# *OpenNP:* A Domain Specific Language for NP-Complete Graph Problems *

Yucen Li
yucenl@andrew.cmu.edu

Sean Zhang
xiaoronz@andrew.cmu.edu

**Abstract**

NP-complete problems have many practical applications in the real world. However, due to their large runtime complexity, they can be infeasible to run. Additionally, NP-complete problems are difficult to implement in parallel due to challenges in problem representation, determining the full solution space, as well as managing search across different threads. We propose *OpenNP*, a domain specific language for the solving and parallelization of NP-complete problems. To create an instance of an NP-complete problem, users only need to implement the verification method. To run the problem on a specific graph, simply pass the graph into the problem instance. The language abstracts away much of the required implementation, greatly simplifying the necessary input from the user.

## 1 Introduction

NP-complete problems are interesting because they can be verified in polynomial time, but no known algorithms exist for finding the solution in polynomial time. Therefore, given a NP-complete problem, the only correct way to determine the solution is to use the brute-force method and try every single possible configuration within the solution space. The number of possible solutions is exponential with respect to the input length, and the search takes a very long time to run.

However, NP-complete problems can be incredibly useful for optimizing real-world tasks. For example, the solving of graph coloring, and if a graph can be colored in $k$ colors such that no node is connected to another node of the same color, can be extended to work scheduling problems and minimizing the total time needed to perform all jobs. The solving of Hamiltonian Path can be applicable to finding the shortest path through a series of endpoints.

*Our project is available at `http://www.andrew.cmu.edu/user/xiaoronz/`.

Additionally, even though these problems have useful applications, they can be extremely tedious for programmers to write properly. The first major point is how the graph is represented. Different data structures, such as adjacency matrix vs adjacency list, result in vastly different runtimes depending on the nature of the problem. Furthermore, graph problems often have low arithmetic intensity due to the randomness of the graph access. To increase this, many different methods such as sharded graphs and compressed edge lists can be used; however, this requires significantly more effort from the client-side, as the programmer must be familiar with these optimizations.

Another point of difficulty is determining the full solution space of an NP-complete problem. It is easy to create iteration methods which are very inefficient and have long runtimes. It is also easy to have duplicate graphs within the solution space, or even miss possible configurations all together. This can be very challenging for the programmer, and they will need to be very careful throughout their implementation to make sure that the output is correct.

Even if the user has correctly implemented the sequential algorithm, large steps need to be taken before the program can correctly run in parallel. Ideally, the total amount of work should be evenly among each thread, with little overhead demanded for this scheduling. To do this, the user needs to select a logical model of communication and manage data sharing between threads. This requires a high level of knowledge, and there is a lot of potential for the automation of this task.

In our project, we create *OpenNP*, a domain-specific programming language dedicated to the parallel solving of NP-complete graph problems. While the problems still take exponential time to solve, the search for the solution can now be optimized across many processors and threads, greatly decreasing the burden on one. Furthermore, the language greatly decreases the amount of work a programmer needs to compute the solution. Instead of slowly checking all of the steps listed above, the user only needs to select the type of problem, and write a validation function to check the state of a graph for correctness.

## 2 Language Design

We chose to restrict the domain of the language to NP-complete problems which can be represented through an undirected graph with labeled nodes and unlabeled edges. This covers a wide range of NP-complete problems while remaining specific enough for optimal speedup, but may require some initial thought from the client for how their NP-complete problem should be represented in this structure. For example, it is possible to check for the existence of $k$-clique, $k$ strongly connected vertices within a graph. This problem

can be represented by labeling each node as 0 or 1 corresponding to its inclusion or exclusion in the set of $k$ vertices.

To prototype, we implemented the language in Java. Our language features two main components: the graph representation as well the as the problem representation.

## 2.1 Graph Representation

To represent the undirected graphs, we number each node as an integer between 0 and $n-1$ where $n$ is the number of nodes in the graph. The graph also stores the label of each node as well as its list of neighbors.

The public API is as follows:

```
// Constructor:
Graph(int size);

// Method for graph creation:
void addEdge(int node1, int node2);

// Methods for checking correctness:
int getSize();
int[] getNodes();
int getNodeLabel(int node);
void setNodeLabel(int node, int label);
Set<Integer> getNeighbors(int node);
```

## 2.2 Problem Representation

While studying the approach to solving NP-complete problems, we found that each problem requires the answering of three main points:

1. What is the polynomial-time validation algorithm to see if the correct solution has been reached?
2. How do we iterate through the complete set of solution spaces?

3. How can we parallelize the search across the specified number of threads?

It seemed clear that it would be illogical to find a generalized implementation for the polynomial-time validation algorithm, since this is specific to each problem. Therefore, we decided that our language would allow clients to write their own problem-specific validation checker of a graph using the representation discussed in section 2.1.

Based on this analysis, we decided to represent each NP-complete problem as a `Problem` class, consisting of two methods for the client to implement.

```
abstract boolean check(Graph graph);
abstract boolean solve(Graph graph, int numThreads);
```

The iteration through solution spaces was a key challenge. It did not seem to be specific to each problem. For example, problems such as independent set and clique could both be represented as nodes labeled 0 or 1, where the full of solution space is the $2^n$ different labelings, where each node is mapped to one of two labels.

However, for problems such as Hamiltonian path, we realized that the best representation of the problem was to label each node as a permutation of the size of the graph. Then, the graph was correct if the nodes could be visited in order, i.e. there was an edge connecting each pair of nodes in increasing order. While this labeling of nodes could also be generalized to problems such as Hamiltonian cycle, it was vastly different from the mapping described above.

We decided that there was an opportunity here for *OpenNP* to implement this behind-the-scenes, without additional input from the client. To do this, we created two types of problems: `MapProblem` and `PermuteProblem`. `MapProblem` was used to represent problems where each node could take on any of a finite list of labels, and `PermuteProblem` was used to represent problems where the nodes are labeled in a specific order.

It was also clear that the parallelization of work across threads could be generalizable if we knew the solution space: given any problem with its complete set of $n$ possible solutions, it seems logical to evenly distribute each of these checks among all of the threads. Therefore, paired with either `MapProblem` or `PermuteProblem`, the `solve` function could also be abstracted away using the programming language, and the client would not need to implement any of the work sharing.

### 2.2.1 MapProblem

Some NP-complete problems involve the mapping of each node to a fixed number of labels. For example, in 3 coloring, each node can be mapped to color 1, 2, or 3. In vertex cover, each node can be labeled as 1 or 0 to represent their respective inclusion and exclusion in the set of selected vertices.

For these types of problems, we created the class `MapProblem`. This class requires the additional parameter `numLabels` to be explicitly defined. Using `numLabels`, we can then automate the parallelization of the search through all solution spaces. This class implements the method

```
boolean solve(Graph graph, int numThreads);
```

so the only method needed for users is

```
abstract boolean check(Graph graph);
```

For example, to create the 3-coloring problem, the only client-side input necessary is the implementation of `check`. Using the Graph API and functions, it is simple to check if a graph contains a valid 3-coloring:

```
boolean check(Graph graph) {
    for (int node : graph.getNodes()) {
        for (int neighbor: graph.getNeighbors(node)) {
            if (graph.getNodeLabel(node) == graph.getNodeLabel(neighbor)) {
                return false;
            }
        }
    }
    return true;
}
```

Then, to solve the problem, we only need to initialize the problem, and then solve it:

```
Graph graph = new Graph(n);
graph.addEdges(...); // add edges to graph
Problem problem = new ThreeColoring();
```

```
problem.solve(graph, numThreads);
```

The `MapProblem` can be applied to a variety of NP-complete problems such as vertex cover, independent set and clique.

### 2.2.2 PermuteProblem

Other NP-complete problems involve the permutation of nodes. For example, Hamiltonian Path, which checks if a path exists two vertices where every vertex is visisted exactly once, can be checked using a specific ordering of the nodes in the grpah.

For these types of problems, we created the class `PermuteProblem`. We automate the parallelization of the search through all solution spaces. This class implements the method

```
boolean solve(Graph graph, int numThreads);
```

so the only method needed for users is

```
abstract boolean check(Graph graph);
```

For example, to create the Hamiltonian path problem, the client creates a class to extend `PermuteProblem`. Using the Graph API and functions, it is simple to check if a graph is avlid:

```
int nodeLabel = graph.getNodeLabel(node);
if (nodeLabel == graph.getSize() - 1) {
    return true;
}
for (int neighbor: graph.getNeighbors(node)) {
    if (graph.getNodeLabel(neighbor) == nodeLabel + 1) {
        return true;
    }
}
return false;
```

Then, to solve the problem, once again, we only need to initialize the problem, and then solve it:

```
Graph graph = new Graph(n);
graph.addEdges(...); // add edges to graph
Problem problem = new HamiltonianPath();
problem.solve(graph, numThreads);
```

This problem can also be generalized to other NP-complete problems such as Hamiltonian cycle and longest path.

# 3 Implementation and Speedup

As a domain-specific language, *OpenNP* uses knowledge about NP-complete problems to make decisions regarding the implementation of the problem without input from the programmer.

## 3.1 Data Structure

A key component in the efficient solving of NP-complete graph problems is the representation of the graph itself. There are many different possibilities for its representation from common data structures such as adjacency matrices, adjacency lists to more complex solutions learned in class, such as sharded graphs, compressed edge lists, etc. Because NP-complete problems have exponential runtime, it is unrealistic for the graphs to be extremely large, as the runtime grows extremely quickly as on the number of nodes increases. Therefore, many of the graph compression techniques are not necessary.

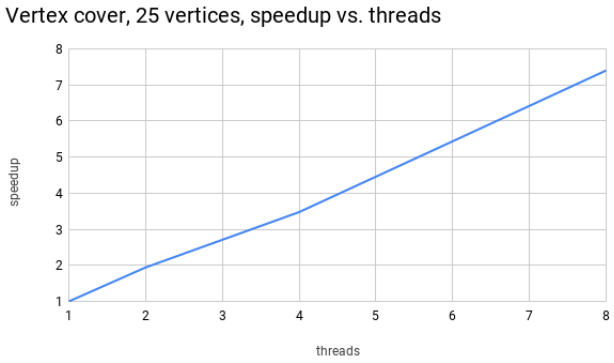
When comparing between adjacency matrices and adjacency lists, we noticed that the constant-time check for the existence of an edge in the adjancecy matrix provided a faster overall runtime. Therefore, *OpenNP* uses an adjacency matrix to represent the graph because of its simplicity and effectiveness.

## 3.2 MapProblem

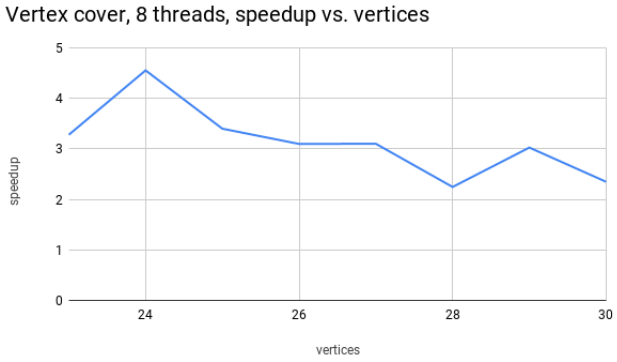
For MapProblem, we observe that (1) checking each assignment can be done in polynomial time and is generally consistent across the different labeling of graphs, and (2) the space of all possible assignments is relatively simple, and hence it is easy divide all assignments into groups where each group roughly has the same amount. Hence, to schedule the assignments among the threads, we chose to use static scheduling, which provides a good balancing of work among threads.

To implement static scheduling, We divided the work among the threads by fixing the first $k$ labels for nodes. For example, if each node could map to 2 labels and we had 4 threads total, we would let thread 1 check all the cases where node 1 is labeled 0, node 2 is labeled 0, thread 2 check all the cases where node 1 is labeled 0, node 2 is labeled 1, etc. Within each thread, an iterator would be initialized to keep the first $k$ labels fixed while going through every single label possibility for the remaining nodes. At the end of all of the checks, the thread would output true if the correct answer was foun. The main thread then compares all of the threads, and outputs true if any single thread contained a valid solution.

To test the speedup of MapProblem, we implemented solvers for 3-coloring and vertex cover, and we ran them on afs for different amount of threads. The plot of speedup versus number of threads is shown in the graphs below.




From the speedup vs threads plot, we can see that we achieve very high speedup compared to number of threads. This is because the program can easily be divided evenly among threads, with little overhead. Additionally, there is very good workload balance since each configuration of the graph takes approximately the same time to check.




From the speedup vs. vertices plot, we see that the speedup is fairly consistent across different graph

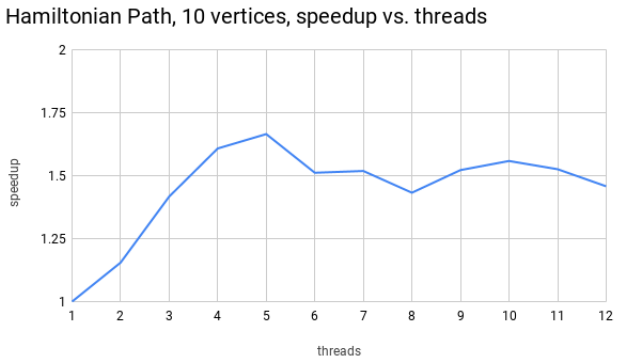
size. For smaller graphs, the speedup fluctuates a bit more since the running time of both programs are below 1 second, but the speedup stabilizes as the number of vertices increases.

### 3.3 PermuteProblem

PermuteProblem was parallelized using dynamic scheduling. This is because we could not easily have a static division of workload which was balanced: for an algorithms that statically divides the $n!$ assignments into $k$ threads, either the threads have an uneven amount of assignments, or the algorithm causes a overhead and slows down the speedup.

To implement dynamic scheduling, we wrote an iterator which used Heap's algorithm to return every possible permutation of the list. Using this iterator, we then gave each thread 1000 different labelings of nodes. Once the thread checked each of these configurations, the iterator would give it 1000 more labelings. This was continued until all of the possible permutations were checked.

The plot of speedup versus number of threads for the Hamiltonian path problem is shown below.




The speedup for this was significantly worse than the speedup for MapProblem. We hypothesize that this is mostly due to the overhead of the iterator. While we time the code, it appears that many threads were free but were work-starved because the iterator had not yet passed them the next set of tasks to complete. Therefore, there may be a need to explore different algorithms for permutation generation which are either faster or are capable of dividing the work among multiple threads.

# 4 Future Work

PermuteProblem was not able to achieve the speedup of MapProblem due to the inefficiencies with task scheduling. More work can be done to find different permutation generation algorithms, or to better divide the work among available threads.

While MapProblem and PermuteProblem span different possibilities, they do not account for weights in nodes or edges. The inclusion of weights would better account for more problems, such as the graph bandwidth problem and subset sum. The creation of new problem types (such as `WeightedProblem`) would increase the domain of our language.

Additionally, we have only experimented with one model of communication. It would be interesting to compare our results using shared address space with that of message passing or the hybrid model to see if the speedup could be improved.

# 5 Conclusion

*OpenNP* is effective in making the representation and solving of NP-complete problems easy for the general programmer. By limiting the decisions which need to be made, the language reduces the burden of implementing these types of problems. Furthermore, the language parallelizes the work well and leads to good speedup.

# 6 References

We referred to Heaps Algorithm to generate permutations. We also used different Java libraries such as omp4j to preprocess the code and run in parallel.

# 7 Work Distribution

For this project, Yucen first implemented the solving and parallelization of 3-coloring. After comparing the results of her work with Sean's implementation of vertex cover, she then discussed the design of the language with Sean and jointly came up with two main representations, MapProblem and PermuteProblem. Yucen

implemented much of the initial layout of the language by creating the structure and creating the design and interactions for the Graph, Problem, MapProblem, and PermuteProblem classes. She then implemented the static scheduling for MapProblem using the creation of multiple threads and their respective iterators, which partitioned the solution space. Yucen also wrote the class ThreeColoring to test her implementation of MapProblem. She then implemented the dynamic scheduling of PermuteProblem, and wrote the class HamiltonianPath to test the implementation. After discussing the details of the final writeup and poster presentation with Sean, she wrote the summary, introduction, language design, future work, and conclusion, and part of the implementation section of the report. She also worked on half of the pages of the final presentation.

Sean first implemented vertex cover for the sequential case and then optimized it for the parallel case. He then worked on designing the language together with Yucen. After the MapProblem and PermuteProblem classes are finished, Sean modified the implementation to generalize the language to other problems, including graphs with weights on vertices. He also attempted a static implementation of PermuteProblem. For the final report, Sean run the program to create the plots and wrote parts of the implementation section. For the presentation, he worked on the other half of the pages.

For the division of credit, Yucen has 70% of the credit, while Sean has 30%.