Exploring the Limits of Computing Through Exhaustive Search

Jeffrey L. Lehman
Huntington University

Abstract

Many computing problems can be solved by identifying all possible moves or combinations of events and then picking the best solution. Problems in this domain provide fertile ground for exploring problem representation, storage requirements, and computational complexity. The problems and solution approaches are easy to understand, yet quickly push the memory and storage limits of a personal computer. This paper describes insights from a preliminary investigating of two exhaustive search problems, the 15-puzzle and Rubik's cube. The insights gained by looking at exhaustive search problems can be integrated into classroom discussions and projects.

1. Introduction

There are many problems that can be solved by identifying all possible moves or combinations of events and then picking the best solution. This exhaustive search approach has been called "God's Algorithm", meaning God can see all possible solutions, thus can always give the optimum result. A trivial example is the game of game of Tic-Tac-Toe which can be solved almost instantly given the limited number of combinations. A non trivial example is the game of checkers with 5×10^{20} possible positions which was solved in 2007 (Schaeffer, et al., 2007). The traveling salesman problem, determining the optimum route for visiting cites, can be solved if the number of destinations is limited, but is considered "impossible" for larger numbers of cities given the required computational time.

Many problems can be simplified by identifying symmetries or sub-problems. These symmetries reduce can greatly reduce the search space and make the problem feasible to solve. In some cases, however, it may not be desirable or possible exploit these symmetries. As processing power and memory increase the number of problems that can be addressed by exhaustive search should increase (Nievergelt, 2000). Problems in this domain provide a fertile ground for exploring problem representation, storage requirements, and computational complexity.

This paper describes insights from a preliminary investigating applying exhaustive search to the 15-puzzle and Rubik's cube. Section 2 begins with an overview of the 15-Puzzle and Rubik's cube. Section 3 describes solution and problem representations. Section 4 describes two exhaustive search approaches using database and memory structures. Section 5 describes insights and conclusions.

2. The Problems

2.1 The 15-puzzle

The 15-puzzle consists of a 4 by 4 grid of tiles placed in frame. There are fifteen tiles labeled "1" to "15" with one open space. The tiles surrounding the empty space may move "up", "down", "left" or "right" into the open position. The puzzle is solved when the tiles are arranged in ascending order with the space at the end. While puzzle promoter Sam Loyd is often given credit for creating the puzzle, it was most likely created in the early 1870's by Noyes Palmer Chapman, a postmaster from Canastota, New York. The puzzle was a national craze throughout the 1880's appearing in numerous news paper advertisements, editorials, and poems (Slocum, 2006).

2.2 Rubik's Cube

The Rubik's' cube consists of six sides with nine tiles on each side. Each side has a unique color. Using the quarter turn model there are twelve possible moves. Each of the six sides of the cube may move a quarter of a turn left or right. Using the face turn model there are 18 moves. Each face may move left and right 90 or 180 degrees. The puzzle is solved when the tiles are positioned with a unique color on each side. The Rubik's' cube was invented in the 1970's by Hungarian Erno Rubik, a sculptor, architect, and teacher of three dimensional design courses (Singmaster, 1982). Like the 15-puzzle the Rubik's cube also achieved national craze status in the 1980's.

3. Solution and Puzzle Representation

3.1 Solution Graph

Both the 15-puzzle and Rubik's cube can be solved, at least in theory, by using exhaustive search to create a solution graph of all possible positions. The graph is created by starting with a solved puzzle state, applying each of the valid moves, and storing each successive move until all combinations are stored. The move that generates the new state is stored providing a link between each state. The level number describing the distance from the origin is also stored. An optimum solution can be found by finding the puzzle in the graph and tracing its path back to the solved state.

The 15-puzzle has $16!/2 = 10{,}461{,}394{,}944{,}000$ combinations. Research has shown the upper bounds for the hardest puzzle to be 88 moves. In other words, the minimal distance to the origin or solved state is at most 88 moves, thus most "scrambled" puzzle can be solved with 88 moves (Culberson & Schaeffer, 1996). The Rubik's cube has 4.33×10^{19} possible combinations. Researchers believe that the minimal distance to the origin is somewhere in the low 20's. Current research has shown this number to be at most twenty-five (Rokicki, 2008). Given the large number of combination states that must be stored can a solution graph for the 15-puzzle

and Rubik's cube be created using current processing power and stored using current storage capacities? We next looked at the storage requirements for each puzzle.

3.2 Character Array Representation

The 15-puzzle contains tiles labeled "1" to "15". An array of 16 characters can be used to store this data. Each tile is represented by a single character with a hex representation used to map tiles "10" to "15" to characters "A" to "F". Character "0" is used to represent the open tile. The first character represents the top left tile. A solved puzzle with the open tile in the lower right would be represented by the array "123456789ABCDEF0" using 16 bytes.

A Rubik's cube has six sides with nine tiles on each side. An array of 54 characters can also be used to store this data. Each tile is represented with a single character corresponding to the color e.g. 'R' red, 'B' blue, 'G' green, 'O' orange, 'Y' yellow, and 'W' white'. Each tile is numbered and grouped by its corresponding side. The characters 1 to 9 represent the tiles for the first side, characters 10 to 18 represent the tiles for the second side, etc ... The array "RRRRRRRRR GGGGGGGGG BBBBBBBBB OOOOOOOOO YYYYYYYYY WWWWWWWWW" would represent a solved puzzle using 54 bytes.

3.3 Bit Array Representation

A bit array can reduce the storage requirements for the 15-puzzle. A four bit binary number can be used to represent each tile rather than a 1 byte character. A solved puzzle with the open tile in the lower right would be represented by the bit array "0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111 0000" using 8 bytes.

A bit array can also be used for the Rubik's cube. Three bits can be used to store the color value for each tile e.g. '000' red, '001' blue, '010' green, '011' orange, '100' yellow, and '101' white'. Using this representation each state can be stored using 20.25 bytes (3 bits * 54 tiles = 162 bits = 20.25 bytes).

3.4 Alternate representations

There are many variations for storing each puzzle. For the 15-puzzle the position of each tile could be stored rather than the value of each tile. The position of the blank tile would not need to be stored as it could be easily calculated as the only position not included. This would reduce the storage to 15 bytes using array of characters and 7.5 bytes using the array of bits. While the storage requirements are reduced additional processing time would be needed to determine the location of the open tile and to handle partial-byte values.

3.5 Storage Implications

The choice of how to represent each state has both memory and long term storage implications. The storage structure limits how many states can be stored. The type of storage

structure will affect may affect processing time. Additional processing is needed to extract data when unique bit representations are used rather than standard byte length structures.
Most personal computers today (May 2009) have 2 to 4 gigabytes (GB) of memory and at most 1 terabyte (TB) of long term storage available. As can be seen in the table 1, storing all states of either the 15-puzzle or Rubik's cube is not feasible given the proposed problem representations. Test algorithms will need to focus on a subset of the problem such as generating all moves for a given level or distance from the origin.

Another option is to use a smaller version of the problem such as the 8-puzzle. The 8-puzzle is a smaller 3 by 3 grid version of the 15-puzzle. It has 9!/2 = 181,440 combinations. Given its smaller size a graph of the 8-puzzle can easily be generated. The hardest 8-puzzle requires at most 31 moves (Reinefeld, 1993).

Puzzle	Total States	Bytes per State	Total Storage
8-Puzzle	181,440	9	2 MB
15-puzzle	10,461,394,944,000	16	152 TB
15-puzzle	10,461,394,944,000	8	76 TB
15-puzzle	10,461,394,944,000	7.5	71 TB
Rubik's cube	43,300,000,000,000,000,000	54	2,126,580,512 TB
Rubik's cube	43,300,000,000,000,000,000	20.25	797,467,692 TB

Table 1. Storage Requirements

4. Algorithms

The Java programming language was used to implement each of the problems. A class structure was developed to store the state of a puzzle and perform the valid moves. Three "proof of concept" algorithms were created. Two used a database and one used memory structures to store the solution states. Each puzzle used the character array representation (Section 3.2). Each sample algorithm was run on a personnel computer with either an Intel Core 2 1.86 GHz or 2.0 GHz processor with 2 GB memory.

4.1 First Database Approach

The first database approach used a MySQL database to store the puzzle states (Figure 1). The solved puzzle state was first added to the database (insert). The initial state is marked as "unvisited". Within the loop the next unvisited state S is retrieved from the database (query). If state S exists, all new moves from this state are added to the database (insert). The new moves are marked as "unvisited" In should be noted that duplicate states are automatically discarded by the database due to the use of primary keys. State S is marked as "visited" (update). This process is repeated until all states are visited.

```
Stop = False
Create solved state I
Add I to database (insert)
Do
    Get next unvisited state S and store in result set (query)
    If S exists
        Insert next moves from S into database (insert)
        Mark S as visited (update)
    Else
        Stop = True
    End If
Loop Until Stop
```

Figure 1. First Database Approach

The first database approach required approximately 4.5 hours of computational time to generate all 181,440 states for the 8-Puzzle (see Appendix Table 2). Reinefeld had reported that it took less than 1 hour on a Sun workstation in 1993 (Reinefeld, 1993). Obviously improvements were needed.

4.2 Second Database Approach

The second database approach (Figure 2) modified the algorithm to retrieve all unvisited states from a given level rather than the single next unvisited state (query). This reduced the need to retrieve individually retrieve each unvisited state. Within the loop all next moves from each state in level X are inserted into the database (insert). This eliminated the need to mark each state as "visited" thus eliminating all update queries with the database. This approach allows subsequent levels to be generated from X to N.

```
Create solved state I
Add solved state I to database (insert)
X = 1
While X less than N
    Get all states S from level X and store in result set RS (query)
    For each state S in RS
        Insert next moves into database (insert)
        Increment X
    End For
End Loop
```

Figure 2. Second Database Approach

The second database approach required approximately 30 minutes of computational time to generate all 181,440 states for the 8-Puzzle.

Further refinement of the algorithm was made to bulk insert new moves rather than individually inserting each new state. New states were added to a list in memory and inserted at the end of each level or once 1,000,000 states were identified. Using multi-valued inserts reduced the time to generate all 181,440 states for the 8-Puzzle to approximately 12 seconds.

The refined database approach was used to test the 15-puzzle. The first 22 levels for a total of 12,318,701 states were generated in approximately 20 minutes before the program terminated due to memory issues (see Appendix Table 3). The refined database approach was used to test the Rubik's Cube. The first 6 levels containing 983,926 moves were generated in approximately 8 hours.

4.3 Memory Approach

The third approach used memory structures to store the puzzle states (Figure 3). A Java Collection Framework (JCF) list was used to store all "unvisited" states and a hash map was used to store the visited states. The solved puzzle state I was first added to the hash map and list. While the list was not empty, the next state S was retrieved and removed from the list. Each subsequent move from S was added to the hash map and list if it did already exist. This process is repeated until all states are visited.

```
Create solved state I
Add I to hash map
Add I to list
While list not empty
    Remove next state S from list
    For each next move from S
        If next move does exist in hash map
            Insert next moves from S into hash map
            Add next moves from S to list
        End If
    End For
End Loop
```

Figure 3. Memory Approach

The memory approach was used to test the 8-puzzle. It took approximately 8 seconds of computational time to generate all 181,440 states for the 8-Puzzle. The memory approach was used to test the 15-puzzle. The first 21 levels for a total of 6,516,290 states were generated in approximately 3 hours before the program terminated due to memory issues. The memory

approach was used to test the Rubik's cube. The first 6 levels for a total of 983,926 states were generated in approximately 2 minutes before termination due to memory issues.

5. Insights and Conclusions

This paper described a preliminary investigating of using exhaustive search algorithms to address the 15-puzzle and Rubik's cube. Representations for solution graphs were developed and implemented using the Java programming language. Test algorithms using a MySQL database and memory structures were implemented. While the test algorithms were of limited success, the insights gained by looking at exhaustive search problems can be integrated into classroom discussions and projects.

A key decision in any problem implementation is how to represent the problem. In many of the programs we develop for introductory programming courses the variable types we select have little consequence. The use of 4-byte numeric variable has no impact when memory is not an issue. These decisions, however, greatly affect the storage requirements for a program and can dictate success or failure of the algorithm. Further research is needed to determine the optimal representation for the 15 puzzle and Rubik's cube.

The choice of data structures also has an impact on processing time. In the sample implementations I quickly observed that as expected using a hash map to store values reduced the processing time rather than using a sequential list. In my investigation I found that I needed a hash map that could guarantee sequential access to all elements. It illustrates the need to have a lower-level understand of the implementation of the programming libraries we use. We talk about these structures in data structures courses, but I am not sure that it is really clear until you have a real problem to apply them to.

There are always ways to refine and improve algorithms. This was illustrated by the test algorithms developed for this research. The database approach required 4.5 hours of processing time. By modifying this algorithm to use multi-valued database inserts the time was reduced to 30 minutes. By eliminating the database and using a memory approach the time was reduced to 12 seconds. Further refinements are needed.

A key benefit of my research was that it required me to gain a deeper understanding of computer software and hardware. The preliminary investigation required a better understanding of the Java memory model. Additional command line argument is needed to make use of the machine memory. The database approach required looking at the efficiencies of queries. For example a single insert query with multiple values is more efficient than individual inserts. Limits were also determined such as how much data can be sent via a single MySQL insert statements. It also required looking at the amount of data that can be stored in a single table.

Both the Rubik's cube and 15-puzzle are good research problems as they are fairly easy to understand, yet quickly push the memory and storage limits of a personal computer. Both of the problems are scalable. If the 3 x 3 cube is ever "solved", the 4 x 4 cube is waiting. The same is true for the 15-puzzle with the 5 x 5 24-puzzle. They provide fertile ground for exploring problem representation, storage requirements, computational complexity, and algorithms. While using exhaustive search to solve the 15-puzzle and Rubik's cube will require further research to address processing and storage limits it can serve as a benchmark for exploring the limits of computing.

Acknowledgements

The author wishes to acknowledge the Ferne and Audry Hammel Research Endowment of Huntington University for a 2008 mini-grant supporting this research.

References

Culberson, J. C., & Schaeffer, J. (1996). Searching With Pattern Databases. (pp. 402-416). Springer-Verlag.

Nievergelt, J. (2000). Exhaustive Search, Combinatorial Optimization and Enumeration: Exploring The Potential of Raw Computing Power. (pp. 18-35). Springer.

Reinefeld, A. (1993). Complete Solution of the Eight-Puzzle and the Benefit Of Node Ordering., (pp. 248-253).

Rokicki, T. (2008). Twenty-Five Moves Suffice for Rubik's Cube. *CoRR , abs/0803.3435.*

Schaeffer, J., Burch, N., Björnsson, Y., Kishimoto, A., Müller, M., Lake, R., et al. (2007). Checkers Is Solved. *Science , 317*, 1518-1522.

Singmaster, D. F. (1982). *Handbook of Cubik Math.* Enslow Publishers.

Slocum, J., & Sonneveld, D. (2006). *The Fifteen Puzzle.* The Slocum Puzzle Foundation.

Appendix

Level	States
0	1
1	2
2	4
3	8
4	16
5	20
6	39
7	62
8	116
9	152
10	286
11	396
12	748
13	1024
14	1893
15	2512
16	4485
17	5638
18	9529
19	10878
20	16993
21	17110
22	23952
23	20224
24	24047
25	15578
26	14560
27	6274
28	3910
29	760
30	221
31	2

Table 2. 8-Puzzle States (complete)

Level	States
0	1
1	2
2	4
3	10
4	24
5	54
6	107
7	212
8	446
9	946
10	1948
11	3938
12	7808
13	15544
14	30821
15	60842
16	119000
17	231844
18	447342
19	859744
20	1637383
21	3098270
22	5802411

Table 3. 15-Puzzle States
(complete through level 22)