Advances in Computational Research, ISSN: 0975–3273, Volume 1, Issue 2, 2009, pp-43-46

GA implementation of the multi dimensional knapsack problem using compressed binary tries

Sunanda Gupta* and Garg M.L.
*School of Computer Science and Engineering, Shri Mata Vaishno Devi University, Katra (J&K), India,
sunanda.gupta@smvdu.ac.in, garg.ml@smvdu.ac.in

Abstract- During the last two decades solving combinatorial optimization problems, using genetic algorithms (GA), has attracted the attention of many researchers. The genetic algorithm on which this work is based on uses a special repair operator to prevent the generation of infeasible solutions and to transform each feasible solution into a locally optimal solution. In longer runs it is likely that this algorithm produces candidate solutions that have already been generated and evaluated before. This effect can significantly reduce the algorithm's overall performance. To prevent the reconsideration of already evaluated solutions, a solution based on a Trie is studied. This paper presents the algorithms and data structures for compressing the Binary Trie and incorporates this in the GA implementation of the Multi Dimensional Knapsack Problem.
Keywords- Genetic Algorithms, Tries, Performance.

Introduction

The Multidimensional Knapsack Problem (MKP) is a combinatorial optimization problem that is a generalization of the well-known 0-1 Knapsack Problem. The problem is known to be strongly NP-hard which means that no deterministic polynomial algorithm is supposed to exist to solve the problem [1]. In 1998, Chu and Beasley [1] published a (hybrid) genetic algorithm for heuristically solving larger instances of the MKP, which is still among the best approximate solution approaches. As a major feature it includes a strong repair and local improvement operator which ensures that only promising feasible solutions at the boundary of the feasible region are produced as candidate solutions. The disadvantage of this approach, however, is that in longer runs the same solutions are repeatedly generated and evaluated many times, and valuable CPU-time is wasted. In this work, Chu and Beasley's algorithm is enhanced by using compressed binary tries a special archive to efficiently avoid these re-computations by inserting each solution in the archive before evaluating it. In digital search methods, the binary trie is famous as one of the fastest access methods, and is utilized for a hash table of trie hashing and a dictionary in natural language processing [2],[3]. However, in the case when the binary trie is implemented as a hash table of the trie hashing, if the key sets to be stored are large, the hash table represented by a binary trie is too big to store into main memory. Therefore, it is very important to compress the binary trie into a compact data structure. Then, Jonge et al. [4] proposed the method to compress the binary trie into a compact bit stream (called the pre-order bit stream) by traversing the trie in pre-order. The potential benefits of this enhancement of the genetic algorithm is investigated and discussed. Section 2 presents the method to compress the binary trie into the compact bit stream according to Jonge et al. Section 3 incorporates the algorithm as subroutines in the GA implementation of the Multi Dimensional Knapsack Problem. Section 4 provides the theoretical evaluation. Finally, our conclusion is summarized in section 5.

A Compression Algorithm for Fast Retrieval

A Trie is a data structure that is suitable to store many strings. The name was first suggested in [7]. It is a kind of specialized search tree that makes use of the string representation of the keys to be inserted into the trie. The difference to binary search trees is that no node in the Trie stores the string that is associated with it, but the position of each node relative to the root node determines the string that is represented by a node. In a Binary trie, the binary sequence obtained from the translation of the characters into their binary code, is used as the value of the key. Namely, the left arc is labeled with the value '0' and the right arc with the value '1'. From this reason the binary trie is called the Binary Digital Search Tree (BDS tree). A solution can only be uniquely identified by a leaf node at the lowest level of trie. Even if only one solution is contained in the trie all internal nodes on the path from the root to the leaf corresponding to the solution string are needed to describe the solution. If each of leaves in the BDS-tree points to record of only one key, the depth of the BDS-tree becomes very deep. So as to reduce the depth of the tree, each leaf has the address of the bucket, where some corresponding keys to the path are stored. When the BDS-tree is implemented, the larger the number of registered keys, the greater the number of nodes in the tree is, and more storage space is required. So, Jonge et al.[4] proposed the method to compress the BDS-tree into a very compact bit stream. This bit stream is called pre-order bit stream. The pre-order bit stream consists of three elements: treemap, leafmap and B_TBL. The tree map represents the state of the tree and can be obtained by a pre-order tree

traversal, where '0' is for every internal node and '1' is for every bucket visited. The leafmap represents the state (dummy or not) of each leaf and by traversing in pre-order the corresponding bit is set to '0' if the leaf is dummy, otherwise the leaf is set to '1'. (Empty buckets and their corresponding leaves are called dummies). For example, let us consider that the following key set B is to be inserted into the BDS-tree.

B = {cat, bat, job, run, see, son, yak, ink, lap, get}

If the binary sequence obtained from the translation of the internal node of each character, where internal codes of a,b,.....z are 1,2,...26 respectively, into binary numbers of 5 bits is used, the corresponding bit strings to be registered are as follows.

cat → 00011 / 00001 / 10100
bat → 00010 / 00001 / 10100
job → 01010 / 01111 / 00010
run → 10010 / 10101 / 01110
see → 10011 / 00101 / 00101
son → 10011 / 01111 / 01110
yak → 11001 / 00001 / 01011
ink → 01001 / 01110 / 01011
lap → 01100 / 00001 / 10000
get → 00111 / 00101 / 10100

If B_SIZE is 2, the corresponding BDS-tree for the key set B is shown in Fig 1. B_SIZE is used to denote the number of keys and records that can be stored in one bucket.

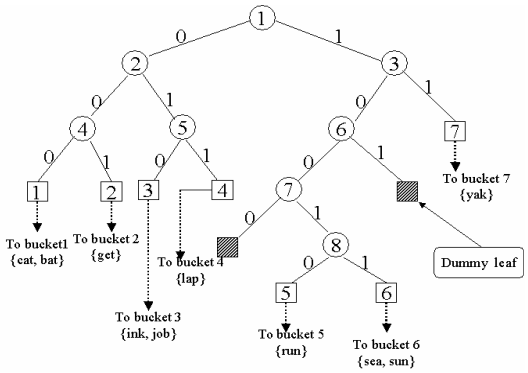

Fig. 1- The BDS-tree for key set B

In order to compress the BDS-tree, a particular leaf is applied, which does not have any addresses for the bucket. This leaf is called dummy leaf. Dummy leaf is introduced because of the following advantages. First, it satisfies the property of binary trees that the number of leaves is one more than the number of internal nodes. This property underlies the search algorithm using the compact data structure. Secondly, if the search terminates in a dummy leaf, the search key is regarded as a key that does not belong to the BDS-tree and no disk access will be needed at all. Fig 2. shows the preorder bit stream corresponding to the BDS-tree of Fig 1. In order to understand the relation between the BDS-tree and the pre-order bit stream easily, we indicate above the treemap the corresponding internal node and leaf number (in the case of the dummy leaf, the number is "d") within the round "()" and square "[]" brackets respectively.All the internal nodes are represented by "0" and all the leaf nodes in a tree map are represented by "1". All the leaves except for the dummy leaves are represented by "1" in leafmap. The search using the pre-order bit stream proceeds bit by bit from the first bit of treemap going to the right, so that, the search is done, traversing the BDS-tree in pre-order.

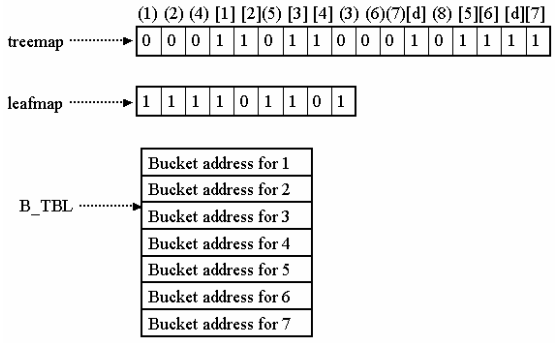

Fig. 2- Pre-order bit stream

The retrieval algorithm using the pre-order bit stream as proposed by Jonge et al. [4] is presented below.

BDS_RETRIEVE
Step S-1: {Initialization}
keypos → 1;
treepos → 1;
leafpos → 1;
Step S-2: {Verification of bit value of the key}
If the bit pointed by keypos is '1' proceeding to Step (S-3) otherwise proceed to Step (S-5).
Step S-3: {Skipping left subtree in treemap}
Advance treepos until the number of '1' bits in treemap is one more than the number of '0' bits and proceed to Step (S-4).
Step S-4: Advance leafpos by the number of '0' bits skipped in treemap from the current treepos.
Step S-5: {Loop invariant until reaching bucket}
Advance treepos by one, and if the bit pointed to by treepos is '0' proceed to step (S-2) after advancing keypos by one.
Step S-6: {Verification of leafmap}
If the bit of leafmap pointed to by leafpos is '0', FALSE is returned.
Step S-7: {Verification of B_TBL}
Count the number of '1' bits in leafmap from the first bit of leafpos and obtain the bucket number indicated by the counted value. If the bucket indicated by bucketnum contains the key return TRUE, otherwise return FALSE.
In the above algorithm the following abbreviations have been used.
keypos: A pointer to the current position in s_key.

treepos : A pointer to the current position in treemap.
leafpos: A pointer to the current position in leafmap.

GA based MKP incorporating binary tries
In longer runs it is likely that the genetic algorithm produces candidate solutions that have already been generated and evaluated before. This effect can significantly reduce the algorithm's overall performance. This is because if the majority of the individuals share the same value for a chromosome, the chromosome is said to have converged and if all the chromosomes have converged, the population is said to have converged [5][6]. To prevent the reconsideration of already evaluated solutions, a solution archive based on a trie is studied. Each newly generated solution is inserted into this archive. If during insertion into the archive a solution is recognized to be a duplicate of an already visited solution then it is discarded. An approach that detects duplicates not only within the current population but also among all chromosomes that have been generated, has not yet appeared in literature to my knowledge. Thus, the genetic algorithm for the multidimensional knapsack problem incorporating compressed binary trie is presented below.

GA_MKP_TRIE
Step 1: Set t: =0;
Step 2: P (t): = $\{S_1,S_2,....S_M\}$, such that $S_i = \{J_k$ where $1 \le k \le n \mid J_k = \{0,1\}\}$
Step 3: Call subroutine BDS_INSERT for creating a trie consisting of all the chromosomes of the population generated.
Step 4: Evaluate P(t) := $\{f(S_1),.....f(S_M)\}$;
where $f(S_i) = \sum$ fitness J_i where $1 \le i \le n \mid$ only if $J_i = 1\}$
Step 5: Find $S^* \in P(t)$ such that $f(S^*) \ge f(S)$ for all $S \in P(t)$
Step 6: while $t < t_{max}$ do
Step 7: Select $\{P_1,P_2\}$: = Φ (t);
/* Φ = binary tournament operator */
Step 8: Crossover C: = $\Omega_c (P_1,P_2)$
/* Ω_c = uniform crossover operator*/
Step 9: Mutate C $\leftarrow \Omega_m(C)$
/* Ω_m = mutation operator */
Step 10: Evaluate f(C)
Step 11: If f(C) is unfit then C $\leftarrow$ GreedyRepair(C)
Step 12: Call subroutine BDS_RETRIEVE
If it returns true i.e C $\equiv$ any S $\in$P(t) then discard C and go to step 7
else
call subroutine BDS_INSERT for inserting the new key into the trie.
Step 13: end if
Step 14: find S' $\in$P (t) such that $f(S') \le f(S)$ for all $S \in P(t)$ and replace S' $\leftarrow$ C
/*steady state replacement*/
Step 15: if $f(C) > f(S^*)$ then
Step 16: $S^* \leftarrow C$
Step 17: end if
Step 18: t $\leftarrow$ t+1
Step 19: end while
Step 20: return S*, $f(S^*)$

As it is evident from the algorithm, BDS_INSERT is a subroutine that has been used for creating the archive for storing all the chromosomes that are generated in a population. The algorithm for creating a binary trie using BDS_INSERT is presented below. The GA_MKP_TRIE algorithm follows steady state replacement strategy, as the initial population created is evolved, by replacing single chromosomes by newly generated ones. Only one child individual is created at a time and this individual replaces the worst individual in the population. GreedyRepair operator designed by Chu and Beasley [1] is used. It consists of two phases. The first part called the DROP phase ensures that every solution that was processed by this DROP phase is feasible. Each variable is examined in ascending order of utility ratios and as long as solution is infeasible the current item examined is excluded from the solution if it was included. The second part called the ADD phase, examines all items in decreasing order of utility ratios and add each item that is not included in the solution as long as no resource constraint gets violated. BDS_RETRIEVE is the compression algorithm which actually checks whether the new chromosome generated is a duplicate or not. It has been explained well in detail in section 2.

BDS_INSERT
The method for inserting the new key into the BDS-tree is divided into following three cases: -

Case A) Required Bucket is partially filled

i. The required bucket is read in.
ii. The new record inserted into it.
iii. Bucket is rewritten to disk.
iv. COUNT incremented by 1. (COUNT keeps the check that the number of keys should not exceed B_SIZE)

Case B) Required Bucket is a Dummy.

i. Dummy bucket is converted into real one.
ii. Dummy buckets don't have disk space allocated to them, inserting a record in a dummy bucket will require allocating a new disk bucket.
iii. Initializing COUNT to 1
iv. New bucket included in B_TBL at the appropriate position.

Case C) Required Bucket is full

i. Bucket must be split into two buckets.
ii. All b+1 keys (the b keys that previously filled the bucket, plus the new one to be inserted) are distributed over the two new buckets.

Theoretical Evaluation

A. *Storage Requirements*

Bucket addresses are stored in a separate table, indexed by bucket number. Fig.2 shows the linear representation of the tree of fig.1. A bucket number is just the position of its leaf bit in the linear representation when zeros (internal nodes) are neglected, whereas the bucket address is a physical or symbolical address. Associated with each tree is a leafmap with as many bits as the tree has leaves. For each leaf (bucket) the corresponding bit is set to zero if the bucket is empty, otherwise the bit is set to 1. When a tree search terminates, the bucket number of the bucket found is used to index into the leafmap to fetch the corresponding bit. If that bit is a '1', the bucket found exists; otherwise the bucket is a dummy. It is straightforward to calculate the number of bits required in the index per bucket in the file. If a tree has N buckets (including Dummies), it will have 2N-1 nodes total, and thus 2N-1 bits are needed for its linear representation. Thus, each leaf (bucket) requires approximately 2 bits. If the leaf map is not used, then every bucket and every dummy has a slot in B_TBL. In B_TBL dummies will have 0's and buckets will contain disk addresses. If the number of bits required for representing the address in each bucket is A, then the number of bits required per bucket is A+2. If a fraction *d* of all the buckets are dummies, the number of bits per nondummy bucket is (A+2) / (1-*d*). If the leafmap is used, the number of bits per leaf is increased from 2 to 3, but only buckets need a slot in the table for their disk addresses, so the total number of bits per bucket is A+3 / (1-*d*). Clearly, the leafmap scheme is to be preferred whenever:

$$(A+3) / (1-d) < (A+2) / (1-d)$$

which occurs whenever $d > 1/A$.

B. *Time Effeciency*

The computational cost of the retrieval algorithm BDS_RETRIEVE is linear in the size of the tree. During a search the algorithm reads from the linear representation all bits up to and including the 1 bit representing the bucket (dummy or not) finally found. The retrieval algorithm spends its time mainly on skipping subtrees. This means that on the average about half the bits of the linear representation will be read. Clearly, about half the bit map will be scanned on the average. On the other hand, if no trie structure is maintained as in case of algorithm given by chu and Beasley [1] then searching for a duplicate in a generation where the chromosomes are stored in a file would require sequential access i.e. reading the entire file sequentially.

Conclusion

In this paper, a genetic algorithm enhanced with binary tries used for detecting duplicates is incorporated for solving the multi dimensional knapsack problem. Chu and Beasley's [1] algorithm also does duplicate elimination, however it detects duplicates that are contained in the population at the time of generation of the duplicate solution. It is however possible and not so unlikely that a candidate solution that was replaced by a different solution is generated again in later iteration. This kind of duplicate occurrence has been detected with the help of compressed binary tries. Trie based archive BDS_INSERT represents the entire search space for the MKP. This enables more opportunities how to handle the detection of a duplicate and that too in minimum time. As future improvements, the GA_MKP_TRIE should be modified in such a manner that in case of detection of duplicate solution, it generates alternative unvisited solution (from the created duplicate solution) that is not contained in BDS archive.

References

[1] Chu P.C.and Beasley J.E. (1998) *Journal of Heuristics*, 4(1), pp 63-86.

[2] Aoe J. (1991) *IEEE Computer Society Press*

[3] Gonnet G.H. (1984) *Handbook of Algorithms and Data Structures, Addison-Wesley, Reading Mass. Ch. 3 (Searching Algorithms*), pp 25-147.

[4] Jonge W.D., et.al (1987) *IEEE Trans. Software Engineering*, SE-13 (7), pp. 799-809.

[5] Gottlieb J. (2000) *AE '99 Selected Paqpers from the 4th European Conference on Artificial Evolution, volume 1829 of Lecture Notes in Computer Science, pp 23-37. Springer-Verlag (2000).*

[6] Freville A. (2004) *European Journal of Operation Research*, 127(1), pp1-21.

[7] Fredkin E. (1960) *Trie Memory; Communication of the ACM*, 3(9). 490-499.