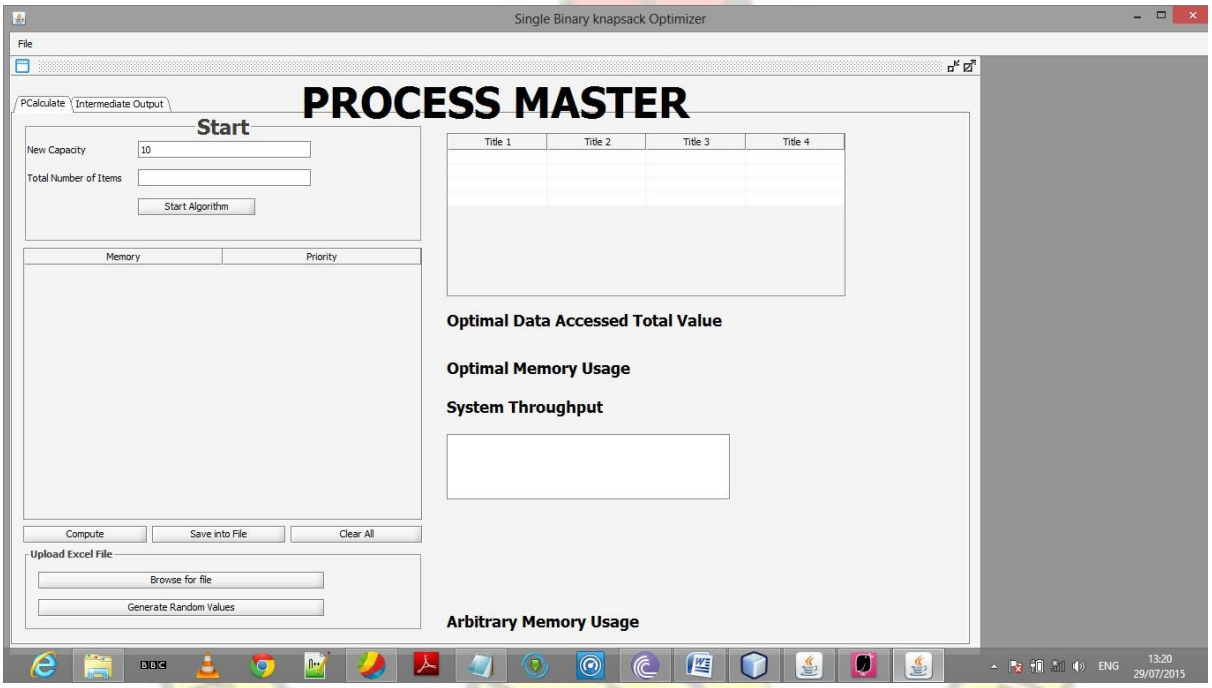


Figure 4.1: PCalculate home screen

4.2.3 The Intermediate output pane

This pane displays an intermediate results of input data. The various comparisms and combinations that the Program goes through to produce the optimal results. It displays the number of loop times or cycles undergone to reach optimal solution. Items which meet the constraints are indicated by 1 whereas items which do not get picked or donot meet the constraints are indicated by 0. This indicate the solution string of input data.

Additionally, it displays the memory space capacity utilized by loaded Processes.

4.2.4 Back-end Specifications

The back-end specification works on methods of the system which the user does not see and serves directly in support of the front-end services. The main idea of the back-end of the system is to execute dynamic programming methodologies given the inputs captured from the user. The back-end computes the following: the system utilisation, the System Through Put and generation of figures for processes as well as saved file and browse for file.

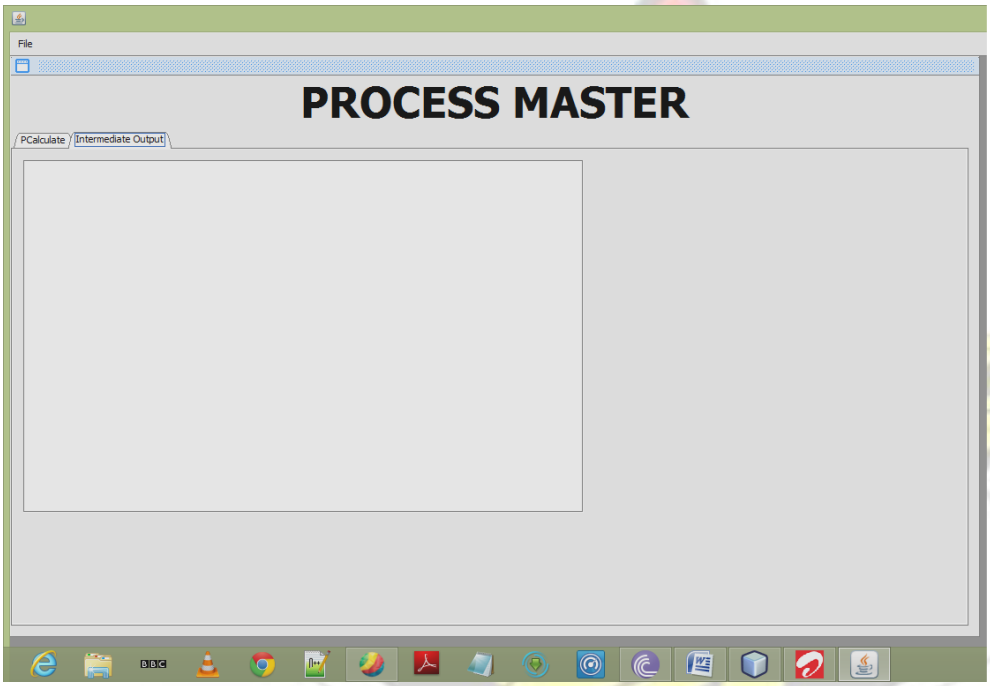


Figure 4.2: Intermediate output home screen

4. 2. 5 Technologies and Software

In considering how to actualize the software, consideration ought to be taken when picking the programming language(s) it will be composed in. The key high state necessities from a software convenience point of view are:

- Easy to utilize Graphical User Interface
- Fast calculation of numerically intensive computations

4.2.6 Java

Java is a universally useful, simultaneous, class-based, object-oriented language that is particularly intended to have as few execution conditions as would be prudent. It is expected to let application designers "write once, run anywhere" implying that compiled Java code can execute at all platform that support Java without the requirement for recompilation.

Java is at present a standout amongst the most prominent programming languages being used, and is generally utilized from application programming to web applications. The program,

The Binary Knapsack optimizer is written using the Java programming language.

4.2.7 **Netbeans**

Netbeans is a an “integrated development environment (IDE)” for developing mostly with Java, and additionally with different languages, specifically HTML5, PHP, C/C++ etc. Netbeans provide the best support for fast and smart coding editing, write bug free code, enable quick user interface creation and provides the best support for latest Java technologies aside its ability to support cross platform and multiple languages. Its tremendous influential software package offers all sort of security and service to the users. This integrated development environment is used to write and edit the code and design the interface.

4.3 Data Analysis

The System information with randomised data value was tested with the computer software developed in Java using the dynamic programming algorithm.

From Table 4.1, there are a total of 15 processes and that of Table 4.2 has 32 processes.

Taken the system information each process carry, if all processes are allowed to load as soon as created, from Table 4.1, the computer system will require total memory capacity of 88mb exceeding the main memory capacity of 32mb. Table 4.2 will demand an arbitrary overall memory space of 2411mb from a System with capacity limit 512mb. Therefore from Table

4.1, an additional memory of 56mb needs to be created and that of Table 4.2 is1899mb. Therefore it is infeasible to allow all the process to run without running into low memory or system crushes. A summary is shown in figure 4.3

Since the memory cannot allocate space or hold all the process at a time, some process(es) should be allowed to run whiles others wait for their turns. Most of the authors, programmers and proponents of computers adopt existing strategies of allocating memory from one of this two:

1. Load data into memory by size (Strategy 1)
2. Load data into memory by number of accesses (Strategy 2)

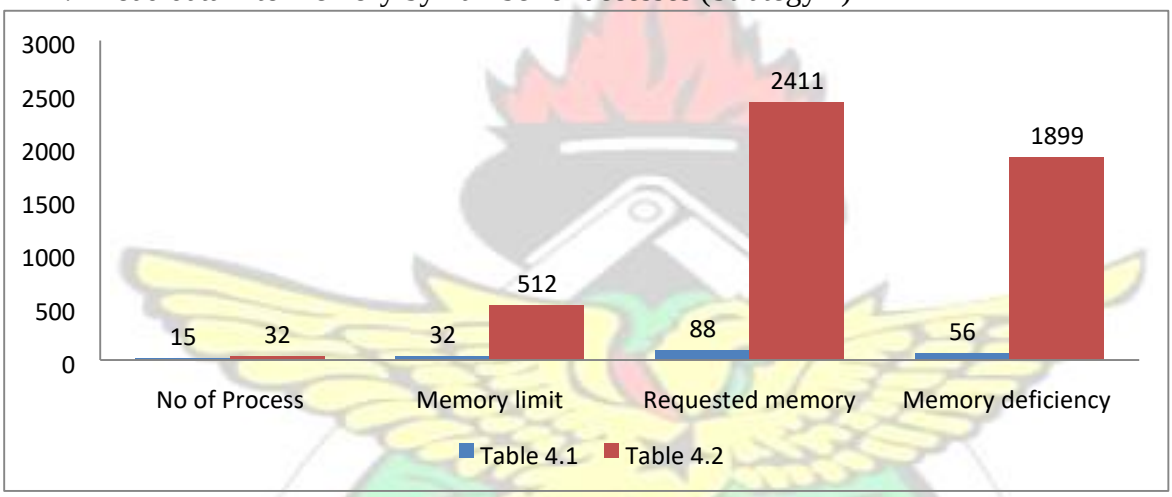


Figure 4.3: Memory demand of Table 4.1 and Table 4.2
Source: Author Survey

Strategy one allows data/process to be loaded according to data size picking data with smallest sizes first until the system capacity is reached. From Table 4.1, processes 1, 2, 4, 7, 9, 11 satisfy the condition over the other processes and therefore by precedence it will be given concern. Together, this six (6) data/process will require 28mb space, with 4mb free. From Table 4.2, processes 1, 2, 5, 7, 8, 16, 19, 20, 23, 24, 25, and 26, a throughput of 12 data/processes satisfy the condition with a total request memory of 495mb, 17mb space left unused.

Using the dynamic programming approach, the solution after passing the various data/Processes of Table 4.1 through the Single Binary Knapsack optimizer identified,

Processes 5, 7, 9, 11, 12, 14, 15 to be the Processes that should be loaded into memory for CPU scheduling (illustrated in figure 4.4). The Processes with a Throughput of 7 (seven) has the total memory space of 31mb leaving 1mb free. For Table 4.2, when the 32 processes are entered into the Program, Single Binary Knapsack optimizer, Processes 1, 2, 3, 4, 5, 7, 8, 19, 24, 25, 26 were the items that meet the criteria totalling 11 (Throughput) with an optimal memory utilisation of 504, 8mb less than the total system capacity (illustrated in figure 4.5).

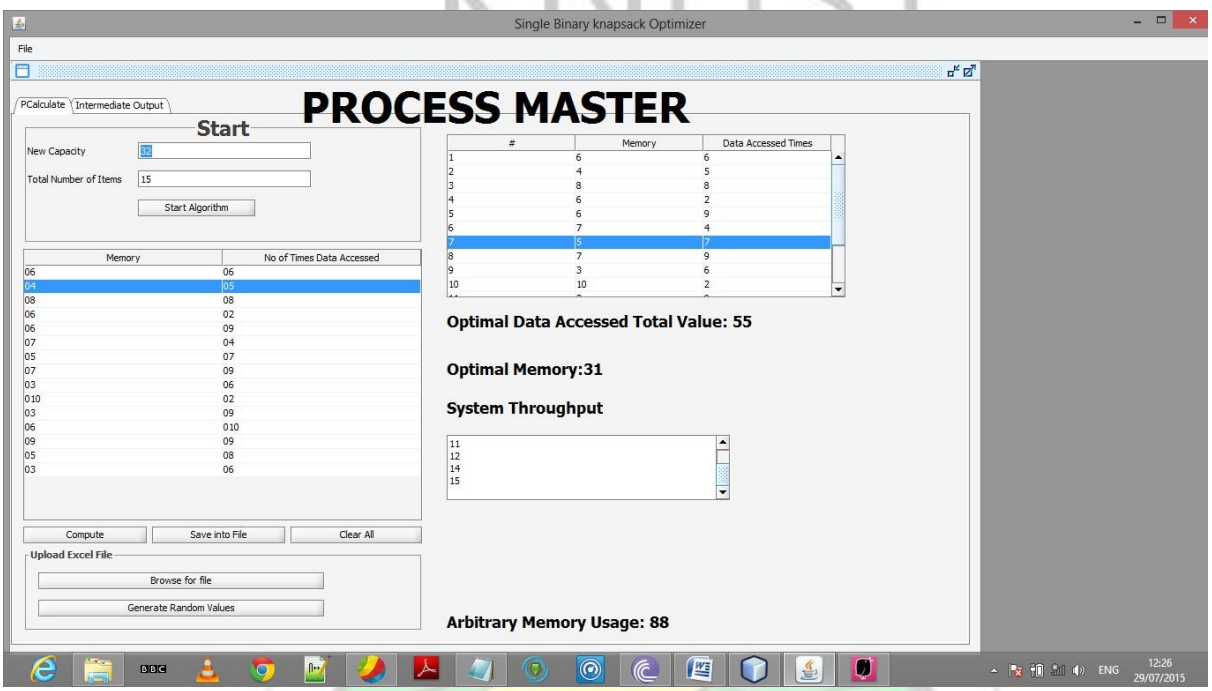


Figure 4.4: Program Solution Output of Table 4.1

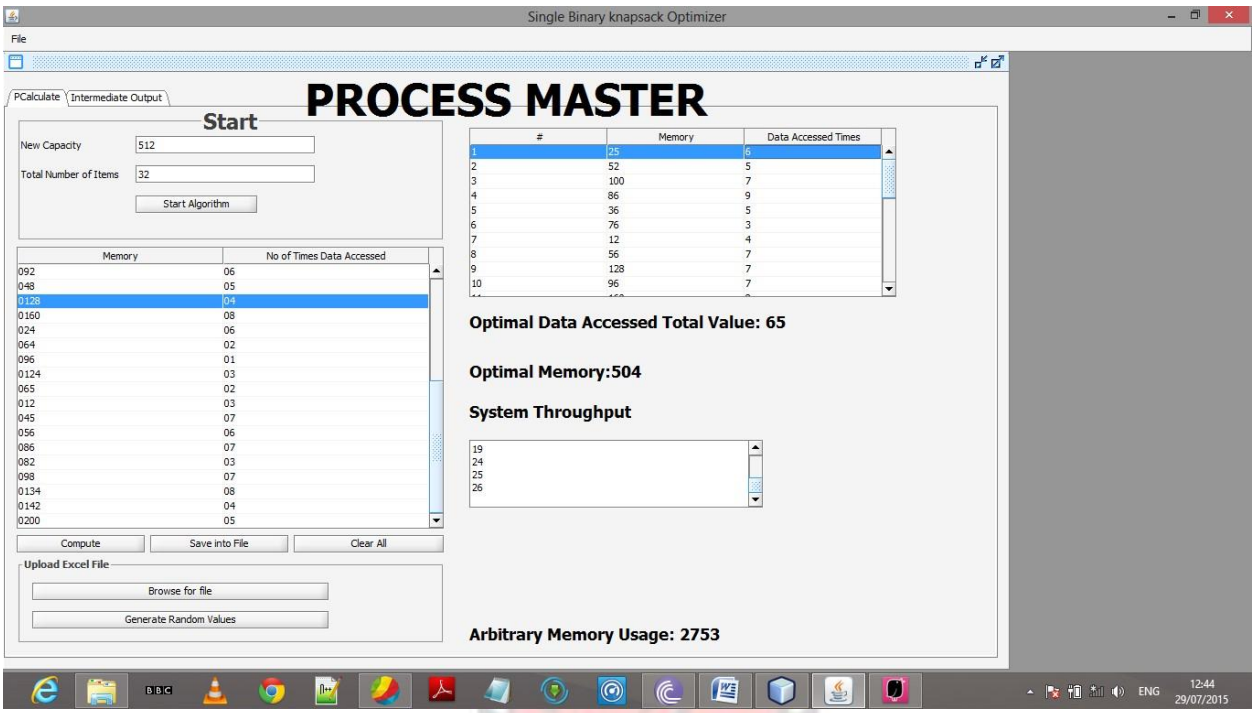


Figure 4.5: Program Solution Output (2)

Table 4.3 compares the optimal data from Table 4.1 of *Load data into memory by size Strategy* and the *Dynamic Programming approach.*

Load data into memory by size (Strategy 1)		Dynamic Programming Approach	
Processes No	data size/mb	Processes No	Data size/mb
1	6	5	6
2	4	7	5
4	6	9	3
7	5	11	3
9	3	12	6
11	3	14	5
		15	3

Table 4.3:Comparism of optimal data of Table 4.1
Source: Author Survey

Table 4.4compares the optimal data from Table 4.2 of *Load data into memory by size Strategy* and the *Dynamic Programming approach*.

Load data into memory by size (Strategy 1)	Dynamic Programming Approach

Processes No	data size/mb	Processes No	Data size/mb
1	25	1	25
2	52	2	52
5	36	3	100
7	12	4	86
8	56	5	36
16	48	7	12
19	24	8	56
20	64	19	24
23	65	24	12
24	12	25	45
25	45	26	56
26	56		

Table 4.4: Comparism of optimal data of Table 4.2

Specification	(Strategy 1)	Dynamic Programming Approach
System throughput	6	7
Memory acquired	28	31
Used memory	4	1

Table 4.5: Memory utilisation analysis of Table 4.1

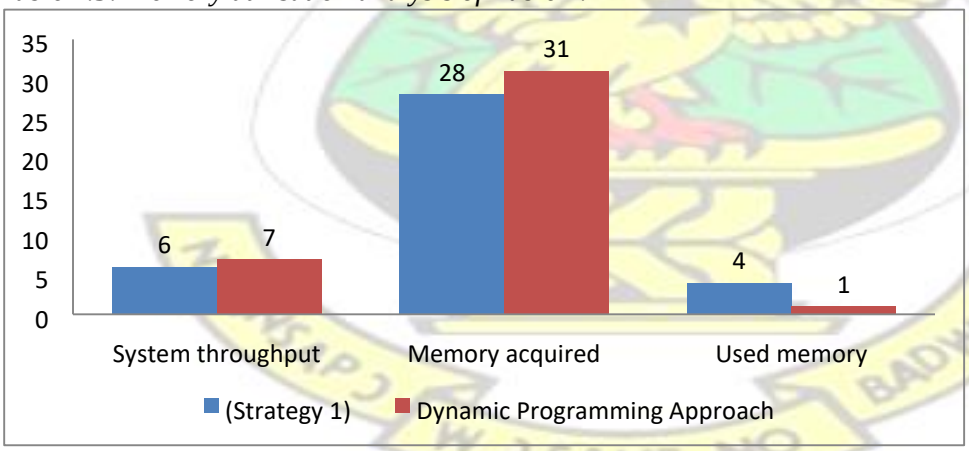


Figure 4.6: Memory utilisation of Table 4.5

Table 4.5 illustrate the analysis of Memory utilisation of Table 4.1 where as Table 4.6 illustrate the analysis of Memory utilisation of Table 4.2

Specification	(Strategy 1)	Dynamic Programming Approach
System throughput	12	11

Memory acquired	495	504
Used memory	17	8

Table 4.6: Memory utilisation analysis of Table 4.2
Source: Author Survey

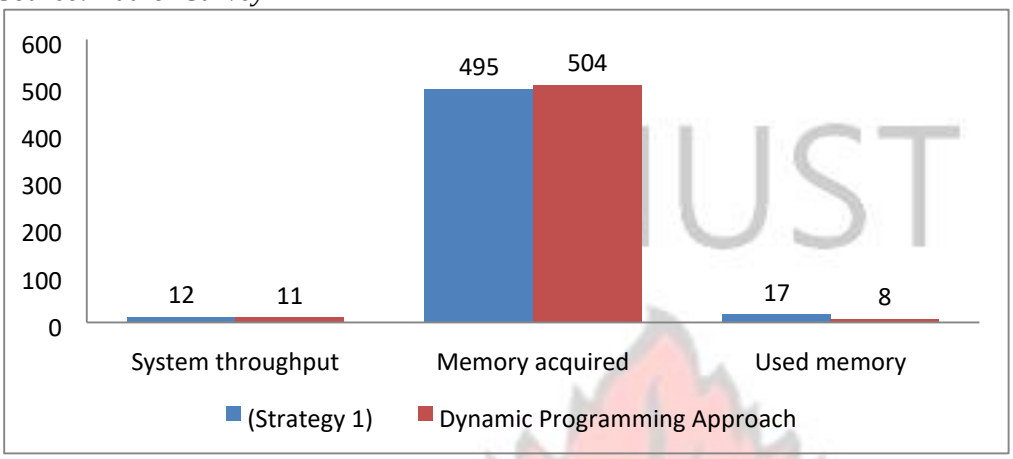


Figure 4.7: Memory utilisation of Table 4.6

Strategy two allows data/process to be selected according to the number of times data is access or the data access times until the system capacity is reached. Passing the data in table 4.1 through this heuristic strategy, processes 5, 8, 11, 12, and 13 satisfy the condition due to its higher access times. A throughput of 5, it requires 31mb space to manage this processes.

Processes 4, 11, and 18 giving a throughput of 3 from Table 4.2 satisfy the condition of heuristic strategy two. The processes will require 406mb to allow the three most frequently accessed data to run in memory holding the remaining data/process in queue. This will leave 106mb free space in memory unutilized.

Using the Dynamic programming approach for solving table 4.1 as already illustrated in figure 4.4, a throughput of 7, Processes 5, 7, 9, 11, 12, 14, and 15 is loaded into memory for CPU scheduling. With Table 4.2, Processes 1, 2, 3, 4, 5, 7, 8, 19, 24, 25, 26 totalling a throughput of 11 is loaded for memory allocation since their memory utilisation requirement is 8mb less than the 512mb memory capacity of the computer system. This is illustrated in figure 4.5.

Table 4.7 compares the optimal data from Table 4.1 *of Load data into memory by number of accesses Strategy* and the *Dynamic Programming approach.*

Load data into memory by number of accesses (Strategy 2)		Dynamic Programming Approach	
Processes No	Number of times data is accessed	Processes No	Number of times data is accessed
5	9	5	9
8	9	7	7
11	9	9	6
12	10	11	9
13	9	12	10
		14	8
		15	6

Table 4.7:Comparism of optimal data of Table 4.1

Source: Author Survey

Table 4.8 compares the optimal data from Table 4.2 of *Load data into memory by number of accesses Strategy* and the *Dynamic Programming approach.*

Load data into memory by number of accesses (Strategy 2)		Dynamic Programming Approach	
Processes No	Number of times data is accessed	Processes No	Number of times data is accessed
4	9	1	6
11	8	2	5
18	8	3	7
		4	9
		5	5
		7	4
		8	7
		19	6
		24	3
		25	7
		26	6

Table 4.8: Comparism of optimal data of Table 4.2
Source: Author Survey

Table 4.9 illustrate the analysis of Memory utilisation of Table 4.1 for strategy two

Specification	(Strategy 2)	Dynamic Programming Approach
System throughput	5	7
Memory acquired	31	31
Used memory	1	1

Table 4.9: Memory utilisation of Table 4.1
Source: Author Survey

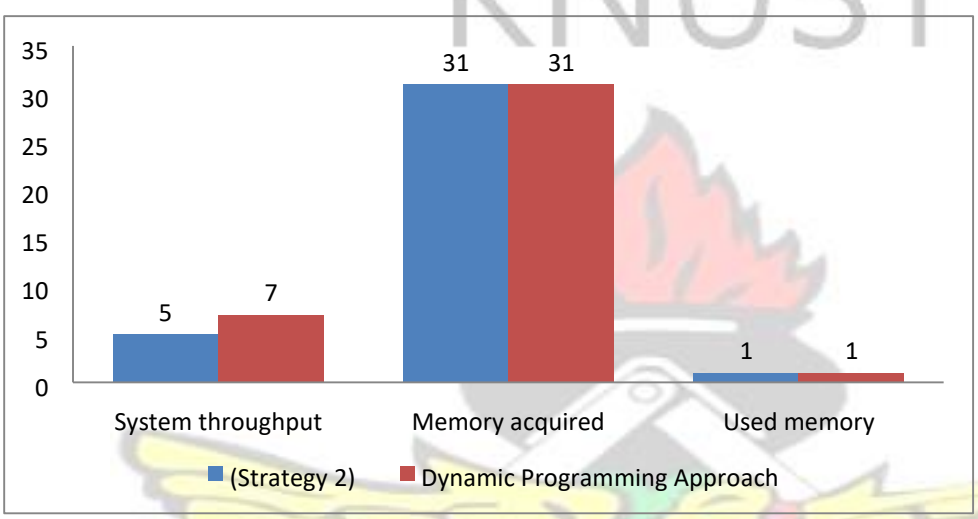


Figure 4.8: Memory utilisation of Table 4.9

Specification	(Strategy 2)	Dynamic Programming Approach
System throughput	3	11
Memory acquired	406	504
Used memory	106	8

Table 4.10: Memory utilisation of Table 4.2
Source: Author Survey

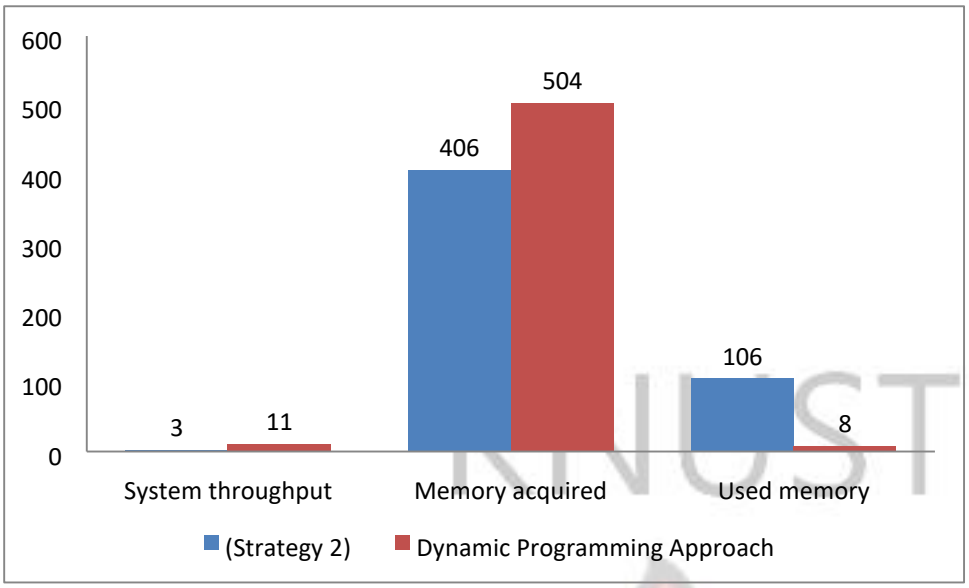


Figure 4.9: Memory utilisation of Table 4.10

4.4 Data findings

Comparing the Dynamic programming approach to other existing strategies employed by computer programmers and system developers for optimising memory as stated in this research work P. 54, the Dynamic programming approach tends to out-perform most of them. The Dynamic programming approach tends to pick data/process that can enhance efficient utilisation of memory. It also picks as many processes as possible provided their data sizes do not exceed the system capacity. from Table 4.5 even though loading data into memory by size strategy (Strategy one) allowed system throughput of 6, it left an unused memory space of 4mb compared to the Dynamic approach of allowing 7 process, and an efficient memory utilisation of 31mb, illustrating that there is optimal utilisation of memory. Same can be made of table 4.6 with strategy 1 picking more data/process than the Dynamic approach, it left more unutilized space than the Dynamic approach which may lead to memory leakage. The strategy of loading data into memory by size strategy tends to favour only process/data with smaller data size but data with larger data size takes a long time to be given space thereby increasing the allocation time of such data irrespective of the higher system priority or access frequency that a data/process may have.

The second strategy although can be used to Optimize memory, it also failed to perform better compared to the proposed approach. From table 4.9, although the memory requirement of selected processes of Table 4.1 for *Load data into memory by number of accesses strategy* (strategy 2) equals the Dynamic programming approach, the system throughput of the heuristic strategy two falls short of 2 more processes compared to the Dynamic programming approach which allow 7 processes to load at time. In table 4.10 the Dynamic programming approach achieves 8 more processes than heuristic strategy two. With only 3 system throughput, strategy two require 406mb memory spaces to allow the most frequently used process to run leaving behind 106mb memory unutilized. The Dynamic programming approach, with 11 system throughput, made an effective utilisation of memory. It required 504mb memory leaving 8mb space free. At this point it can be deduced that *The Load data into memory by number of accesses strategy* may some times not make use of efficient use of memory and may lead to memory loses and leakages

Additionally, the Load data into memory by number of accesses strategy take a longer time for a fresh new data to be loaded into memory space since it favours data/process with a higher number/time accessed otherwise such process/data is held in queue. Therefore newly created process which probably may need little space to load will have to wait a long while to execute.

CHAPTER FIVE

SUMMARY, CONCLUSION AND RECOMMENDATIONS

5.0 Introduction

This chapter presents summary of the findings from the study, and recommendations for software and operating system developers and builders. It again recommends further study possibility areas for future researchers. The chapter provides the concluding statements of the research based on the findings.

5.1 Summary

The main purpose of the research is to optimize memory of Processes to reduce system crushes, system running low memory, system underperformance, system overheat and difficulty in accessing data. The research model the selection procedure of data/Process from process queue and loading them into memory for execution as a 0/1 knapsack problem. Known for its combinatorial optimisation problem, the 0–1 knapsack problem is NP-hard. Therefore majority of algorithms for solving Knapsack problem typically, use implicit enumeration approaches or a greedy approach. The method adopted to solve this problem is the Dynamic programming approach.

The study considered sampled values of a Process that holds its memory usage and Number of times data is accessed information and uses them to solve the problem. Input values which are negative or alphabet/symbols were not considered.

The study was compared to other existing heuristic strategies of optimising memory - Load data into memory by size (Strategy 1) and Load data into memory by number of accesses (Strategy 2). It was found that the Dynamic approach out-performed these strategies. The Dynamic programming approach provides better results, makes efficient use of memory and allowed optimal number of processes to run at a time.

The use of dynamic programming approach helps to prevent frequent loss of data and prevent uncontrolled, uncounted loss of memory by other strategies.

The proposed system was implemented using java.

5.2 Conclusion

In this research thesis, we propose a partial enumeration technique based on an exact enumeration algorithm like the dynamic programming for effective utilisation and optimization of memory. The problem identified is one that has a single linear constraint, a linear objective function which sums the values of data/process in memory, and the added restriction that each

data/process should be in memory or not. The Dynamic programming approach proved to quickly find an optimal solution or a near optimal solution in some situations where exact solution is not possible as opposed to a heuristic that may or may not find a good solution.

From the thesis, it is shown that the Dynamic programming algorithm is more efficient and yield better result than other existing heuristic algorithm. Dynamic Programming algorithm is easy to implement since no sorting is necessary, saving the corresponding sorting time. Additionally, the time complexity taken to solve the Dynamic programming is 0(n*W) compared to the 0/1 knapsack algorithm running time of O(2^n). Taken that n is the number of items and W is the Capacity limit.

5.3 Recommendations

Most of out daily activities can be modelled as a knapsack problem which needs a careful approach in undraping at the optimal solution. Dynamic programming is an easy but useful technique of solving this kind of problems.

The Dynamic programming algorithm outperform heuristics based on local search techniques and local search algorithm since they fail to fully exploit the structural properties of knapsack problems and find better solutions than those even obtained by a greedy algorithm. It is envisaged that authors, software creators and system programmers will employ the Dynamic programming algorithm when they arrive at such problems.

The software which is easy to use is not only systematic, but also swift. It can produce and displays results within 30 seconds. It is therefore envisage that higher results within the shortest possible time may be achieved when adopted by Operating system developers, System creators or Software engineers in their design creation to model real life computer problems to achieve optimum results.

5.4 Further Studies

Whiles it is acknowledged that the focused is on 0-1 knapsack algorithm, we suggest further that in future work, we will investigate evolutionary heuristics (Genetic Algorithms and ANT method) and hybrid heuristics for finding suitable solution to optimising memory in computer system.

REFERENCES

Abraham Silberschatz, Greg Gagne, and Peter Baer Galvin (2012). *Operating System Concepts*, 9th Ed. John Wiley & Sons, Inc.

Aggarwal and Hartline (2006*). Knapsack Auctions*. www.research.microsoft.com

Akinc, U. (2006). *Approximate and exact algorithms for the fixed-charge knapsack problem.*www.sciencedirect.com

Alfredo Giménez, Todd Gamblin, Barry Rountree, Abhinav Bhatele, Ilir Jusufi, Bernd Hamann, and Peer-Timo Bremer. *Dissecting On-Node Memory Access Performance: A Semantic Approach. In Supercomputing 2014 (SC'14)*, New Orleans, LA, November 16-21 2014. LLNL-CONF-658626

Amponsah, S. and Darkwah, F. (2007), *Operations Research*. Institute of Distance Learning, KNUST.

Asma'a Lafi (2013). *Memory allocation for real time operating system*. Jordan University of Science and Technology.

Babaioff, M. et al (2007). *Matroids, secretary problems, and online mechanisms*. Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms. Pages 434-443

Bala et al.(2014). *International Journal of Advanced Research in Computer Science and Software Engineering* 4(9), September - 2014, pp. 747-751

Balas, E. and Zemel, E.(1980). *An Algorithm for Large Zero-One Knapsack Problems*. Operations Research.28, 1130-1154

Bazgan, C., Hugot, H and Vanderpooten, D. (2009). *Solving efficiently the 0-1 multiobjective knapsack problem*. Computers and Operations Research, 36(1), 260-279

Bazgan, et.al (2007). *A practical efficient fptas for the 0-1 Multi-objective Knapsack Problem*. Proceedings of the 15th annual European conference on Algorithms. Pages 717-728

Beasley J. E., and Chu, P., C. (1996). *A genetic Algorithm for the set covering problem.* European Journal of Operations Research 94:392-404

Bellman, R., E. (1957). *Dynamic programming*. Princeton University Press, Princeton, NJ.

Benisch, M., Andrews, J., Bangerter, D., Kirchner, T., Tsai, B. and Sadeh, N. (2005). *CMieux analysis and instrumentation toolkit for TAC SCM.* Technical Report CMUISRI-05-127, School of Computer Science, Carnegie Mellon University.

Bortfeldt A, Gehring H. (2001). *A hybrid genetic algorithm for the container loading problem* [J]. European Journal of Operational Research, 2001, 131(1):143-161.

Boryczka U.(2006). *The influence of trail representation in ACO for good results in MKP*. Advances in Computer Science and Technology (ACST 2006), IASTED Conference, IOS Press, Puerto Vallarta, Mexico

Bottomupcs.com, (2015). What virtual memory is. [online] Available at: http://www.bottomupcs.com/virtual_memory_is.html [Accessed 5 Sep. 2015].

Boyd, S., C. And Cunningham, W., H. (1988). *Small travelling salesman polytopes,* Preprint, Carleton University, Ottawa.

Brookshear, J. G. (1997). *Computer Science: An Overview*. Fifth Edition, AddisonWesley, Reading, MA.

Burger, D., Goodman, J.R. and Sohi, G.S. (1997). *Memory Systems*. The Computer Science and Engineering Handbook, 47-461.

Cáceres E. N., and Nishibe, C. (2005). *0-1 Knapsack Problem: BSP/CGM Algorithm and Implementation.* IASTED PDCS: 331-335.

Chang, J.T., Meade, N., Beasley, J., E. and Sharaiha, Y.M.. (2000). *Heuristics for cardinality constrained portfolio optimization. Comp*. Operasions. Research. 27: 1271-1302

Chavan, A., S. (2011). *A Comparison of Page Replacement Algorithms*. IACSIT International Journal of Engineering and Technology, Vol.3, No.2, April 2011

Chekuri, C. (2005), *A PTAS for the multiple knapsack problem*. Society for industrial and applied mathematics : 213-222

Chen, B., Kamel, I. and Marsic, I. (2010). *Memory Management in Smart Home Gateway, Smart Home Systems*. Mahmoud A. Al-Qutayri (Ed.), ISBN: 978-953-307050-6, In Tech, DOI: 10.5772/8406. Available from: http://www.intechopen.com/books/smart-home-systems/memory-management-insmart-home-gateway

Chirag S. (2011). *Role of the Memory management unit in Operating system*. [Online] Available from:http://www.techulator.com/resources/4498-Role-Memorymanagement-unit-Operating.aspx

Chu P.C and Beasley J. E. (1998), *A genetic algorithm for multidimensional knapsack problem.* Journal Heuristics. 4:63-68

Chu, P. C and Beasley J. E (1997). *A genetic algorithm for generalized assignment problem*. Computer Operations Research 24: 17-23

Chu, P. C and Beasley J. E (1998b). *Constraint handling in genetic algorithm: the set partitioning problem.* Journal Heuristics 4: 323-357

Coquand,T., Dybjer, P., Nordström, B., and Smith, J. (1999). Types for Proofs and Programs, International Workshop TYPES'99, Lökeberg, Sweden, Available from: Selected Papers. Lecture Notes in Computer Science 1956, Springer 2000, ISBN 3540-41517-3

Cyberiapc.com (2015). Operating Systems and Kernels: Concepts, Virtual Memory. Retrieved 5 September 2015, from http://www.cyberiapc.com/os/concepts-mmvm.htm

Dantzig B. G. (1957). *Discrete-variable Extremum*. Operations Research vol. 5:266-288

Dantzig T. (1937). *Number; Language of Science.* Macmillan N.Y.

Denning, P.J. (2005). *The Locality Principle.* Communications of the ACM, 48(7), 19- 24.

Dipti Diwase, Shraddha Shah, Tushar Diwase, Priya Rathod (2012). *Survey Report on Memory Allocation Strategies for Real Time Operating System in Context with Embedded Devices.* International Journal of Engineering Research and Applications (IJERA) ISSN: 2248-9622 ,Vol. 2, Issue 3, May-Jun 2012, pp.1151-1156

Dizdar, D., Gershkov, A. and Moldovanu, B. (2010). *Revenue maximization in the dynamic knapsack problem.* Theoretical Economics 6 (2011), 157–184

Elhedhli, S. (2005). *Exact solution of a class of nonlinear knapsack problems.* Operations Research Letters archive, Volume 33 Issue 6, Pages 615-624

Essandoh, K. (2012). *Modeling site development for garbage disposal as a 0-1 knapsack problem, a case study of the sekondi-takoradi metropolis.* Master's degree thesis, KNUST

Fabiano do Prado Marques and Marcos Nereu Arenales (2007). *The constrained compartmentalised knapsack problem*. www.sciencedirect.com

Faigle, U. and Kern, W. (1991). *The Shapley value for cooperative games under precedence constraints*. International Journal of Game Theory 21 (3), 249-266

Feng-Tse Lin (2008). *Solving the knapsack problem with imprecise weight coefficients using genetic algorithms.* www.sciencedirect.com

Feng-Tse Lin, and Jing-Shing Yao(2001). *Using fuzzy numbers in knapsack problems.* European Journal of Operational Research 135(1): 158-176

Figuera, J.R., Wiecek, M. and Tavares, G. (2006). *Multiple criteria knapsack problems : Network models and computational results*. In Proceedings of the multi-Objective Programming and Goal Programming Conference (MOPGP'06), Tours

Fleszar, Khalil S. Hindi (2009). *Fast, effective heuristics for the 0–1 multi-dimensional knapsack problem*. www.sciencedirect.com

Florios, K. et. al. (2009). *Solving multi objective multi constraint knapsack problem using Mathematical programming and evolutionary algorithm*. European Journal of Operational Research 105(1): 158-17

Freville, A., and Plateau, G. (2004). *An efficient preprocessing procedure for the multidimensional 0–1 knapsack problem.* www.sciencedirect.com

Gallego, Guillermo and Garrett van Ryzin (1994), "*Optimal dynamic pricing of inventories with stochastic demand over finite horizons."* Management Science, 40, 999–1020. [158, 165, 167]

Gay, E, D. (2001). *Memory Management with Explicit Regions*. Ph.D. Thesis, Department of Computer Science, University of California, Berkeley Retrieved from: theory.stanford.edu/~aiken/publications/.../gay.pdf

Gershkov, Alex and Moldovanu, B. (2009), *"Dynamic revenue maximization with heterogeneous objects:* A mechanism design approach." American Economic Journal:

Gholamian, M.R., Fatemi Ghomi, and Ghazanfari, M. (2007). *A hybrid system for multiobjective problems – A case study in NP-hard problems*. www.sciencedirect.com

Gil-Lafuente, A., de-Paula, L., Merig-Lindahl, J., M., Silva-Marins, F., and de AzevedoRitto, A. (2013). Decision Making Systems in Business Administration: Proceedings of the MS'12 International Conference. World Scientific Publishing Co., Inc. Retrieved from http://dl.acm.org/citation.cfm?id=2509785

Giménez, A. (2014). *Dissecting on-node memory access performance: a semantic approach.* Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, November 16-21, 2014, New Orleans, Louisana

Goguen, H., Brooksby, R. and Burstall, R.M.(2002). *Memory Management: An Abstract Formulation of Incremental Tracing.* Selected papers from the International Workshop on Types for Proofs and Programs, p.148-161

Gomes da Silva, C., Figueira, J. and Clímaco, J. (2007). Integrating partial optimization with scatter search for solving bi-criteria {0,1}-knapsack problems. European Journal of Operational Research, 177(3), pp.1656-1677.

Gupta, L and Luss, H. (1980). *Logic Programming*: 21st International Conference of ICLP, Spain.

Hanafi, S. and Freville, A. (1998), *An efficient tabu search approach for the 0-1 multidimensional knapsack problem.* http://www.sciencedirect.com

Hifi, M. and Sadfi,S.(2002). *The knapsack sharing problem: An exact algorithm*. Journal of Combinatorial Optimization, vol. 6, pp. 35-45.

Horowitz, E. and Sahni, S. (1972). *Computing partitions with applications to the Knapsack Problem.* Journal of ACM, 21, 277-292

Hristakeva, M. and Dipti, S. (2011). *Solving the 0-1 Knapsack Problem with Genetic Algorithms.* http://freetechebooks.com/file-2011/knapsack-problem Microeconomics, 1(2), 168–198. [158] http://en.wikipedia.org/wiki/Knapsack_problem

http://www.bottomupcs.com/memory.html

http://www.cs.cornell.edu/courses/cs312/2004fa/lectures/lecture23.htm

http://www.techulator.com/resources/4498-Role-Memory-management-unit-

http://www.tekniske.rapporter/1995/95-1.ps.gz.

https://www.github.com/ianw/bottomupcs

Hugot et.al, (2006). *A bi criteria approach for the data association problem*. Annals of Operations Research, 147(1),217-234

Hutter, M and Mastrolilli, M. (2006), *Hybrid Rounding Techniques for Knapsack Problems*. Discrete Applied Mathematics, 154:4 640-649

Islam, M. T. (2009). *Approximation Algorithms For Minimum Knapsack Problem*. Master's degree thesis, Islamic University of Technology

ITL Education Solutions Limited (2009). *Introduction to Information Technology*. Pearson Education Canada

Jingwei, D. et al.(2012). *Optimizing Software of Memory Management on ARM.* Proceedings of the 2012 Second International Conference on Instrumentation, Measurement, Computer, Communication and Control Pages 1390-1392, Washington, DC, USA doi>10.1109/IMCCC.2012.327

Kalai, R. and Vanderpooten, D. (2006). *Lexicographic a-Robust Knapsack Problem* http://ieeexplore.ieee.org/xpl/freeabs

Kellerer, H., Pferschy, U., Pisinger, D. (2004). *Knapsack Problems*. Springer, Berlin Heidelberg

Kolesar, P., J. (1967). *A branch and bound algorithm for the knapsack problem*. Management Science, 13, 723-735

Kornaros, G., Papaefstathiou, I. Nikologiannis, A. Zervos, N. (2003). *A fully programmable memory management system optimizing queue handling at multi gigabit rates.* Proceedings of the 40th annual Design Automation Conference, Pages 54 – 59, 2-6 New York, NY, USA doi> 10.1109/DAC.2003.1218800

Kosuch, S. (2010). *An Ant Colony Optimization Algorithm for the Two-Stage Knapsack Problem* .Institutionen for datavetenskap (IDA), Linkopings Universitet, Sweden

Kosuch, S. (2011). *Towards an Ant Colony Optimization algorithm for the Two-Stage Knapsack problem.* In: VII ALIO/EURO Workshop on Applied Combinatorial Optimization, pp. 89{92. http://paginas.fe.up.pt/~agomes/temp/alio_euro_2011/uploads/Conference/ALIOEURO_2011_proceedings_v2.pdf#page=101

Kosuch, S. and Lisser, A. (2009). *On two-stage stochastic knapsack problems*. Discrete Applied Mathematics Volume 159, Issue 16

Leeman, M. et al (2005). *Methodology for Refinement and Optimisation of Dynamic Memory Management for Embedded Systems in Multimedia Applications.* Journal of VLSI Signal Processing (Impact Factor: 0.73). 01/2005; 40:383-396. DOI>10.1007/s11265-005-5272-4

Leeman, M., Atienza, D., Deconinck, G., Vincenzo De Florio, Mendías, J. M., YkmanCouvreur, C., Catthoor, F. and Lauwereins, R. (2005). *Methodology for Refinement and Optimisation of Dynamic Memory Management for Embedded Systems in Multimedia Applications.* Journal of VLSI Signal Processing (Impact Factor: 0.73). 01/2005; 40:383-396. DOI>10.1007/s11265-005-5272-4

Li V.C., et al (2004). *Towards the real time solution of strike force asset allocation problems.* http://www.researchgate.net/publication/257153198_Towards_the_real_time_solution_of_strike_force_asset_allocation_problems

Li, V. C. and Curry, G. L. (2005). *Solving multidimensional knapsack problems with generalized upper bound constraints using critical event tabu search.* Computer and operational research. Vol. 32. pp. 825 - 848.

Li, X. and Elhanany, I. (2005). *A pipelined memory management algorithm for distributed shared memory switches.* Communications, 2005. ICC 2005. 2005 IEEE International Conference on (Volume:3)

Lin and Wei (2008). *Solving the knapsack problem with imprecise weight coefficients using Genetic algorithm.* www.sciencedirect.com

Lin, C. (2013).Virtual Memory. http://www.cs.umd.edu/class/sum2003/cmsc311/Notes/Memory/virtual.html

Lin, Y.S. and Mattson, R.L. (1972). *Cost-Performance Evaluation of Memory Hierarchies,* IEEE Transactions Magazine, MAG-8(3), 390-392.

Luss, H. (1975). *Allocation of effort resource among competing activities,* Operations Research 23:360-366.

Martello, S. and Toth P. (2000). *New trends in exact algorithms for 0-1 knapsack problems.* European Journal of Operations Research 123: 325-332

Martello, S., & Toth, P. (1977). *An Upper Bound for the Zero-One Knapsack Problem and a Branch and Bound algorithm.* European Journal of Operational Research, 1, 169-175

Martello, S., Pisinger, D. and Paolo, T. (2000). *New trends in exact algorithms for the 0 – 1 knapsack problem.* http: // citeseerx.istpsu/viewdoc/download?doi10.1.11.89068rep=rep|type=ps

Masmano, M., Ripoll, I. and Crespo, A. (2004). "*TLSF: A new dynamic memory allocator for real-time systems*," in proceedings of 16th Euromicro Conference on Real-Time Systems, Catania, Italy, July 2004, pages 79-88.

Moraga, R.J., DePuy, G.W. and Whitehouse, G.E. (2005). *Meta-RaPS approach for the 0-1 Multidimensional Knapsack Problem.* www.sciencedirect.com

Munapo, E. (2008). *The efficiency enhanced branch and bound algorithm for the knapsack model*. Adv. Appl. Math. Anal. ISSN 0973-5313, 3(1): 81-89

Nauss, R,. M. (1976). *An Efficient Algorithm for the 0-1 Knapsack Problem*. Management Science, 23, 27-31

Nutt, G. (1997*), Operating Systems: A Modern Perspective*, First Edition, AddisonWesley, Reading, MA

Ofori, O. E. (2009). *Optimal resource Allocation Using Knapsack Problems: A case Study of Television Advertisements at GTV*. Master's degree thesis, KNUST

Operating.aspx

Owusu-Bempah, E. (2013). *Optimal resources allocation; a Case Study of Sekyere Central District Assembly, Nsuta-Ashanti.* Master's degree thesis, KNUST

Pisinger, D. (1994). *Core problems in knapsack algorithms*. Operations Research 47, 570575 .

Pisinger, D. (2003). Where are the hard knapsack problems? Technical Report 2003/08, Department of Computer Science, University of Copenhagen, Copenhagen, Denmark.

Qi Zhu , and Ying Qiao (2012). *A Survey on Computer System Memory Management and Optimization Techniques,* American Journal of Computer Architecture, Vol. 1 No. 3, pp. 37-50. doi: 10.5923/j.ajca.20120103.01.

Qian, F. and Ding, R. (2007). *Simulated annealing for the 0/1 multidimensional knapsack problem*. Numerical Mathematics - English Series -, Vol 16:320{327, 2007. ISSN 1004-8979.

Rathod, V. and Chawan, P. (2013). *Implementation of Page Replacement Algorithm with Temporal Filtering for Linu*x. Journal of Engineering, Computers & Applied Sciences (JEC&AS) Volume 2, No.6, pp 72-76,www.informit.com

Robert M, & Thompson, K (1978). *Password Security: A Case History*. Bell Laboratories, K8.

Samphaiboon, N. and Yamada, Y (2000). Heuristic and Exact Algorithms for the Precedence-Constrained Knapsack Problem. Journal of Optimization Theory and Applications June 2000, Volume 105, Issue 3, pp 659-676

Shang, R., Ma, W. and Zhang, W (2006). *Immune Clonal MO Algorithm for 0/1 Knapsack Problems*. Lecture Notes in Computer Science, 2006, Volume 4221/2006, 870-878.

Silberschatz, A., and Peterson, J. (1989). *Operating System Concepts*. Addison-Wesley, Reading

Silva et.al (2008). *Core problem in bi criteria 0-1 knapsack problems*. Retrieved from: www.sciencedirect.com

Simoes, A, and Costa, E. (2001). *Using Genetic Algorithm with Asexual Transposition.* Proceedings of the genetic and evolutional computation conference (pp 323-330)

Sinapova, L. (2014). *An Introduction to Algorithms*. Simpson College, Department of Computer Science 701 North C Street, Indianola IA 50125. [online] Available at: http://faculty.simpson.edu/lydia.sinapova/www/cmsc250/LN250_Levitin/L13GA.htm [Accessed 6 Sep. 2015].

Singer, J. and Jones, R.E. (2011). *Economic theory for memory management optimization*. Proceedings of the 6th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems, Jul 26, 2011, New York, NY, USA, doi>10.1145/2069172.2069176

Singer, J., and Jones, R.E. (2011). *Economic theory for memory management optimization.* Proceedings of the 6th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems. Article No. 4

Stackoverflow (2015). *Printing Items that are in sack in knapsack*. [Online] Available from:http://stackoverflow.com/questions/23186171/printing-items-that-are-in-sack-inknapsack [Accessed: 12th January 2015].

Steven S. Skiena (1997). *The Algorithm Design Manual*. Stony Brook, NY 11794-4400

Taniguchi, F., Yamada, T., and Kataoka, S. (2008). *Heuristic and exact algorithms for the max–min optimization of the multi-scenario knapsack problem* http://www.sciencedirect.com

Tao, Z. and Young, R. (2009). *Multiple Choice knapsack problem. Example of planning choice in transportation.* www.sciencedirect.com

The Memory Management Glossary, http://www.memorymanagement.org/glossary/b.html" Tutorialspoint. (2015). *Operating System - Memory Management*. [Online] Available from:http://www.tutorialspoint.com/operating_system/os_memory_management.htm. [Accessed: 12th January 2015].

Vanderster, D. C. (2008). *Resource Allocation and Scheduling Strategies using Utility and the Knapsack Problem on Computational Grids*. Ph.D Thesis, Department of Electrical and Computer Engineering, University of Victoria

Vercellis, C. (1994). *Constrained multi-project plannings problems: A Lagrangean decomposition approach*. In: European Journal of Operational Research. RePEc:eee:ejores:v:78:y:1994:i:2:p:267-275.

Yamada, T, Futakawa, M., and Kataoka, S. (1998). *Some exact algorithms for the knapsack sharing problem.* www.sciencedirect.com

You, B. and Yamada, T. (2007). *A pegging approach to the precedence-constrained knapsack problem.* www.sciencedirect.com

APPENDIX *Simplified*
Program Code

```
package Knapsack; import

java.awt.Toolkit; import

java.io.File; import

java.util.Vector; import
```

```
javax.swing.JFileChooser; import
javax.swing.JOptionPane;


public class calc extends javax.swing.JInternalFrame {
Vector<Integer> weight = new Vector<Integer>();
   Vector<Integer> value = new Vector<Integer>();
   File f;     public calc()
{
initComponents();
      Dimension d = Toolkit.getDefaultToolkit().getScreenSize();       setBounds(0, 0,
d.width, d.height);       jTable2.setModel(new DefaultTableModel(new
String[]{"Memory", "Priority"}, 0));
jTable2.getDefaultEditor(String.class).addCellEditorListener(               new
CellEditorListener() {                  public void editingCanceled(ChangeEvent e) {
                  System.out.println("editingCanceled");
               }
               public void editingStopped(ChangeEvent e) {
if (jTable2.getSelectedColumn() == 1) {
                     String a = String.valueOf(jTable2.getValueAt(jTable2.getSelectedRow(),
jTable2.getSelectedColumn()));
                     int val = Integer.parseInt(a);
                     if (val > 10) {
                        JOptionPane.showMessageDialog(null, "Please value should not be more than
10");
jButton5.setEnabled(false);
                     }
                  }
               }
            });
```

```
    }

    public void solve(int[] wt, int[] val, int W, int N) {
int NEG = Integer.MIN_VALUE;        int[][] m =
new int[N + 1][W + 1];        int[][] sol = new int[N +
1][W + 1];
        String intermediate = "";
String loop_cycles = "";        int
optimal_memory = 0;
        int cycles = 0;        for (int i = 1; i <=
N; i++) {            for (int j = 0; j <= W; j++)
{                int m1 = m[i - 1][j];
int m2 = NEG;                if (j >= wt[i]) {
m2 = m[i - 1][j - wt[i]] + val[i];
                }
                m[i][j] = Math.max(m1, m2);
sol[i][j] = m2 > m1 ? 1 : 0;
intermediate += ("V[" + i + ", " + j + "] = Max
(" + m1 + ", " + m2 + ") = " + m[i][j]) + " ... (" +
sol[i][j] + ")\n";                cycles++;
            }
        }
        intermediate += "loop cycles: " + cycles + "\n";        jLabel6.setText("Optimal Data
Accessed Total Value: " + String.valueOf(m[N][W]));        String selected_string = "";
int[] selected = new int[N + 1];        for (int n = N, w = W; n > 0; n--) {            if (sol[n][w]
!= 0) {                selected[n] = 1;                w = w - wt[n];
            } else {
selected[n] = 0;
```

```
            }
            selected_string += (selected[n]);
        }
        String items = "";        int
total_nums = 0;        for (int i = 1; i <
N + 1; i++) {            if (selected[i] ==
1) {                items += (i + "\n");
optimal_memory += (wt[i]);
            }
        }
        jLabel8.setText("Optimal Memory:" + optimal_memory);
jTextArea1.setText(items);

        String a[][] = new String[wt.length - 1][3];
int memory = 0;        for (int i = 0; i < N; i++) {
a[i][0] = String.valueOf(i + 1);            a[i][1] =
String.valueOf(wt[i + 1]);            memory +=
Integer.parseInt(a[i][1]);            a[i][2] =
String.valueOf(val[i + 1]);
        }
        jLabel2.setText("Arbitrary Memory Usage: " + memory);
intermediate += "Solution String: " + selected_string + "\n";
intermediate += "Arbitrary Memory Usage: " + memory;
jTextArea2.setText(intermediate);
        String cols[] = {"#", "Memory", "Data Accessed Times"};
jTable1.setModel(new DefaultTableModel(a, cols));
        // System.out.println();
    }
```

```
   public String[][] default_values(int row, int col) {
String a[][] = new String[row][col];        for (int i =
0; i < a.length; i++) {            for (int j = 0; j <
a[i].length; j++) {                a[i][j] = "0";
         }
      }
      return a;
   }
```



```
package Knapsack; import
java.util.Scanner; public
class knapsack
{
   public void solve(int[] wt, int[] val, int W, int N)
   {
      int NEG = Integer.MIN_VALUE;
int[][] m = new int[N + 1][W + 1];
int[][] sol = new int[N + 1][W + 1];
for (int i = 1; i <= N; i++)
      {
         for (int j = 0; j <= W; j++)
         {
            int m1 = m[i - 1][j];              int
m2 = NEG;               if (j >= wt[i])
m2 = m[i - 1][j - wt[i]] + val[i];
m[i][j] = Math.max(m1, m2);
sol[i][j] = m2 > m1 ? 1 : 0;
```

```
        }
      }
      int[] selected = new int[N + 1];
for (int n = N, w = W; n > 0; n--)
      {
        if (sol[n][w] != 0)
        {
          selected[n] = 1;
          w = w - wt[n];
        }          else
selected[n] = 0;
      }
      System.out.println("\nItems selected : ");
for (int i = 1; i < N + 1; i++)          if
(selected[i] == 1)
          System.out.print(i +" ");
      System.out.println();
   }
   public static void main (String[] args)
   {
      Scanner scan = new Scanner(System.in);
System.out.println("Knapsack Algorithm Test\n");       knapsack
ks = new knapsack();
       System.out.println("Enter number of elements ");
int n = scan.nextInt();         int[] wt = new int[n + 1];
int[] val = new int[n + 1];
       System.out.println("\nEnter weight for "+ n +" elements");
```

```
      for (int i = 1; i <= n; i++)
wt[i] = scan.nextInt();
      System.out.println("\nEnter value for "+ n +" elements");
      for (int i = 1; i <= n; i++)
val[i] = scan.nextInt();
       System.out.println("\nEnter Weight of Memory ");
      int W = scan.nextInt();
ks.solve(wt, val, W, n);
   }
}
package Knapsack;
   public main() {
initComponents();
  }

   @SuppressWarnings("unchecked")
   // <editor-fold defaultstate="collapsed" desc="Generated Code">//GEN-BEGIN:initComponents
private void initComponents() {        jDesktopPane1 = new javax.swing.JDesktopPane();
jMenuBar1 = new javax.swing.JMenuBar();        jMenu1 = new javax.swing.JMenu();
jMenuItem1 = new javax.swing.JMenuItem();
setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON_CLOSE);        setTitle("Single
Binary knapsack Optimizer");
      javax.swing.GroupLayout jDesktopPane1Layout = new
javax.swing.GroupLayout(jDesktopPane1);
jDesktopPane1.setLayout(jDesktopPane1Layout);
jDesktopPane1Layout.setHorizontalGroup(
jDesktopPane1Layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
.addGap(0, 2175, Short.MAX_VALUE)
```

```
        );
        jDesktopPane1Layout.setVerticalGroup(
jDesktopPane1Layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
            .addGap(0, 756, Short.MAX_VALUE)
        );
        jMenu1.setText("File");        jMenuItem1.setText("Process Master");
jMenuItem1.addActionListener(new java.awt.event.ActionListener() {
public void actionPerformed(java.awt.event.ActionEvent evt) {
jMenuItem1ActionPerformed(evt);
            }
        });
        jMenu1.add(jMenuItem1);        jMenuBar1.add(jMenu1);
setJMenuBar(jMenuBar1);        javax.swing.GroupLayout layout = new
javax.swing.GroupLayout(getContentPane());        getContentPane().setLayout(layout);
layout.setHorizontalGroup(
layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
            .addComponent(jDesktopPane1)
        );
        layout.setVerticalGroup(
layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
            .addGroup(layout.createSequentialGroup()
                .addComponent(jDesktopPane1, javax.swing.GroupLayout.PREFERRED_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE)
.addGap(0, 11, Short.MAX_VALUE))
        );
pack();
    }// </editor-fold>//GEN-END:initComponents
    private void jMenuItem1ActionPerformed(java.awt.event.ActionEvent evt) {//GEN-
FIRST:event_jMenuItem1ActionPerformed
```

```
        calc c = new calc();
        c.setVisible(true);
        c.setBounds(0, 0, jDesktopPane1.getWidth(), jDesktopPane1.getHeight());
c.pack();
        jDesktopPane1.add(c);       // TODO add your handling code here:
    }//GEN-LAST:event_jMenuItem1ActionPerformed
    public static void main(String args[]) {            /*
Create and display the form */
java.awt.EventQueue.invokeLater(new Runnable() {
            public void run() {
new main().setVisible(true);
            }
        });
        }
```