DOI: https://doi.org/10.53555/nncse.v2i4.451 **Publication URL: http://nnpub.org/index.php/CSE/article/view/451**

GREEDY ALGORITHM

Abhishek Jain, Manjeet Saini, Manohar Kumar[1]*

*[*1,2,3]CSE Department, MD UNIVERSITY (Dronacharya College of Engineering)*

*[*1]kumarkarnmanohar@gmail.com [2]manjeetsaini60@gmail.com [3]vasujain94@gmail.com*

Corresponding Author: -

Email ID - kumarkarnmanohar@gmail.com

Abstract: -

This paper describes the basic technological aspects of algorithm, algorithmic efficiency and Greedy algorithm. Algorithmic efficiency is the property of an algorithm which relate to the amount of resources use by the algorithm in computer sciences. An algorithm is considered efficient if its resource consumption (or computational cost) is at or below some acceptable level.

Keywords: - *Greedy, Huffman, activity, optimal algorithm*

INTRODUCTION

Algorithmic efficiency are the properties of an algorithm which relate to the amount of resources used by the algorithm in computer sciences. An algorithm must be checked to determine its resource usage. Algorithmic efficiency can be thought of as analogous to engineering productivity for a repeating or continuous process.

For maximum efficiency we wish to minimize resource usage. However, the various resources (e.g. time, space) cannot be compared directly, so which of two algorithms is considered to be more efficient often depends on which measure of efficiency is considered the most important.

An algorithm is considered efficient if its resource consumption (or computational cost) is at or below some acceptable level. Roughly speaking, 'acceptable' means: will it run in a reasonable amount of time on an available computer. Since the 1950s computers have seen dramatic increases in both the available computational power a **Theoretical issues**

In the theoretical analysis of algorithms, the normal practice is to estimate their complexity in the asymptotic sense, i.e. use Big O notation to represent the complexity of an algorithm as a function of the size of the input n. This is generally sufficiently accurate when n is large, but may be misleading for small values of n (e.g. bubble sort may be faster than quicksort when only a few items are to be sorted).

I. Measures of resource usage

The two most common measures are:

- Time: how long does the algorithm take to complete. Analyze the algorithm, typically using time complexity analysis to get an estimate of the running time as a function as the size of the input data. The result is normally expressed using Big O notation. This is useful for comparing algorithms, especially when a large amount of data is too processed. More detailed estimates are needed for algorithm comparison when the amount of data is small (though in this situation time is less likely to be a problem anyway). Algorithms which include parallel processing may be more difficult to analyses
- Space: how much working memory (typically RAM) is needed by the algorithm. This has two aspects: the amount of memory needed by the code, and the amount of memory needed for the data on which the code operates.

II. GREEDY ALGORITHM

A greedy algorithm is an algorithm that follows the problem solving heuristic of making the locally optimal choice at each stage[1] with the hope of finding a global optimum. In many problems, a greedy strategy does not in general produce an optimal solution, but nonetheless a greedy heuristic may yield locally optimal solutions that approximate a global optimal solution in a reasonable time.

For example, a greedy strategy for the traveling salesman problem (which is of a high computational complexity) is the following heuristic: "At each stage visit an unvisited city nearest to the current city". This heuristic need not find a best solution, but terminates in a reasonable number of steps; finding an optimal solution typically requires unreasonably many steps. In mathematical, greedy algorithms solve combinatorial having the properties of Metroid.

III. GREEDY ALGORITHM TO OBTAIN AN OPTIMAL SOLUTION

Geometric distortions manifest themselves as errors in the position of a pixel relative to other pixels in the scene and with respect to their absolute position within some defined map projection. If left uncorrected, these geometric distortions render any data extracted from the image useless. This is particularly so if the information is to be compared to other data sets, be it from another image or a GIS data set.

Rectification

The process of geometrically correcting an image so that it can be represented on a planar surface, conform to other images or conform to a map. That is, it is the process by which geometry of an image is made plan metric. It is necessary when accurate area, distance and direction measurements are required to be made from the imagery. It is achieved by transforming the data from one grid system into another grid system using a geometric transformation.

Rectification is not necessary if there is no distortion in the image. For example, if an image file is produced by scanning or digitizing a paper map that is in the desired projection system, then that image is already planar and does not require rectification unless there is some skew or rotation of the image. Scanning and digitizing produce images that are planar, but do not contain any map coordinate information.

Consider the jobs in the non-increasing order of profits subject to the constraint that the resulting job sequence J is a feasible solution.

In the example considered before, the non-increasing profit vector is

(100 27 15 10) (2 1 2 1) p1 p4 p3 p2 d1 d4 d3 d2

J = {1} is a feasible one

J = {1, 4} is a feasible one with processing sequence (4, 1)

J = {1, 3, 4} is not feasible

J = {1, 2, 4} is not feasible

J = {1, 4} is optimal

Theorem 1: Let J be a set of K jobs and
σ= (i1,i2,….ik) be a permutation of jobs in J such that di1 ≤ di2 ≤…≤ dik.
J is a feasible solution iff the jobs in J can be processed in the order without violating any deadly.
We know di1 ≤ di2 ≤ … dia≤ dib ≤… ≤dik.
Since ia =rb, drb≤dra or dra≥drb.
In the feasible solution dra≥ a drb≥ b
So if we interchange ra and rb, the resulting permutation σ11= (s1, …sk) represents an order with the least index in which σ11 and σ differ is incremented by one.
Also the jobs in σ11 may be processed without violating a deadline.
Continuing in this way, σ1 can be transformed into σ without violating any deadline.
Hence the theorem is proved.

Theorem2 :The Greedy method obtains an optimal solution to the job sequencing problem.
Proof: Let(pi, di) 1≤ I ≤n define any instance of the job sequencing problem.
Let I be the set of jobs selected by the greedy method.
Let J be the set of jobs in an optimal solution.
Let us assume I≠J .
If J C I then J cannot be optimal, because less number of jobs gives less profit which is not true for optimal solution.
Also, I C J is ruled out by the nature of the Greedy method. (Greedy method selects jobs (i) according to maximum profit order and (ii) All jobs that can be finished before dead line are included).

So, there exists jobs a and b such that a∈I, a∉J, b∈J,b∉I.

Let a be a highest profit job such that a∈I, a∉J.

It follows from the greedy method that pa ≥pb for all jobs b∈J,b∉I. (If pb> pa then the Greedy method would consider job b before job a and include it in I).
Let Si and Sj be feasible schedules for job sets I and J respectively.

Let i be a job such that i∈I and i∉J.
(i.e. i is a job that belongs to the schedules generated by the Greedy method and optimal solution).
Let i be scheduled from t to t+1 in SI and t1to t1+1 in Sj.
If t < t1, we may interchange the job scheduled in [t1 t1+1] in SI with i; if no job is scheduled in [t1 t1+1] in SI then i is moved to that interval. With this, i will be scheduled at the same time in SI and SJ.
The resulting schedule is also feasible.
If t1 < t, then a similar transformation may be made in Sj.
In this way, we can obtain schedules SI1 and SJ1 with the property that all the jobs common to I and J are scheduled at the same time.
Consider the interval [Ta Ta+1] in SI1 in which the job a is scheduled.
Let b be the job scheduled in Sj1 in this interval.
As a is the highest profit job, pa ≥pb.
Scheduling job a from ta to ta+1 in Sj1 and discarding job b gives us a feasible schedule for job set J1 = J-{b} U {a}.
Clearly J1 has a profit value no less than that of J and differs from in one less job than does J.
i.e., J1 and I differ by m-1 jobs if J and I differ from m jobs.
By repeatedly using the transformation, J can be transformed into I with no decrease in profit value.
Hence I must also be optimal

IV. APPLICATION OF GREEDY METHOD

- Greedy algorithms mostly (but not always) fail to find the globally optimal solution, because they usually do not operate exhaustively on all the data. They can make commitments to certain choices too early which prevent them from finding the best overall solution later. For example, all known greedy coloring algorithms for the graph coloring problem and all other NP-complete problems do not consistently find optimum solutions. Nevertheless, they are useful because they are quick to think up and often give good approximations to the optimum.
  If a greedy algorithm can be proven to yield the global optimum for a given problem class, it typically becomes the method of choice because it is faster than other optimization methods like dynamic programming. Examples of such greedy algorithms are Kruskal's algorithm and Prim's algorithm for finding minimum spanning trees, and the algorithm for finding optimum Huffman trees.

V. CONCLUSION

Greedy algorithms are usually easy to think of, easy to implement and run fast. Greedy algorithms are infamous for being tricky. Missing even a very small detail can be fatal. But when you have nothing else, they may be the only solution. With backtracking or dynamic programming you are on a relatively safe ground. With greedy instead, it is more like walking on a mined field. Everything looks fine on the surface, but the hidden part may backfire on you when you least expect. While there are some standardized problems, most of the problems solvable by this method call for heuristics. There is no general template on how to apply the greedy method to a given problem, however the problem specification might

give you a good insight. In some cases, there are a lot of greedy assumptions one can make, but only few of them are correct. They can provide excellent challenge opportunities.

REFERENCES

[1].Algorithms Design and Analysis by Udit Agarwal
[2].A.R. Barron. Universal approximation bounds for superpositions of a sigmoidal function. *IEEE Transactions on Information Theory*, 39(3):930–945, 1993.
[3].Y. Freund and R.E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *J. Comput. Syst. Sci.*, 55(1):119–139, 1997.
[4].Jerome Friedman, Trevor Hastie, and Robert Tibshirani. Additive logistic regression: A statistical view of boosting. *The Annals of Statistics*, 28(2):337–407, 2000. With discussion.
[5].T. J. Hastie and R. J. Tibshirani. *Generalized additive models*. Chapman and Hall Ltd., London, 1990.