A Binary Search Algorithm for the Bottleneck Problem of Distinct Representatives

D. Magos

Department of Informatics, Athens University of Economics and Business, Athens, Greece

Binary Search is considered to be one of the most effective and easy-to-use searching techniques. In the current work, an algorithm for the Bottleneck problem of Distinct Representatives based on binary search is presented. Application of the algorithm to the general 0–1 Minimax problem is straightforward. Computational experience including instances with up to 160 000 variables is reported.

Keywords: Binary Search, Minimax problem, Distinct Representatives, Bottleneck Matching.

1. Introduction

Given m sets $S_1, S_2, .., S_m$, not necessarily distinct, a collection $A = (a_1, a_2, .., a_m)$ with $a_i \in S_i$, $a_i \neq a_j$, $\forall\, i \neq j$, is called a *System of Distinct Representatives* (SDR). We say that a_i is the representative of the set S_i. Let $\mathbf{S} = \bigcup i = 1^m S_i$ and $|\mathbf{S}| = n$, where $|\mathbf{S}|$ denotes the cardinality of $\mathbf{S}$. Then we refer to each element of $\mathbf{S}$ through the index i, $i \in I$, $I = \{1, .., n\}$. Accordingly, we refer to the set S_j through the index j, $j \in J$, $J = \{1, .., m\}$. Obviously, it is necessary that $m \leq n$ for an SDR to exist. A necessary and sufficient condition on the existence of an SDR is given by P. Hall's theorem (HALL (1935)). Suppose that each $i \in S_j$ represents the set S_j at a given cost c_{ij}. The *Bottleneck* SDR (BSDR) problem deals with finding the SDR with the cost of the most expensive representative being as small as possible. The problem can be expressed in integer programming format by using a binary variable x_{ij} which takes the value of 1 if element i represents the set S_j and 0 otherwise, as follows:

$$
\begin{aligned}
\text{(BSDR)} \qquad & \min z \qquad s.t. \\
& c_{ij} x_{ij} \leq z, \\
& \sum_{i \in S_j} x_{ij} = 1, \qquad \forall\, j \in J, \\
& \sum_{j \in Q_i} x_{ij} \leq 1, \qquad \forall\, i \in I, \\
& x_{ij} \in \{0, 1\},
\end{aligned}
$$

where $Q_i = \{j \in J \mid i \in S_j\}$.

Alternatively, BSDR can be stated as the *Bottleneck Weighted Matching problem* on a bipartite graph $G(X, Y, E)$. Let (X, Y) be a bipartition of G where vertices of X correspond to the sets S_j and vertices of Y to the elements of $\mathbf{S}$. An edge e_{ij} ($e_{ij} \in E$) connects a set-vertex S_j to an element-vertex i if the element belongs to the set. A weight is associated with every edge of the graph. Find the maximum matching (if one exists) with the property that the maximum weight of all edges participating in the solution is minimized.

A real-life application results from having n operators available to carry out m jobs. Each operator can perform only a specified number of jobs. It takes c_{ij} time units for operator j to perform job i. It is required to assign the operators to jobs, possibly with some operators left unassigned, so that the completion time for all jobs is minimized.

BSDR can be reduced to the bottleneck assignment problem by adding $n - m$ *dummy* vertices to set X so that the number of vertices on both sets (X, Y) are equal. Each dummy node is connected to all nodes of set Y through edges with zero weight.

BSDR can be solved either by an algorithm for the bottleneck assignment problem (MARTELLO and TOTH (1987)) or by a general purpose minimax algorithm (GARFINKEL and NEMHAUSER (1972), JORGENSEN and POWEL (1987)). In the current work, we present an algorithm for BSDR which can be easily extended to the general minimax problem. In the following sections we describe the algorithm and some modifications that may have a great impact on the performance. Computational experience is included.

2. The algorithm

Binary search is widely used for effectively searching a sorted set of data. The same principle can be applied for finding the optimum of a *minimax* problem. We consider an upper (z^{UB}) and a lower (z^{LB}) bound on the value of the solution. At each iteration the gap between z^{UB} and z^{LB} is decreased by a factor of 2. A threshold value $z^T H$ is calculated equal to the average value of the upper and lower bound. For each (i, j) with $c_{ij} > z^{TH}$ the corresponding x_{ij} is set to zero. A feasible solution is sought through the remaining set of variables. If such a solution exists then z^{UB} is updated. Otherwise, if no feasible solution exists, all previously deleted edges are re-established and z^{LB} is set to z^{TH}. The next iteration starts by re-calculating z^{TH}. The algorithm terminates when $z^{UB} = z^{LB}$. At termination z^{UB} is the value of the optimum. An initial value for z^{UB} can be obtained by finding a feasible solution for the problem with all variables free. Alternatively, an initial upper bound is given by:

$$(1) \qquad z^{UB} = \max\{c_{ij}, \quad \forall i \in I, j \in J\}.$$

An initial lower bound value is:

$$(2) \qquad z^{LB} = \max\{\min_i\{c_{ij}\}, \quad \forall j\}.$$

The algorithm in mock Pascal format is listed next. Text in { } is commentary.

Binary Minimax Algorithm

```
Initialize;
While (z^UB − z^LB > 1) do
  begin
  Attempt to find a feasible solution;
  If solution feasible then
    begin
    {z* : the value of the feasible solution}
    z^UB = z*;
    If ((z^UB − z^LB) > 1) then
      begin
      z^TH = (z^UB + z^LB)/2;
      If (c_ij > z^TH) then x_ij = 0;
      end
    end
    else {if no feasible solution exists}
    begin
      z^LB = z^TH
      If ((z^UB − z^LB) > 1) then
        begin
        z^TH = (z^UB + z^LB)/2;
        If ((c_ij ≤ z^TH) and (x_ij = 0))
                then set x_ij free;
      end
    end {end if then else}
  end {end while}
```

Example

Consider the sets $S_1 = \{1, 2, 3, 4\}$, $S_2 = \{1, 2, 3, 4\}$, $S_3 = \{2, 3, 5\}$, $S_4 = \{2, 3, 4, 5\}$, $S_5 = \{1, 4, 5\}$. The corresponding cost coefficients are given in Table 1. Elements correspond to rows (i index) and sets to columns (j index). If element i cannot represent set S_j then $c_ij = \infty$. This is so when either $i \notin S_j$ or when at an iteration variable x_{ij} is set to zero due to the test $c_{ij} > z^{TM}$. At each iteration the solution is illustrated in terms of cells marked with $(*)$.

Iteration 1

Initial bounds according to (1) and (2) are $z^{UB} = 12$, $z^{LB} = 4$. A feasible solution is $x_{ij} = 1, \forall i = j$.

Table 1

3*	6	∞	∞	6
6	7*	3	6	∞
5	4	10*	5	∞
9	2	∞	6*	7
∞	∞	12	2	9*

The new values are: $z^* = 10 \Longrightarrow z^{UB} = 10$, $z^{LB} = 4$. Therefore $z^{TH} = 7$.

Iteration 2

Table 2

3	6	∞	∞	6*
6	7	3*	6	∞
5*	4	∞	5	∞
∞	2*	∞	6	7
∞	∞	∞	2*	∞

$z^* = 6 \Longrightarrow z^{UB} = 6$, $z^{LB} = 4 \Longrightarrow z^{TH} = 5$.

Iteration 3

Table 3

3	∞	∞	∞	∞
∞	∞	3	∞	∞
5	4	∞	5	∞
∞	2	∞	∞	∞
∞	∞	∞	2	∞

Set S_5 (last column) cannot be represented by any element at the current iteration. The problem is infeasible since no element can represent set S_5. The lower bound is updated: $z^{LB} = 5$. The algorithm terminates since $z^{UB} - z^{LB} = 1$. The value of the optimum is 6.

At each iteration a feasible solution is produced (if one exists) by the *hungarian* method (BONDY, MURTY (1976)). The proposed algorithm is polynomial since both finding a feasible solution (by using the hungarian method) and binary search are polynomial. To prove that, we recall the definitions of X, E when stating BSDR as the *Bottleneck Weighted Matching problem*. The complexity of the hungarian method is $O(|X| \cdot |E|)$ (PAPADIMITRIOU and STEIGLITZ (1982)) and of binary search is $O(\log_2 |E|)$ (SEDGEWICK (1990)).

The algorithm is implemented under the assumption of integer costs. In the case that real costs are used 1 must be replaced by a tolerance ε $(1 > \varepsilon \geq 0)$ in the condition $(z^{UB} - z^{LB}) > 1$.

3. Modifications and Variations

The algorithm proposed can be employed to solve other Minimax problems as well. In the light of this generalization several modifications can be made, their success depending on the structure the problem solved each time.

Employing Heuristics

At each iteration, instead of finding just a feasible solution a heuristic can be employed so as to find a "good" feasible solution (z^{UB}). If heuristic rules are employed, it is expected that the time spent at each iteration to increase. This can be outweighed by a decrease in the number of iterations if sharp bounds are produced. In our implementation we have used the greedy notion: at the process of forming a feasible solution always select first the free variable with the smaller c_{ij}.

Reducing the number of iterations

The disadvantage of binary search is that it spends several iterations with the gap between z^{UB} and z^{LB} being really small. This can be overcome by establishing a critical value for the gap. If the gap is smaller than this value and several sequential iterations have not produced a feasible solution our belief that z^{UB} is the optimum grows. Then, instead of setting $z^{TH} = (z^{UB} + z^{LB})/2$ we set $z^{TH} = z^{UB} - 1$. If an infeasible solution is produced then $z^{LB} = z^{TH}$ and the algorithm terminates immediately. The above mechanism is implemented in terms of

Table 4

Average CPU times								
m \ n	50	100	150	200	250	300	350	400
50	0.08	0.067	0.072	0.097	0.121	0.130	0.152	0.176
100		0.227	0.158	0.186	0.231	0.268	0.302	0.332
150			0.435	0.315	0.343	0.400	0.462	0.502
200				0.691	0.496	0.547	0.630	0.676
250					1.027	0.711	0.752	0.842
300						1.352	0.993	1.072
350							1.722	1.293
400								2.093

percentages. Let π be a number between 0 and 1. Each time the algorithm fails to find a feasible solution p is calculated:

$$p = (z^{UB} - z^{LB})/z^{UB}.$$

Then p is compared to π. If $p < \pi$ then set $z^{TH} = z^{UB} - 1$. π can be thought of as a measure of our belief that the current solution is the optimum or that the optimum value is close to the current upper bound. Two strategies can be used with respect to the values of π: either use a constant value through every iteration or use a sequence of values. We have implemented the first option.

The idea of prematurely attempting to terminate the search can also be implemented by using a divisor δ when computing the value of z^{TH} (i.e. $z^{TH} = (z^{UB} + z^{LB})/\delta$). Note that z^{TH} is a decreasing function of δ.

Proposition δ takes values in the interval $[(z^{UB}+z^{LB})/(z^{UB}-1), (z^{UB}+z^{LB})/(z^{LB}+1)]$.

Proof.

$$z^{TH} \leq z^{UB} - 1 \Longrightarrow (z^{UB}+z^{LB})/\delta < z^{UB} - 1$$
$$\Longrightarrow (z^{UB} + z^{LB})/(z^{UB} - 1) \leq \delta$$
$$z^{LB}+1 \leq z^{TH} \Longrightarrow z^{LB}+1 \leq (z^{UB}+z^{LB})/\delta$$
$$\Longrightarrow \delta \leq (z^{UB} + z^{LB})/(z^{LB} + 1) \blacksquare$$

One can set high or low values on δ, within these bounds, according to his (hers) belief on the optimality of the current solution. In the case of binary search $\delta = 2$.

4. Computational Considerations

A set of 288 problems with m sets and n elements ($m = 50, 100, 150, \ldots, 400$, $n = 50, 100, 150, \ldots, 400$ and $m \leq n$) was produced with the following characteristics. Eight problems were produced for each combination of m and n with the number of variables at each problem being $m \times n$ (i.e. each set contains all the elements). Therefore the size of the problems solved ranges from 2 500 to 160 000 variables (400 sets and 400 elements). The costs were integers sampled from a uniform distribution in the interval $[1, 1e + 06]$. The algorithm was implemented in C and run on a VAX 6320 under ULTRIX-32 (vs 4.2). All arithmetic was carried out with "long" integers. Table 4 displays the average CPU times in seconds for each combination of m and n.

Results show that the algorithm can be used for real time applications involving a great number of variables. Table 4 shows that even big instances (160 000 variables) are solved in a few seconds.

The number of iterations is limited by the complexity of binary search ($O(log_2 k)$), where k is the number of elements through which the search is carried out). In our case the elements are the variables of the problem (or the edges of the bipartite graph). Therefore for the problem set used the number of iterations is $\log_2 m \times n$. This leads to 11.28 ($log_2 2500$) iterations for small problems (50 sets, 50 elements) and to 17.28 ($log_2 160\,000$) iterations for big instances (400 sets, 400 elements). These bounds stand for the worst case. In practice we noticed that no

problem needed more than 12 iterations. This is a result of the greedy rule which allowed "good" feasible solutions to be found and of the premature termination of the search through parameter π. After some experimentation we have found that a good value for π is 0.35. This is the value used in our implementation.

It is important to note that for the same number of sets, say m, an increase in the number of elements (therefore in the number of variables) does not necessarily cause an increase in CPU time. For instance the average time for solving the eight problems with $m = 200$ and $n = 200$ is 0.691 secs. Increasing n to 250 leads to a decrease of CPU time to 0.496. Then the time increases gradually following the increase in the number of elements (n). The same observation stands for every value of m tested. This behaviour can be explained by the fact that as the number of elements increases, the task of producing a feasible solution at each iteration becomes easier because more elements can represent the same number of sets. The results of this effect gradually disappear as the number of elements increases beyond a certain limit.

References

J. A. BONDY, U. S. R. MURTY (**1976**) *Graph Theory with Applications*, Macmillan.

R. S. GARFINKEL, G. L. NEMHAUSER (**1972**) *Integer Programming*, John Wiley & Sons.

P. HALL (**1935**) On representatives of subsets. *Journal London Mathematical Monthly* 63, 26–30.

C. JORGENSEN, S. POWEL (**1987**) Solving 0–1 minimax problems. *Journal of Operational Research Society* 38, 6, 515–522.

S. MARTELLO, P. TOTH (**1987**) Linear assignment problems. In *Surveys in combinatorial optimization* (S. MARTELLO, G. LAPORTE, M. MINOUX, C. RIBEIRO eds), *Annals of Discrete Mathematics* 31, 259–282.

C. H. PAPADIMITRIOU, K. STEIGLITZ (**1982**) *Combinatorial Optimization, Algorithms and Complexity*. Prentice-Hall, Englewood Cliffs, New Jersey.

R. SEDGEWICK (**1990**) *Algorithms in C*, Addison & Wesley.

Received: March, 1993
Accepted: November, 1993

D. MAGOS graduated from the Athens University of Economics and Business in 1987 (B.Sc. on Computers Science and Statistics). He continued his studies at the LSE (department of Operational Research) where he received his M.Sc. in 1988. He is currently working on his Ph.D. thesis at the Athens University of Economics. His main interests lie in the fields of mathematical programming, combinatorial optimization and optimization algorithms.

Contact address:
D. Magos
Department of Informatics
Athens University of Economics and Business
76 Patission Str, 104 34 Athens, Greece
Tel: 01-8225 268
E-mail: nnk@aueb.ariadne-t.gr