Implementation and Analysis of Iterative MapReduce Based Heuristic Algorithm for Solving N-Puzzle

Rohit P. Kondekar
Visvesvaraya National Institute of Technology, Nagpur, India
Email: rohitkondekar@gmail.com

Mohit Modi, Akash Gupta, Parag S. Deshpande, Gulshan Saluja, Richa Maru and Ankit Rokde
Visvesvaraya National Institute of Technology, Nagpur, India
Email: mohitmodi.cse12@gmail.com, psdeshpande@cse.vnit.ac.in

Abstract — **MapReduce Programming paradigm provides an elegant and efficacious platform for catering large scale parallel implementations of Heuristic Search Algorithms. We present here an implementation and analysis of Parallel Breadth First Heuristic Search (PBFHS) Algorithm for solving very large combinatorial problems. Using N-Puzzle as our application domain we found that the scalability of Breadth First Search (BFS) and Iterative Deepening A* (IDA*) is limited on a single machine due to hardware constraints. In this algorithm, we generate a remarkably restrictive, yet a large search space using combination of highly efficient admissible and non-admissible heuristics. The graphs compiled from resulting output advocates our design and implementation flow. A 7 node Hadoop cluster setup on Amazon EC2, solves the hardest 24 Puzzle in 3 hours, and 35 Puzzle in 13 hours of computing time.**

Index Terms — **hadoop, heuristic, n-puzzle, parallel breadth first heuristic search, mapreduce**

I. Introduction

N-Puzzle is a classical problem for modeling algorithms involving heuristics. It is a sliding puzzle (Fig. 1) that consists of a frame of numbered square tiles in random order with one tile missing. They are normally solved using tree traversal techniques like breadth first search, A*, IDA* [1][3] etc. But it becomes more and more difficult to keep track of puzzle states as the search space increases exponentially with the height of the tree. Furthermore in case of N-Puzzle, the search space is extremely large, which makes it difficult to be processed and stored on a single machine.

MapReduce [2] is a parallel programming paradigm for processing big data sets over clusters of compute nodes. The processing is divided into two phases namely Map & Reduce. Map phase processes key/value pairs to generate a set of intermediate key/value pairs, and reduce phase merges the key/value pairs with the same key. In nutshell: *Map(k1,v1) → list(k2,v2) : Reduce(k2, list (v2)) → list(v3).*

The inbuilt shuffle, sort and merge property of Hadoop can be exploited to process large and repeated data sets of N-Puzzle. The sequence of moves in N-Puzzle may lead to visited configurations (states), which on a simple platform necessitates recording of visited puzzle states, whereas in case of MapReduce they can be simply emitted as key/value pairs with a type flag marked. The type flag is either a parent which shows that, the state has already been expanded or else, a child. The hadoop's architecture with our configuration settings, takes care of domain independent tasks such as data partitioning, shuffling, sorting, task scheduling, data merging, node communication, synchronization, and automatic restart of failed tasks. Our scheme provides an approach for effective utilization of resources within a cluster (such as processor, main memory and disks) for solving large combinatorial problems. The bottleneck which comes into play is the performance of I/O system and data transfer rate over the network.

MapReduce processing is not efficient for small data sets because in such cases, job initialization time starts dominating the processing time. Our input being just a single N-Puzzle's start configuration, we devised our algorithm in three phases of MapReduce processing. The first phase acts as an initialization phase and the other two phases are chained together and iterated upon until

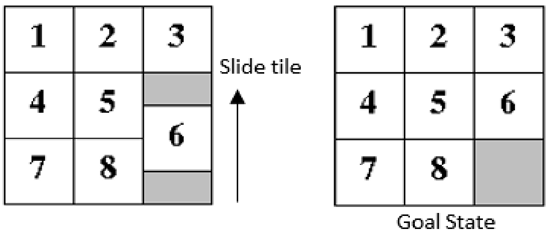

Figure 1. Eight Sliding Puzzle

the goal state is reached. In the first MapReduce phase, the mapper generates a sufficiently large amount of successor puzzle states (child configurations) using stack based depth first traversal, which are then emitted as key-value pairs, where key being the puzzle configuration and value consisting of parent/child flag, heuristics and depth. The partitioner is designed such as to feed similar configurations to same reducer. Abiding by the delayed duplication technique, the reducers combine these key/value pairs and eliminate the duplicate puzzle states, thereby outputting a refined data set for subsequent MapReduce stages.

In the second MapReduce phase, the mapper traverses three levels of successive puzzle states and emits them as key-value pairs, from which the duplicates are eliminated by reducers, as mentioned above.

The third phase acts as a special elimination phase which is executed after every few iterations, whenever the number of child states exceeds the pre-decided threshold. In this phase, the mapper randomly emits the key-value pairs, where key is the combination of reducer number & heuristics and value consisting of the puzzle configuration & depth. The reducer in this phase receives random puzzle configurations as opposed to earlier phases. In addition, it receives sorted key-value pairs based on their heuristic values, which aids in limiting the amount of generated successor puzzle states for further processing, thus keeping a tight upper bound.

II. Process Overview

We have considered 35 puzzle as a sample to analyze & demonstrate the aforementioned algorithmic process and its implementation layout as depicted in Fig. 2 is discussed here.

A. Input Configuration

The job input consists of an N-Puzzle start configuration expressed as comma separated string with its type flag set as child.

B. Initialization MapReduce Phase (1st Phase)

This phase as shown in Fig. 2 is executed once in application lifecycle to produce sufficient amount of puzzle states to be processed on distributed MapReduce framework. The mapper implements a stack based depth first traversal, generating 12 levels of (exponentially increasing) successive puzzle states, which are then emitted as keys and the corresponding type flag, heuristics & depth as values.

The partitioner groups the similar keys (states), feeding them into same reduce partition. The reducer is executed for each partition thereby eliminating duplicate states, keeping the lowest depth number intact. These distinct puzzle states are then emitted by reducer to next MapReduce phase.

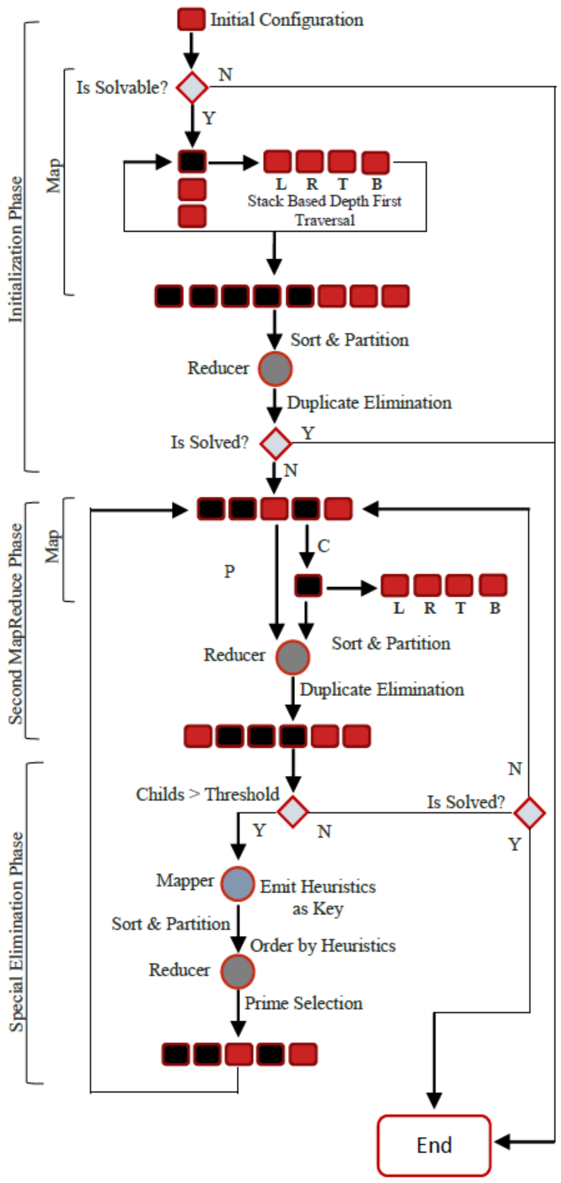

Figure 2. Process Overview

C. Second MapReduce Phase (2nd Phase)

The Second MapReduce phase as shown in Fig. 2 marks the beginning of the iterative chained jobs. The mapper inspects the type attribute of incoming key-value pairs, the puzzle states marked as parent are not processed but simply emitted to reducers thereby providing a mechanism to maintain a list of visited states. The states marked as child are converted to parents and used for subsequent, three levels of child generation (left/right/bottom/top if any) using stack based iteration method. Each map phase traverses three levels to generate sufficient states to process in further MapReduce phases. Heuristics are evaluated for new child states, and are emitted with their appropriate depths. The parents generated in the above process are also emitted along with their heuristics and depth.

The partitioner and reducer remains same as used in the Initialization MapReduce phase.

D. Elimination MapReduce Phase (3rd Phase)

The special elimination phase as depicted in Fig. 2, is executed after every few iterations, whenever the number of child states exceeds the pre-decided threshold. The mapper randomly emits the puzzle states to different reducers with a custom key comprising of reducer number and the four calculated heuristics and value being full configuration and type details.

The Partitioner partitions the keys depending upon the reducer number allotted. The customized Grouping Comparator sorts the keys based on the heuristics. The ordering can be inferred from Fig. 3, which shows that the percentage change (1) between the average magnitudes of each heuristics, grouped five subsequent levels (depths) at a time.

$$\%\ Change = \frac{|2nd\ Group - 1st\ Group|}{1st\ Group} \times 100 \quad (1)$$

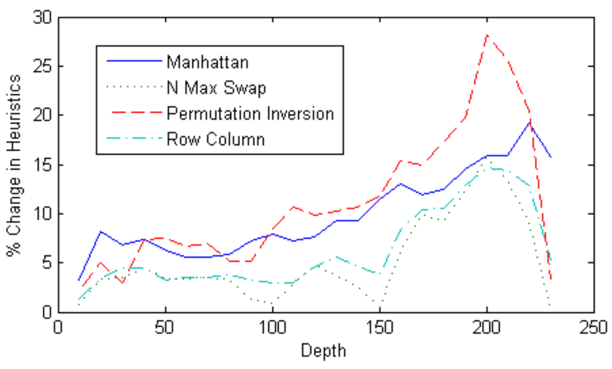

Figure 3. Percentage change in Heuristics for 35 Puzzle

More the percentage change in the average value, better is the performance of the heuristic. The heuristics are ordered by Manhattan Distance with Linear Conflicts, Permutation Inversion, Tiles out of row & column and N-Max Swaps respectively. Manhattan distance is given more priority, as Permutation inversion is a non-admissible heuristic i.e. it is not optimal and may result in suboptimal solution, but may do so in much shorter time, hence given more priority than other heuristics.

Each reducer, restricts the population to prime few thousands based on the sorted order, thereby putting a tight upper bound on maximum number of child states to be processed in each iteration. The Fig. 4 shows the number of states generated in each iteration of MapReduce phases for 35 Puzzle and the Fig. 5 shows the total number of parents getting accumulated as iteration proceeds.

III. Solvability Test

The solvability of N-Puzzle can be checked using configuration factors such as number of inversions, grid width and blank position [9][10]. An inversion is when a tile precedes another tile with a lower number on it. The solution state has zero inversions.

The following conditions are checked for solvability:

a. If the grid width is odd, then the number of inversions should be even.

b. If the grid width is even, and the blank is on an even row counting from the bottom (second-last, fourth-last etc.), then the number of inversions should be odd.

c. If the grid width is even, and the blank is on an odd row counting from the bottom (last, third-last, fifth-last etc.) then the number of inversions should be even.

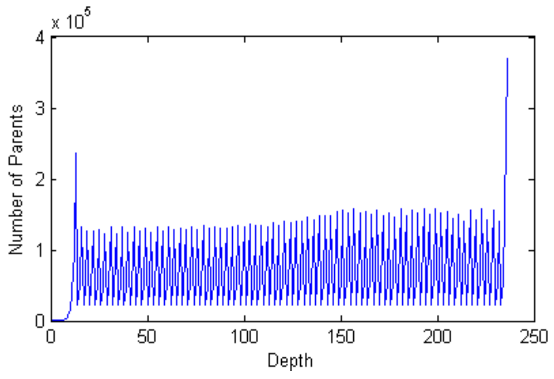

Figure 4. Number of Parents per Depth for 35 Puzzle

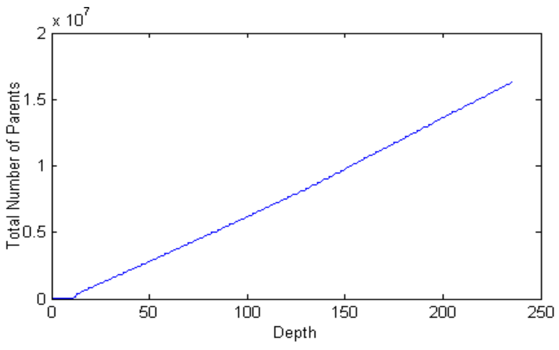

Figure 5. Total number of parents for 35 Puzzle

7	3	1
5	x	6
8	2	4

Figure 6. Sample 8 Puzzle Configuration

IV. Heuristics

Heuristic is a function, h(n) defined on puzzle state of search tree, which serves as an estimate of how close the current and final goal configurations' are. It ranks alternative configurations' and helps in making appropriate decision of which branch to follow during search process. The heuristics applied are described in this section.

A. Manhattan Distance Heuristic with Linear Conflicts (MDLC)

The Manhattan Distance is the distance between two points measured along axes at right angles defined as in (2).

$$H(n) = \sum_{i=1}^{n}(| X_i(s) - X_{fi} | - | Y_i(s) - Y_{fi} |) \quad (2)$$

Where X_i and Y_i are coordinates of tile 'i' in state s and X_{fi} and Y_{fi} are coordinates of tile 'i' in the goal state and n represents the total number of non-blank tiles. The limitation of the Manhattan Distance heuristic is that it considers each tile independently, while in fact tiles interfere with each other.

This interference can be reduced by using an enhancement called Linear Conflict. Two tiles t1 and t1 are in a linear conflict if t1 and t2 are in the same line, the goal positions of t1 and t2 are both in that line, t1 is to the right of t2 and goal position of t1 is to the left of the goal position of t2. The linear conflict adds at least two moves to the Manhattan Distance of the two conflicting tiles, by forcing them to surround one another. Both Manhattan Distance as well as Linear Conflict are admissible heuristics. Fig. 7 relates the maximum, average and minimum MDLC values of a 35 puzzle problem.

The value of Manhattan distance with linear conflicts for sample 8 puzzle configuration (Fig. 6) is:

Manhattan Distance = 2+1+2+1+0+1+2+3 = 12
Linear Conflicts = (3, 1) = 2
Total Distance = 14

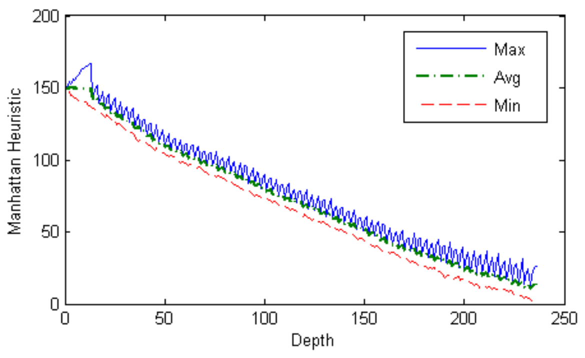

Figure 7. Manhattan Distance Heuristic with Linear Conflicts

B. Permutation Inversions Heuristics

A pair of tiles (t_i, t_j) is called inversion in permutation t if $i > j$ and $t_i < t_j$ [6][7][8].

$$N = \sum_{i=1}^{n} n_i \qquad (3)$$

(3) where n_i denotes the permutations of order n for ith tile and N the total number of permutations. It is a non-admissible heuristic. Fig. 8 relates the maximum, average and minimum permutation inversion heuristic values of a 35 puzzle problem. The value of permutation inversion heuristic for sample 8 puzzle configuration shown in Fig. 6 is:

N = 6 + 2 + 0 + 2 + 2 + 2 + 0 + 0 = 14

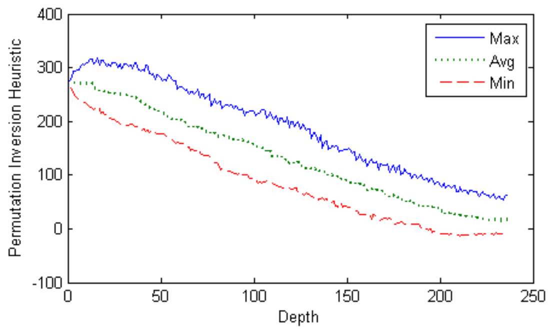

Figure 8. Permutation Inversion Heuristic

C. Tiles out of Row and Column Heuristics

It is summation of the total number of tiles which are not in their correct row and the tiles which are not in their correct column. It is an admissible heuristic.

H(n) = Number of tiles out of row + number of tiles out of column

Fig. 9 relates the maximum, average and minimum heuristic values of a 35 puzzle problem.
The value of Tiles out of row and column heuristic for sample 8 puzzle configuration (Fig. 6) is:

H(n) = 3 + 5 = 8

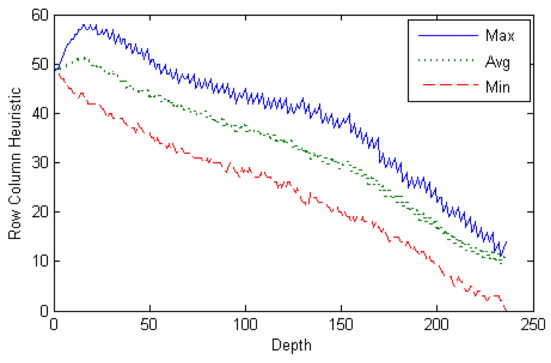

Figure 9. Tiles out of Row and Column Heuristic

D. N Max Swaps Heuristics

It refers to the number of steps it would take to solve the problem, if any tile (not just adjacent tile) can be swapped with the blank tile. It is an admissible heuristic. The heuristic function is implemented using two arrays P[ ] represents current permutation, and B[ ] represents the location of element i in the permutation array. Then P[B[n]] and P[B[B[n]]] are iteratively swapped until the goal state is reached.

Figure 9 relates the maximum, average and minimum N Max Swaps heuristic values of a 35 puzzle problem. The value of N Max Swap heuristic for sample 8 puzzle configuration (Fig. 6) is:

H(n) = 8

V. Test Environment

The test environment used:-

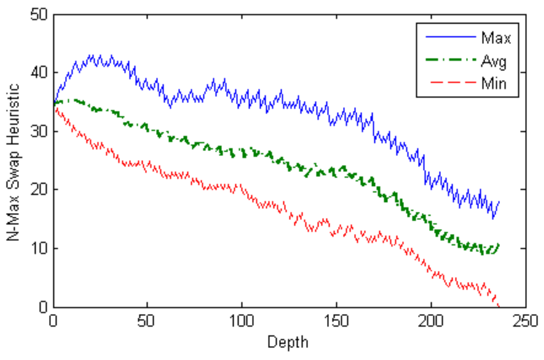

Figure 10. N Max Swap Heuristic

- Platform: Amazon Elastic Compute Cloud (EC2).
- Instance Type: Small Instances.
- Number of Instances: 7
- Operating System: CentOS 5.8
- Memory: 1.7 GB
- Programming Language: Java
- Framework: Hadoop

VI. Conclusion

We provide here an efficient implementation of Parallel Breadth First Heuristic Search (PBFHS) fused with stack based depth first traversal using MapReduce Programming model. The algorithm is proficient, fault-tolerant and easy to implement, because the Hadoop's MapReduce framework takes care of all domain independent tasks. With its efficient utilization of distributed resources (CPU, main memory and disks), MapReduce provides us a platform for solving large search space problems by efficaciously distributing the workload across the cluster.

This algorithm solves large N Puzzle problems which cannot be solved on a single computer due to severe memory constraints, limited processing power and slower I/O capability. The graphs obtained from the computation data provide means to compare different heuristics and to apply them in an ordered fashion, thereby drastically reducing the search space. The distributed approach solves the hardest 24 puzzle taking 32 iterations (109 level generation) in 3 hours and 35 puzzle taking 75 iterations (235 level generation) in 13 hours of computation time.

Acknowledgment

We would like to thank Persistent Systems Ltd. and Visvesvaraya National Institute of Technology for providing us computing resources and facilities required for implementing this algorithm.

References

[1] A. Reinefeld, T. Schütt, "Out-of-Core Parallel Heuristic Search with MapReduce", *High-Performance Computing Symposium* HPCS 2009, Kingston, Ontario.

[2] J. Dean, S. Ghemawat, "MapReduce: simplified data processing on large clusters", *Magazine Communications of the ACM*, vol. 51, 2008, pp. 107-113.

[3] Korf, R.E.: Depth-first iterative-deepening: *An optimal admissible tree search*, Artificial Intelligence, 97-109, 1985.

[4] Zhou, R., Hansen, E.A.: Parallel breadth-first heuristic search on shared-memory architecture. In: Workshop on heuristic search, memory-based heuristics and their appl. (2006)

[5] Knuth, D. E. *The Art of Computer Programming*, Vol. 3: Sorting and Searching, 2nd ed. Reading, MA: Addison-Wesley, 1998.

[6] Skiena, S. "Inversions and Inversion Vectors." §1.3 in *Implementing Discrete Mathematics: Combinatorics and Graph Theory with Mathematica. Reading*, MA: Addison-Wesley, pp. 27-31, 1990.

[7] Pemmaraju, S. and Skiena, S. Computational Discrete Mathematics: *Combinatorics and Graph Theory in Mathematica*. Cambridge, England: Cambridge University Press, 2003.

[8] Johnson, W. W. "Notes on the '15 Puzzle. I.' "Amer. J. Math. 2, 397-399, 1879.

[9] Story, W. E. "Notes on the '15 Puzzle. II.' "Amer. J. Math. 2, 399-404, 1879.

[10] Zhou, R., Hansen, E.A.: Parallel breadth-first heuristic search on shared-memory architecture. In: *Workshop on heuristic search, memory-based heuristics and their appl.* 2006.

Rohit P. Kondekar was born in Nagpur, India in 1990. He received the B.Tech degree in Computer Science and Engineering from National Institute of Technology (VNIT), Nagpur, India in 2012.

Since July 2012, he has been working as Member of Technical Staff (MTS) at Oracle India, Bangalore. He is passionate about Big Data innovations and has three internationally acclaimed IEEE conference publications on solving large scale Genetic and Image Processing Algorithms using Hadoop. He also published a paper in association with Indian Institute of Technology CSE Labs for devising a Bluetooth based system for simulating optical mark recognition sheets on mobile phones.

Mohit Modi was born in Alwar, India, in 1991. He received the B.Tech degree in Computer Science and Engineering from National Institute of Technology (VNIT), Nagpur, India in 2012.

He has worked as Summer Trainee at Birlasoft. Since June 2012, he has been working with OEM Patching team on Exadata Systems at Oracle India, Bangalore. He is passionate about latest research in Information Retrieval and Data Science.

Akash Gupta was born in Patna, India in 1990. He received the B.Tech degree in Computer Science and Engineering from National Institute of Technology (VNIT), Nagpur, India in 2012.

Since June 2012, he has been working as Rotational Software Engineer at Microsoft India, Hyderabad in the Master Data Management Team. He is passionate about latest innovation in the field of Big Data & machine learning