An IDA* Algorithm for Optimal Spare Allocation

Michail G. Lagoudakis*
Center for Advanced Computer Studies
University of Southwestern Louisiana
Lafayette, LA 70504

Optimal Spare Allocation, IDA* search, TreadMarks™.

Abstract

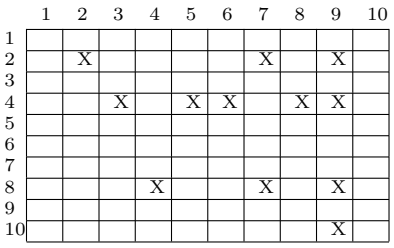
A new algorithm for the Optimal Spare Allocation problem in reconfigurable arrays is presented. It is based on a previously published branch and bound search method [4]. Enhanced with a strong heuristic function, it yields an effective A* search, which performs more efficiently in the iterative deepening implementation (IDA*). A parallel implementation of the algorithm for a distributed shared memory machine using TreadMarks™ was used for evaluation. The results demonstrate the efficiency of the scheme and show how computationally intensive problems can be handled by appropriate domain heuristics and cooperative distributed computational power.

1 The Optimal Spare Allocation Problem

The Optimal Spare Allocation (OSA) problem (or equivalently Vertex Cover in Bipartite Graphs) [4] is an NP-HARD problem arising in fault tolerant computing. It deals with the optimal allocation of spare rows and columns over a two-dimensional array of cells where some cells are faulty. The cells can be processors, memory units or VLSI components in practical applications. A number of spare rows and a number of spare columns are given, with two costs associated to spare rows and columns respectively. One spare row (column) can replace any row (column) in the array, repairing all the faulty cells along this row (column). The purpose is to repair all the faulty cells using spares, with the minimum possible overall cost.

The sample instance [4] given below, consists of a (10×10) array with 12 faulty cells. There are 3 spare rows with a cost of 8 associated to each one of them and 3 spare columns with a cost of 15 associated to each one of them. An optimal solution with a cost of 39 would include the spare row allocations {4, 2, 8} and the spare column allocation {9}.

*Currently with the Department of Computer Science, Duke University, Durham, NC 27708; Email: mgl@cs.duke.edu.



2 Previous Work

Kuo and Fuchs [4] proposed an algorithm for the OSA problem with three steps: (1) Must-Repair Analysis, (2) Early-Abort Analysis, and (3) Final Analysis.

The Must-Repair Analysis repairs rows and columns for which there is no other choice. A *must-repair row* is a row, where the number of faulty cells is greater than the number of available spare columns. A *must-repair column* is a column, where the number of faulty cells is greater than the number of available spare rows. Clearly, the faulty cells in a must-repair row (column) can be repaired only by a spare row (column). This eliminates many faulty cells and may cause an early abort in the absence of the required spares.

The Early-Abort Analysis calculates a lower bound to the number of spares required to repair the remaining faulty cells, using a polynomial graph-theoretic algorithm for maximum matching. It aborts if the array cannot be repaired.

The Final Analysis is a branch and bound search in the space of partial solutions. Each node records the spare row and column assignments and the (so far) cost for the partial solution it represents. A node is expanded by selecting an uncovered faulty cell and covering it by either a spare row or a spare column, creating at most two children nodes. The algorithm selects always the node with the minimum cost for expansion (uniform cost search) to ensure optimality.

The basic limitations of this algorithm are summarized in the following: (1) The must-repair analysis is performed only once, although it could eliminate dead-end branches of the tree if applied at each node. (2) The early-abort analysis could be used at each node as a heuristic for informed search. However, it does not take into account the costs of the spares and thus it is not the best possible. (3) The final analysis is uninformed and, since pruning is not applied, it will lead very soon to exponential explosion. Moreover, due to their implementation, a solution path cannot have a length less than the total number of faulty cells remaining after the must-repair analysis.

3 The Algorithm

The new algorithm attempts to overcome the limitations above. Its heart is a heuristic cost function that gives an (optimistic) estimate of the cost from a node (partial solution) n to the complete solution. This estimate added to the so far cost of n gives an estimate of the total cost to the solution through n. This estimated cost is used for guidance of the search and pruning of the tree.

The heuristic function requires the following data with respect to the node under consideration:

CFR (Current Faulty Rows) : The number of rows that contain faulty cells.
CFC (Current Faulty Columns) : The number of columns that contain faulty cells.
$FCRD_i$ *(Faulty Cell Row Distribution)* : The number of faulty cells in a faulty row, for i = 1,...,*CFR*. The values are sorted in descending order.
$FCRC_i$ *(Faulty Cell Row Coverage)* : The number of faulty cells that can be covered in the *best* case given i spare rows, for i = 1,...,*CFR*. Obviously, $FCRC_i = \sum_{k=1}^{i} FCRD_k$.
$FCCD_j$ *(Faulty Cell Column Distribution)* : Similar to $FCRD_i$, but for the columns.
$FCCC_j$ *(Faulty Cell Column Coverage)* : Similar to $FCRC_i$, but for the columns.
TFR (Totally Faulty Rows) : The number of faulty rows that have a number of faulty cells equal to the total number of faulty columns.
TFC (Totally Faulty Columns) : Similar to *TFR*, but for the columns.
CTFC (Current Total Faulty Cells) : The total number of the uncovered faulty cells.
SR (Spare Rows) : The number of available spare rows.
SC (Spare Columns) : The number of available spare columns.
SRC (Spare Row Cost) : The cost of a spare row.
SCC (Spare Column Cost) : The cost of a spare column.

The idea runs as follows: we iterate over all possible combinations of x spare rows and y spare columns that *could* cover *all* the uncovered cells in the *best* case. For each pair the cost is calculated and the minimum is returned (to avoid overestimation).
The first loop iterates over x and the second over y up to the values $maxx = \min(CFR, SR)$ and $maxy = \min(CFC, SC)$.

The extreme cases (x=0 or CFR, y=0 or CFC) are straightforward. In all other cases, the corresponding y's and x's are calculated using

$$\arg\min_y \left(\sum_{k=1}^{y} cov_x(k) \geq rem_x \right), \ \arg\min_x \left(\sum_{k=1}^{y} cov_y(k) \geq rem_y \right) \quad (1)$$

$$cov_x(k) = \min(CFR - x, \ FCCD_k - \min(TFR, x)) \quad (2)$$

$$cov_y(k) = \min(CFC - y, \ FCRD_k - \min(TFC, y)) \quad (3)$$

$$rem_x = CTFC - FCRC_x, \quad rem_y = CTFC - FCCC_y \quad (4)$$

The remaining uncovered cells for x selected spare rows (or y spare columns) are given by eq. 4, whereas eq. 2 (or 3) delineates how they would be covered *in the best case* as spare columns (rows) are added. Finally, the minimum y (or x) that satisfies the inequality in eq. 1 is returned to form the pair (x,y), whose validity is subject to

$$0 \leq x \leq SR \text{ and } 0 \leq y \leq SC \quad (5)$$

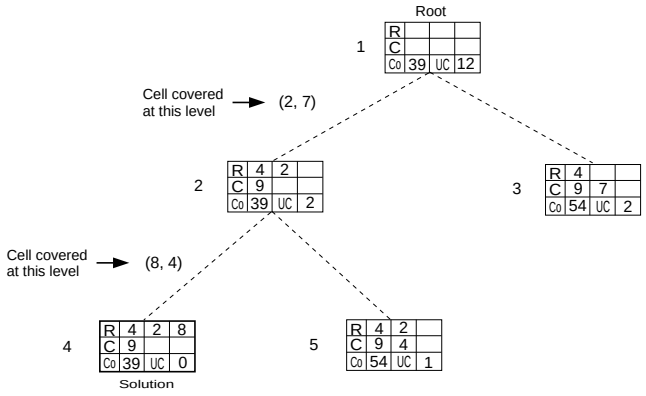
and its cost is

$$cost[(x, y)] = SRC \times x + SCC \times y. \quad (6)$$

HEURISTIC FUNCTION H
if ($CTFC = 0$) **then return** 0
Set $ESTIMATE \leftarrow \infty$
for $x \leftarrow 0$ **to** $maxx$ **do**
  if ($x = 0$) **then** $y \leftarrow CFC$
  elseif ($x = CFR$) **then** $y \leftarrow 0$
  else calculate y using eqs. 1, 2, 4
  if ((x, y) is a valid configuration (eq. 5) and ($\text{cost}[(x, y)] < ESTIMATE$))
  then $ESTIMATE \leftarrow \text{cost}[(x, y)]$
for $y \leftarrow 0$ **to** $maxy$ **do**
  if ($y = 0$) **then** $x \leftarrow CFR$
  elseif ($y = CFR$) **then** $x \leftarrow 0$
  else calculate x using eqs. 1, 3, 4
  if ((x, y) is a valid configuration (eq. 5) and ($\text{cost}[(x, y)] < ESTIMATE$))
  then $ESTIMATE \leftarrow \text{cost}[(x, y)]$
return *ESTIMATE*

Other heuristics used are the following: (1) If a partial-solution node has no spares available, it is pruned. (2) If the estimated cost of a node is greater than what the available spares of the node can afford, it is pruned. (3) If the node contains must-repair rows (columns) that cannot be repaired it is pruned. (4) The node with the minimum cost has the highest priority for expansion. Among nodes with the same cost, the node with the fewest uncovered faulty cells has the highest priority. (5) The next cell to be covered when a node is expanded is the one that resides on the row and column that aggregately have the maximum number of faulty cells. This reduces the size of the tree from the very early steps.

With all the heuristics above, our algorithm is an informed search procedure along the lines of A* search [6]. However, the large number of nodes that must be kept ordered in the memory causes early memory overflows and adds significant overhead for sorting, as the problem instances grow. An iterative deepening implementation IDA* ([3],[6]) overcomes these limitations with the cost of repeating expansions. Nevertheless, its overall performance is much better and seems to be more suitable for the OSA problem.



The resulting search tree for the sample instance is given above (R/C=Row/Column Allocations, Co=Cost, UC=Uncovered Cells). For this instance, Kuo-Fuchs algorithm creates 11 nodes, whereas a previous algorithm by Day [2] creates 18. Our approach is done with 5 nodes, which is a significant improvement.

Problem Instance	Number of Faulty Cells	Array Dimension	Number of Spare Rows	Number of Spare Columns	Cost of Spare Row	Cost of Spare Column
1	100	100 × 100	50	50	3	3
2	110	100 × 100	55	55	3	3
3	600	4000 × 4000	300	300	3	3
4	600	4000 × 3000	400	200	4	2
5	800	1024 × 4096	200	600	20	10

Problem Instance	Optimal Cost		1 Processor	2 Processors	4 Processors	8 Processors
1	168	Time	7 min 54 sec	4 min 10 sec	2 min 16 sec	1 min 33 sec
		Speedup	1	1.90	3.49	5.10
		Expansion	456,258	456,964	454,548	451,997
2	177	Time	54 min 54 sec	27 min 24 sec	14 min 33 sec	8 min 14 sec
		Speedup	1	2.00	3.77	6.67
		Expansion	2,831,874	2,850,157	2,887,076	2,959,850
3	1590	Time	23 min 9 sec	13 min 4 sec	7 min 16 sec	4 min 41 sec
		Speedup	1	1.77	3.19	4.94
		Expansion	14,283	14,624	15,014	16,040
4	1628	Time	2 h 3 min	1 h 2 min	30 min 56 sec	16 min 21 sec
		Speedup	1	1.98	3.98	7.54
		Expansion	76,478	77,843	76,321	76,393
5	7260	Time	2 min 28 sec	1 min 54 sec	1 min 50 sec	1 min 38 sec
		Speedup	1	1.30	1.35	1.51
		Expansion	1,540	2,290	2,549	4,284

4 Results

The algorithm was implemented as parallel IDA* search [5] on a network of workstations using TreadMarks™, a tool that provides shared memory abstraction and synchronization mechanisms for distributed parallel computing. In our case, search proceeds in parallel along different paths within the same search contour. All processors are synchronized between contours to determine the next cost limit and ensure optimality. Work balance was achieved by a shared queue of nodes, whereby processors deposit nodes (if the queue is empty) or withdraw nodes (if they run out of work). Each processor performs IDA* locally, periodically checking the shared queue and also whether a solution has been found.

The performance of the scheme was tested on several instances of the OSA problem. A complication is due to the large number of independent variables in the problem, leading to a plethora of different choices. Basically, the variable that dominates the complexity is the number of faulty cells (the total number of search nodes is exponential to this number), as well as their distribution in the array.

The tables above summarize the results. The top table shows the sample instances tested and the bottom table the different execution times, speedup and expanded nodes for 1, 2, 4 and 8 processors (workstations). All experiments were conducted on eight SUN SPARCstations 4 connected through a 10 Mbit shared ethernet under a typical machine and network load for more realistic results[1]. The distribution of faulty cells was taken random (lying somewhere between totally clustered and totally independent cells) which seems to be the most difficult case.

In the first four instances, more processors imply shorter execution time, but the amount of speedup differs. Notice that best speedup was gained for the hardest instances (2 and 4). The first two instances demonstrate how the number of faulty cells affects the execution time (actually, the size of the search tree). The next two demonstrate how the dimension of the array affects the execution time. By "shrinking" the array, clustering phenomena, that can "fool" easily the heuristic function, are more possible. The last instance is an "easy" case. The heuristic function returned an estimate very close to the actual cost and the solution was found early (in the 7^{th} search contour), where parallel search had not been stabilized yet.

[1] During the experiments the workstations were being used also by other users running a variety of applications (word processors, web browsers, etc.).

5 Conclusion

From the experience gained, we conclude that many problems proven to be hard, can be confronted (at least for instances of practical importance) by algorithms that utilize appropriate domain heuristics. Moreover, by employing parallelism significant speedup can be gained, and thus solutions to harder instances. The difference between a few minutes and some hours is analogous to the difference between solving or not the problem in practice.

I would like to thank Prof. N.F. Tzeng, Prof. A. Maida and A. Kongmunvattana for their help, and the Lilian-Boudouri Foundation in Greece for financial support.

References

[1] Amza, C., Cox, A., Dwarkadas, S., Keleher, P., Lu, H., Rajamony, R. and Zwaenepoel, W. "TreadMarks™: Shared Memory Computing on Networks of Workstations," *IEEE Computer*, **29**, 2, February 1996, pp. 18–28.

[2] Day, J. "A Fault-Driven Comprehensive Redundancy Algorithm," *IEEE Design & Test*, **2**, 3, June 1985, pp. 35–44.

[3] Korf, R. "Depth-first Iterative Deepening: An Optimal Admissible Tree Search," *Artificial Intelligence*, **27**, 1985, pp. 97–109.

[4] Kuo, S. and Fuchs K. "Efficient Spare Allocation for Reconfigurable Arrays," *IEEE Design & Test*, **4**, February 1987, pp. 24–31.

[5] Lagoudakis, M., in preparation, draft available at `http://www.cs.duke.edu/~mgl/parallel.ps.gz`.

[6] Russell, S. and Norvig, P. *Artificial Intelligence: A Modern Approach*, Prentice Hall, 1995, ch. 4.