Multi-Agent Path Finding Implementation: Sliding Puzzle

Kirt Narain
SUNY Polytechnic Institute
Rome, New York
naraink@sunypoly.edu

Brett Bala
SUNY Polytechnic Institute
Herkimer, New York
balab@sunypoly.edu

ABSTRACT

With Multi-Agent Path Finding techniques utilized, we created a sliding puzzle solver, aiming for as few moves as possible. The group managed to finish the creation of the puzzle solver, but did not manage to add addition challenges such as stuck pieces or multiple blank spaces. We are, however, confident in the understanding on how to implement the latter.

PVLDB Reference Format:
Kirt Narain and Brett Bala. Multi-Agent Path Finding Implementation: Sliding Puzzle. PVLDB, 14(1): , 2020.

Github Link:
The source code, data, and/or other artifacts have been made available at https://github.com/BrettBala/MAPF-Sliding-Puzzle.

1 INTRODUCTION

Multi-Agent Path Finding (abbreviated as MAPF), is a problem in which the task is to deliver any n number of agents to a specific location. The main restriction applied is that any of the n agents are able to move independently, but must avoid a number of possible collisions depending on the application of the algorithm. The main conflict that will always have to be avoided is a 'Vertex' collision, where two agents can not occupy the same space at the same time.

MAPF algorithms have several uses in the modern industry which have drawn attention to it. Anything from self driving autonomous cars, robotics, and automated warehouses to artificial intelligence path-finding in video games. Based on the application, the benchmarks for MAPF algorithms can vary greatly. Independent agents will have different scenarios that can cause conflicts, and different evaluations for recording the efficiency.

For our implementation, we have created a sliding block puzzle that generates a random square board, with each movable piece being treated as an individual agent for our implementation. Each piece has the ability to move into an empty space if there is one adjacent. As with any MAPF implementation, the pieces are not allowed to occupy the same spaces as each other in one turn.

2 EVALUATING MAPF

Evaluating a path finding algorithm with multiple agents usually occurs in one of two ways; the 'Makespan', or the 'Sum of Costs' of a board. It depends on the application which one is used. For most real world implementations, the sum of costs is used. In classical MAPF scenarios, it is assumed that every action (including moving, waiting, or any swapping) takes one time step [3]

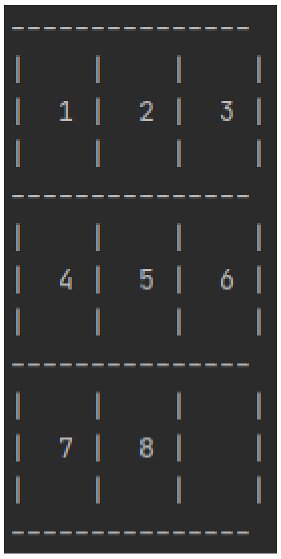
2.1 Makespan

The makespan is the number of steps required for all agents to reach their target. With certain applications, all movement happens in 'turns', where each agent makes a move if available. After any movements are made, that is considered to be one turn. The makespan is the number of turns until every agent has reached the goal. For our implementation, the efficiency of our puzzle board was calculated using both the makespan of the board and the time it took to process the correct result.

2.2 Sum of Costs

The sum of costs is the summation of each agent's makespan, also known as the 'Flowtime' of the board. This is useful for when the agents are using resources individually, such as taxiing an airport.

3 STRUCTURE OF A SLIDING PUZZLES

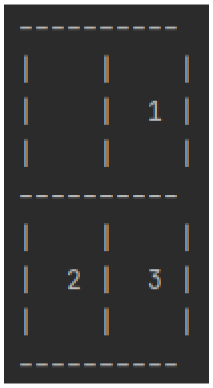


```
---------------
|    |    |    |
|  1 |  2 |  3 |
|    |    |    |
---------------
|    |    |    |
|  4 |  5 |  6 |
|    |    |    |
---------------
|    |    |    |
|  7 |  8 |    |
|    |    |    |
---------------
```

Figure 1: A completed 3 by 3 sliding puzzle

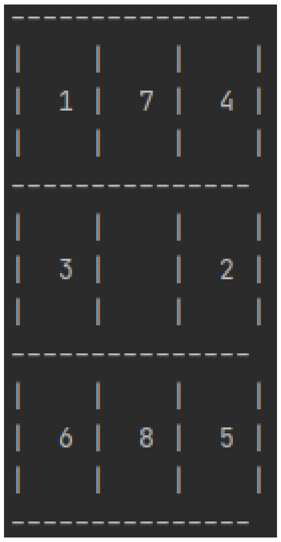
A sliding puzzle typically consists of an n by n board, with (n^2-1) numbers and a singular blank space. From which all pieces must order themselves in numeric order, with 1 being in the top left and the blank space being in the bottom right. Pieces may only move onto the blank space. See Figure 1 to see a complete board.

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.



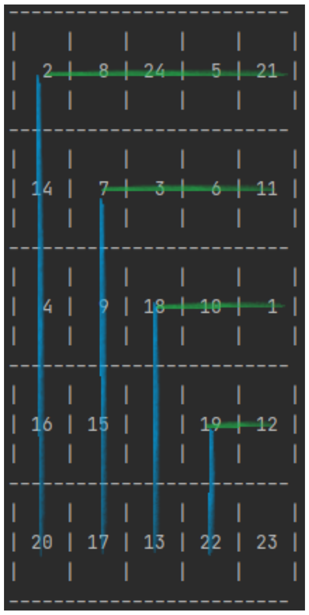
```
----------
|    |    |
|    | 1  |
|    |    |
----------
|    |    |
| 2  | 3  |
|    |    |
----------
```

Figure 2: This 2 by 2 puzzle is unsolvable



```
---------------
|    |    |    |
| 1  | 7  | 4  |
|    |    |    |
---------------
|    |    |    |
| 3  |    | 2  |
|    |    |    |
---------------
|    |    |    |
| 6  | 8  | 5  |
|    |    |    |
---------------
```

Figure 3: This board has an inversion count of 10 and has an odd *n*, therefore making it solvable



```
-------------------------
|    |    |    |    |    |
| 2  | 8  | 24 | 5  | 21 |
|    |    |    |    |    |
-------------------------
|    |    |    |    |    |
| 14 | 7  | 3  | 6  | 11 |
|    |    |    |    |    |
-------------------------
|    |    |    |    |    |
| 4  | 9  | 18 | 10 | 1  |
|    |    |    |    |    |
-------------------------
|    |    |    |    |    |
| 16 | 15 |    | 19 | 12 |
|    |    |    |    |    |
-------------------------
|    |    |    |    |    |
| 20 | 17 | 13 | 22 | 23 |
|    |    |    |    |    |
-------------------------
```

Figure 4: *Divide and Conquer*: How the board is solved and viewed by the AI

3.1 Generating a board

In order to start solving sliding puzzles, we must first realize how to generate a solvable one. Not any set of *n* by *n* boards are solvable. For example look at the 2 by 2 board, Figure 2, try as much as you like the board is unsolvable. To determine whether a board is solvable depends on three factors:

(1) *Whether the n is odd or even*
(2) *The number of inversions*
(3) *What row the blank exists*

Criteria *(1)* and *(3)* are self explanatory, but what is an inversion? An inversion is when two numbers, at some point, will have to invert their positions relative to each other. A simpler way to count the number of inversions is to follow the board in reading order (left to right and top to bottom) and anytime a value is smaller than a previous value, that is an inversion. Figure 3 shows a board and its count. Now to tell if the board is actually solvable? To be solvable, the board must adhere to these rules[1]:

- If N is odd, then puzzle instance is solvable if number of inversions is even in the input state.
- If N is even, puzzle instance is solvable if
  - the blank is on an even row counting from the bottom (second-last, fourth-last, etc.) and number of inversions is odd.
  - the blank is on an odd row counting from the bottom (last, third-last, fifth-last, etc.) and number of inversions is even.
- For all other cases, the puzzle instance is not solvable.

3.2 Seeing Future Moves

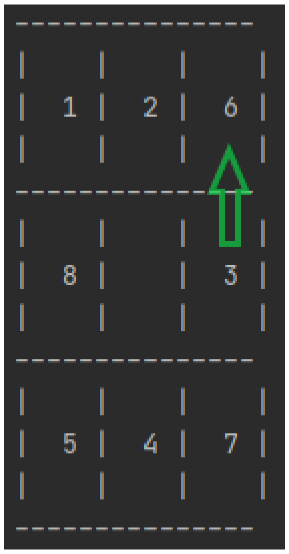
At first glance, you may think we have to map out moves in some sort of code or array, such as Piece 3 moves left, Piece 2 moves up, etc. However, we find it much easier to simply clone the existing board, and make all possible legal moves. From that, we get all the potential boards our current board can lead to. We can then do the same for all of those boards to get a depth search.

4 SOLVING A SLIDING PUZZLE

The most obvious thought when tasked with solving a sliding puzzle is using a divide and conquer approach, which is exactly what the group did. However, there are more complexities than just a simple divide and conquer strategy.

4.1 Divide and Conquer

By solving the top row and left column, it simplifies the board down to a smaller board. You may keep doing so until the board is completely solved. The group decided to solve a row, then a column, in that order until the board is solved. See Figure 4 to see in what order the board is solved by the AI. However, you cannot just place

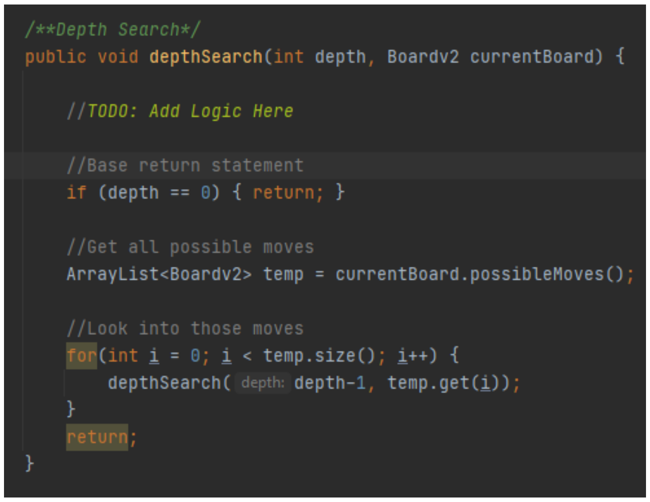


```
---------------
|    |    |    |
|  1 |  2 |  6 |
|    |    |    |
---------------
|    |    |    |
|  8 |    |  3 |
|    |    |    |
---------------
|    |    |    |
|  5 |  4 |  7 |
|    |    |    |
---------------
```

Figure 5: Notice how the board wants to get 3 into place to complete the row. However it will have to, at some point, move 2 out of place in order to get 3 into place.

the pieces in place one by one and leave them there. For example, Figure 5 showcases how the board will sometimes need to move an already correct piece in order to get another piece into place. So you must find a way to judge a board and leave some flexibility so that the board may move pieces out of place to ensure all pieces can get to their goal destination.

4.2 Recursive Depth Search

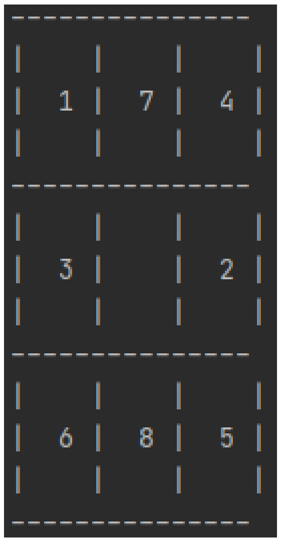


```java
/**Depth Search*/
public void depthSearch(int depth, Boardv2 currentBoard) {

    //TODO: Add Logic Here

    //Base return statement
    if (depth == 0) { return; }

    //Get all possible moves
    ArrayList<Boardv2> temp = currentBoard.possibleMoves();

    //Look into those moves
    for(int i = 0; i < temp.size(); i++) {
        depthSearch( depth: depth-1, temp.get(i));
    }
    return;
}
```

Figure 6: A simple plan on how to search the board for the best moves

In order to see our different options for moves, we must create a way to see future moves beyond just the next. A recursive depth search is a great way to accomplish this. See Figure 6 for an example on how the depth search will work. The code shows how implementing a depth search with our given board state is quite simple. Just have a parameter for the depth and board, a base case, and calling depthSearch again on all possible board states/moves. The only thing missing is our analysis of the given boards.

4.3 Points System



```
---------------
|    |    |    |
|  1 |  7 |  4 |
|    |    |    |
---------------
|    |    |    |
|  3 |    |  2 |
|    |    |    |
---------------
|    |    |    |
|  6 |  8 |  5 |
|    |    |    |
---------------
```

Figure 7: Piece 1 has a score of 0 since it is in the proper place. Piece 2 has a score of 2, since it is 2 moves away. Piece 3 has a score of 3, since it is 3 moves away. This gives the entire top row a score of 5.

Scoring a given board would allows us to compare it to other board states to determine which is better. At first glance, you can think of scoring a board by how many numbers are in the right spot. This would work, but the score would not reflect which boards are *closer* to having their numbers in the correct spot. To combat this, we scored boards based on the minimum number of moves a piece would have to make if there were no obstructions (any other pieces). Figure 7 showcases the point system. The top row in the board found in Figure 7 has a score of 5. The objective is to get a score as low as possible, and eventually 0, showing you have completed the row/column.

4.4 Putting it All Together

By using these 3 ideas. We can solve a row, as shown by the method above in Figure 6. We check to see if the top row has a score less than the current best board, if it does, then it becomes the best board. We then keep searching the next possible moves. This will get all possible moves, but this will certainly be taxing on computing power and will certainly chug. There can be anywhere between 2 - 4 possible moves from each board state, so it could grow at an exponential rate. However, there are many efficiencies we can make to improve the performance.

5 OPTIMIZING SEARCHING

As currently outlined, the sliding puzzle solver is horribly inefficient and would take very long to solve even a basic 3 by 3 board. However, with a few tweaks and pruning strategies, we can great increase the speed of our AI.

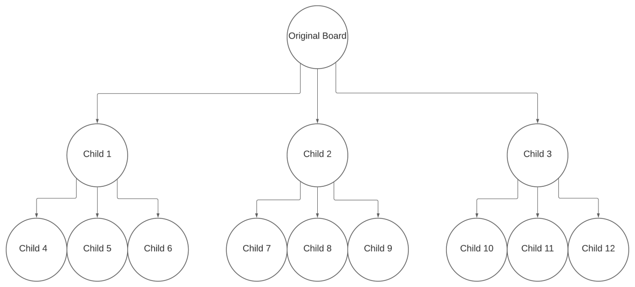


Figure 8: Each board will have up to 3 possible children, encompassing every possible move that can be made.

5.1 Avoiding Repeat Moves

A very basic way to greatly reduce the number of searches, we can avoid repeating the last move. Say we just moved a piece, if we moved that same piece, it would end up right where it started. To implement this, we simply had each board state save their last move, and when returning all possible moves, it would void the last move made and not consider it a potential move. This lowers the possible moves from 2 - 4 to 1 - 3 for any given board state. But more improvements can be made.

5.2 More Frequent, Lower Depth Searches

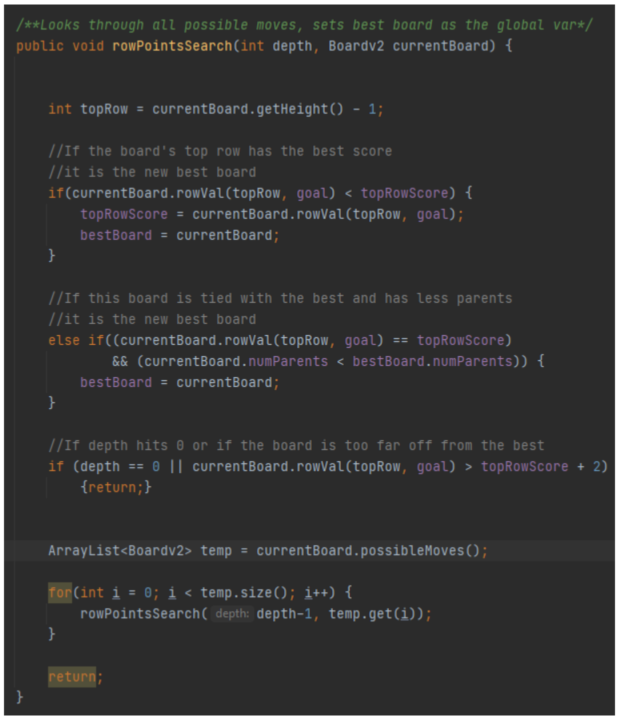
Rather than trying to solve an entire row with say, a depth of 30, we can try to iterate the recursion with a smaller depth, say 15. Remember, the depth search will return the best scoring board state with the smallest number of required moves (later we will explain how we know it is the fewest moves possible). So we would do a depth search of 15, take the output and depth search it again. If we assume, on average, the board has 2 possible moves in each state, the number of boards it is must crunch through is as follows in our two scenarios:

(1) We do a single depth search with a depth of 30, $2^{30} = 1,073,741,824$ boards to analyze
(2) We do two depth searches, each with a depth of 15, $2^{15}+2^{15} = 65,536$ boards to analyze

As you can see, by using less depth more frequently, we can achieve similar (not quite the same but close enough) results to a much larger depth search. In this case, we searched *16,384* times fewer boards with this approach, an extraordinary amount.

5.3 Finalizing Efficiencies

Now that we have optimizations, we can further improve our row depth search. For example, we can add to our base case to add addition pruning as seen in Figure 9. If the depth is 0 OR if the current board has a score much greater than the best score (in this



```
/**Looks through all possible moves, sets best board as the global var*/
public void rowPointsSearch(int depth, Boardv2 currentBoard) {


    int topRow = currentBoard.getHeight() - 1;

    //If the board's top row has the best score
    //it is the new best board
    if(currentBoard.rowVal(topRow, goal) < topRowScore) {
        topRowScore = currentBoard.rowVal(topRow, goal);
        bestBoard = currentBoard;
    }

    //If this board is tied with the best and has less parents
    //it is the new best board
    else if((currentBoard.rowVal(topRow, goal) == topRowScore)
            && (currentBoard.numParents < bestBoard.numParents)) {
        bestBoard = currentBoard;
    }

    //If depth hits 0 or if the board is too far off from the best
    if (depth == 0 || currentBoard.rowVal(topRow, goal) > topRowScore + 2)
        {return;}


    ArrayList<Boardv2> temp = currentBoard.possibleMoves();

    for(int i = 0; i < temp.size(); i++) {
        rowPointsSearch( depth: depth-1, temp.get(i));
    }

    return;
}
```

Figure 9: More complex showcase of a depth search for a given row

case more than 2), stop looking. Remember, a board wants to have a score of 0. By pruning moves that are wildly off from the current best board, we can reduce the number of boards we look at even more. From our testing, 2 is the most a board needs to deviate from the best in order to get all pieces into a row or column when there is only a single blank.

6 MORE ON BOARD DESIGN

With the methodology put forward, sliding puzzles can now be solved. There are a few more tweaks implemented to ensure that the program can display these boards and work slightly better.

6.1 Cloning Boards

When passing and assigning values, Java will always pass by value for primitives and by address for anything else, such as objects. We will discuss later the difficulties with this, but we created a clone method method to perform a deep copy of the board, to perform the potential move and not clobber the parent board

6.2 Parent Boards

During this cloning process, we have the new child board store the parent board. This allows us to see all moves that led up to the current board state. Once we have a board with the solution, we can see all moves before it to display them to the user. There is no use in a program that just spits out the complete board without showing us the steps, who is to say it isn't cheating and just displaying the goal? Furthermore, we keep a *numParents* variable for each child.

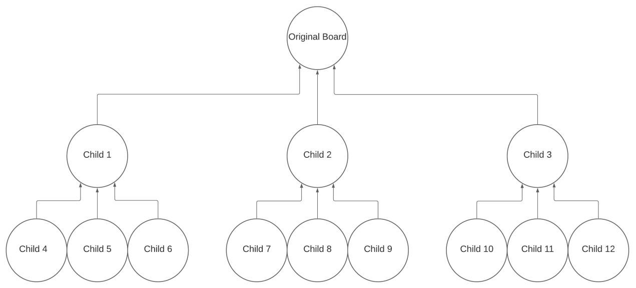


Figure 10: A simple chart showing how children can point to their parent boards.

This lets us known how many parents/moves it took to get to the current board state. When depth searching and comparing a board to the best board like in Figure 9, in the event of a tied score, the board with fewer parents/moves will be crowned the best board. This ensures us we are getting the fastest solution to a given board.

7 DIFFICULTIES AND POTENTIAL IMPROVEMENTS

We certainly did face difficulties and challenges while creating this AI. It is the first time either group member has ever created a program with this style of depth search and pruning requirement. We also did not manage to finish and make the ability to trial 2 blanks. That being said, the group is aware of other improvements it can make.

7.1 Java Deep Copying

Java's pain when it comes to cloning objects was also a striking point for the group. We originally had a Piece class, which held the data for each piece, including its value, position, and potentially even its score. This Piece class was used by our original Board class. The issue came with cloning these objects to look at potential moves. Java only passes primitives by value, objects are passed by reference. This annoyance led to a major refactoring of a different Board class that did not utilize a Piece class and used as many primitives as possible to makes cloning easier.

7.2 Time Constraints

Starting and finishing the project within 3 weeks was certainly a challenging feat, along with this given report. With that being said, the group is happy to have served a working product, though we can see the faults it has. Both members work near full-time along with other classes, so that certainly makes it harder to work on the given project as much as they would have liked. With more time, they are certain more things would have been ironed out.

7.3 Code Inefficiency

There are certainly points where there is inefficient code, such as using ArrayLists, which are known to run slowly. Furthermore the choice of Java when speed is a big priority is also a questionable choice, a language like C would have suited better. The group, however, wished to use Java for better practice with it. As for the inefficiency, they stem from time crunch. We could optimize the code even better, however we cared more about optimizing where it mattered, especially with pruning. This code can certainly be made more efficient and there can certainly be less loops.

7.4 Multi-Threading Output

The program would run much nicer, and could probably have a deeper depth, if we utilized threads. We could have a thread dedicated to a slow, board per second out, and another thread looking for next board states. As it stands now, the board displays after each recursive depth search ends, so it may display up to 16 or so boards. However, while it displays these boards, it simply sleeps for a seconds then displays the next board rather than computing in the background. This would make the solver appear seamless to the human watching it solve.

8 IMPLEMENTING 2 BLANK SPACES

The group in confident in their ability to create a 2 blank sliding puzzle. However, primarily due to time constraints and the amount of refactoring required, we were unable to do so for this project. That being said, we did plan out how to implement a board with this capability.

8.1 Many More Possibilities

Currently with 1 blank space, barring a repeat move, there are only a max of 3 possible moves. For 2 blank spaces, barring a repeat move, there are 17 possible moves. There are 16 ways both can move at the same time, and 2 in which one blank moves and the other still. Discounting the last move brings it to 17. That means there are many more board states to look at each level, we would have to look with much less depth. However, the board would also require less moves to complete the board with 2 available blank spaces. Rather than say 50 moves, it may only take 20, so that less depth may not end up mattering.

8.2 Better Pruning

With 2 blank spaces, we can prune even more aggressively. Look back to Figure 9. Remember how we discard boards that are more than $2{+}bestRowScore$? We do this to allow flexibility and allows the board to move already correct pieces to help get everything where it needs to go. With 2 blanks, the board will never have to move an already correct piece, this means we can be more aggressive and try discarding any board that is $1 + bestRowScore$ and maybe even try discarding everything greater than the bestRowScore. While we have many more possibilities to look at, we can get the the solution with less total moves and can discard more boards. The group questions that 2 blanks might run even faster than 1 blank.

9 GROUP TAKEAWAYS

This is the first time for anyone in the group to really work on creating a system that dynamically solves the set problem. Using concepts from other pathfinding applications, sorting algorithms, and various programming techniques to both create and solve the board, we are happy with the efficiency of our program in the given the time frame. Further improvements could be made as described above, and may possibly be done in the future. The group has certainly become more thoughtful when it comes to performance big picture rather than sweating the small stuff.

10 CITATIONS

Citations Below, all else rests in our GitHub:

[1] “How to check if an instance of 15 puzzle is solvable?,” 05-Oct-2020. [Online]. Available: https://www.geeksforgeeks.org/check-instance-15-puzzle-solvable/.

[2] O. Gordon, Y. Filmus, and O. Salzman, Revisiting the Complexity Analysis of Conflict-Based Search: New Computational Techniques and Improved Bounds, Apr. 2021.

[3] R. Stern, N. R. Sturtevant, A. Felner, S. Koenig, H. Ma, T. T. Walker, J. Li, D. Atzmon, L. Cohen, T. K. S. Kumar, E. Boyarski, and R. Bartak, Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks, Jun. 2019.