**Max Independent Set Problem Analysis**

**Brute Force vs Greedy Approximation**

**California University of Pennsylvania**

**CSC 360 Analysis of Algorithms**

**Professor Sible**

**Fall 2020**

**Kevin Andor Scutaru**

**Nathaniel DeHart**

**Cole Stewart**

Before writing an algorithm for a computer, it’s always important to analyze it first. If an algorithm for a problem were found to be classified under non deterministic polynomial time (NP), a computer would be unable to run it within a reasonable amount of time (especially for large amounts of data). There are approximation algorithms that exist for these NP problems that can be solved within polynomial time and produce a sufficient enough result, however, they might not always be as accurate. The analysis of having a computer solve an NP problem via brute force algorithm versus an approximation algorithm shall be explored, using the Max Independent Set problem as a basis, including data on processing times and data accuracy using a program that has implemented both the brute force and approximation algorithms for said problem. Heuristic algorithms will also be explored, along with examples of implementation.

As the name might suggest, the goal of the Max Independent Set problem is to find the maximum set of vertices in a graph that do not have an edge connected to one other. Using the brute force method, the algorithm would check for edges between every possible combination of vertices. While checking every possible combination in the graph for an independent set would yield the most accurate results, it would also yield a complexity of Θ(2n), where n is equal to the amount of vertices in the graph (hence the NP classification), meaning it would run for an unreasonable amount of time as the amount of vertices that need to be checked go up. Using a greedy approximation method, the algorithm will act using the data it has in any particular moment, without looking ahead. In the case of the Max Independent Set problem, this will involve knocking out nodes with the least connections along with their connections and taking the nodes that had the least connections as the independent set. Since every single combination of nodes is not being checked this will yield a set of nodes within a reasonable amount of time, but for that same reason it might not always produce the absolute maximum set that the brute force algorithm would.

Below is a table showing the running times of a program with the implementation of the Max Independent Set problem using both the brute force and greedy approximation algorithms that were specified above:

|  |  |  |  |
| --- | --- | --- | --- |
| **Node Count** | **Edge Density** | **Brute Force Time (seconds)** | **Greedy Approx. Time (seconds)** |
| 1-10 | 25% | 0 | 0 |
| 11 | 25% | 0.001 | 0 |
| 12 | 25% | 0.001 | 0 |
| 13 | 25% | 0.003 | 0 |
| 14 | 25% | 0.006 | 0 |
| 15 | 25% | 0.017 | 0 |
| 16 | 25% | 0.038 | 0 |
| 17 | 25% | 0.066 | 0 |
| 18 | 25% | 0.157 | 0 |
| 19 | 25% | 0.322 | 0 |
| 20 | 25% | 0.617 | 0 |
| 21 | 25% | 1.593 | 0 |
| 22 | 25% | 3.016 | 0 |
| 23 | 25% | 6.724 | 0 |
| 24 | 25% | 12.965 | 0 |
| 25 | 25% | 28.962 | 0 |
| 26 | 25% | 60.894 | 0 |
| 27 | 25% | 127.281 | 0 |
| 28 | 25% | 277.69 | 0 |
| 29 | 25% | 474.635 | 0 |
| 30 | 25% | 690.099 | 0 |
| 31 | 25% | 2268.45 | 0 |
| 32 | 25% | 4254.67 | 0 |
| 35 | 25% | 27519.800 | 0 |
| 37 | 25% | 108746 | 0 |

In the table above, Node Count refers to the number of vertices total in the graph. Edge Density refers to the chance that two vertices are connected to each other when generating the graph (in the case of the data gathered for this table, Edge Density has consistently been set to 25%). Lastly, Brute Force Time and Greedy Approx. Time shows the time it took the program to run each algorithm respectively (in seconds). The program was run across multiple machines that are similar to each other, but it should be noted that time discrepancies could exist from one result to another. The table shows that as the node count increases, the time it takes for the brute force algorithm to finish processing increases, with noticeable change occurring starting around node 11 and upward, and drastic increases in time occurring around node 21. The full complexity equation for the program function using the brute force algorithm is O(2 + (6 + 7n + 2n^2) \* 2^n). In a stark contrast, the program had shown to finish processing the greedy approximation algorithm at time 0 all the way from node 1 to node 37. The full complexity equation for the program function using the greedy approximation algorithm is O(7n + 7n^2 + 3n^3).

Below is a table that shows the accuracy of the results from the approximation algorithm:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Node Count | Trial 1 Accuracy | Trial 2 Accuracy | Trial 3 Accuracy | Trial 4 Accuracy | Trial 5 Accuracy | Average Accuracy |
| 10 | 100.00% | 100.00% | 100.00% | 100.00% | 100.00% | 100.00% |
| 11 | 100.00% | 100.00% | 100.00% | 100.00% | 100.00% | 100.00% |
| 12 | 100.00% | 100.00% | 100.00% | 100.00% | 100.00% | 100.00% |
| 13 | 100.00% | 100.00% | 100.00% | 100.00% | 100.00% | 100.00% |
| 14 | 100.00% | 100.00% | 100.00% | 85.71% | 100.00% | 97.14% |
| 15 | 100.00% | 100.00% | 100.00% | 85.71% | 100.00% | 97.14% |
| 16 | 100.00% | 100.00% | 85.71% | 100.00% | 100.00% | 97.14% |
| 17 | 100.00% | 100.00% | 100.00% | 100.00% | 100.00% | 100.00% |
| 18 | 100.00% | 100.00% | 100.00% | 100.00% | 87.50% | 97.50% |
| 19 | 100.00% | 100.00% | 100.00% | 100.00% | 100.00% | 100.00% |
| 20 | 100.00% | 100.00% | 100.00% | 100.00% | 100.00% | 100.00% |
| Total |  |  |  |  |  | 98.99% |

The percent accuracy for each trial represents the number of nodes in the independent set found by the approximation algorithm divided by the true number of nodes in the independent set found by the brute force algorithm, which is confirmed to provide the most accurate results from previous testing. There were 5 trials conducted for each number of nodes (25% edge density still applied). The accuracy table shows that with the given trials, the approximation algorithm has an accuracy rate of 98.99% in comparison to the independent set produced by the brute force method. This Means that on average the approximation algorithm will usually find an independent set that is at least 98.99% as large as the true maximum independent set.

While the brute force approach and the approximation approach have been mentioned for solving NP-Complete algorithms, there is another approach that can be used for NP-Complete problems known as heuristics. Unlike the approximation algorithm or the brute force algorithm, a heuristic method would attempt to solve the problem based on previous observations to find a good answer to the problem in question. While it is not a NP-Complete problem, one example of a heuristic solution to a problem is the Best-First-Search pathfinding algorithm. The Best-First-Search algorithm is an algorithm based on Dijkstra’s algorithm which is made to find the shortest path from one point to another on a 2d map. By breaking up the map into small squares/nodes that either contain part of an obstacle or not, Dijkstra’s algorithm branches out in all directions from the start point until part of it reaches the end point. This would be considered more of a brute force approach. The Best-First-Search method instead assigns a “cost” to each node in the map with nodes further from the end point having a greater cost. The Best-First-Search method would weigh the cost of all adjacent nodes and create a path using the nodes with the lowest cost. When an obstacle is in the way of continuing with the lowest cost, then the adjacent node with the next to lowest cost would be used. This would continue until the end point is reached. While the Best-First-Search pathfinding algorithm may not always find the fastest path, this heuristic solution is much faster than Dijkstra’s brute force based algorithm. There is one algorithm that is better than either of these algorithms though.

The A\* pathfinding algorithm builds off of the heuristic approach that the Best-First-Search algorithm uses to find the fastest path between two points on a 2d map while still being much faster than Dijkstra’s algorithm. Along with having the regular cost (distance away from the end point) it also takes into account the distance away from the start point. A good way to imagine the difference between the Best-First-Search algorithm and the A\* algorithm is a scenario in which there is a U-shaped obstacle in between the start and the end with the opening facing the start. The Best-First-Search algorithm would go all the way into the U-shaped obstacle until it hit the back of the obstacle and would proceed to diverge around the inside of the obstacle until it reaches all the way around. This is an example where the Best-First-Search algorithm would not find the fastest path. On the other hand, the A\* algorithm would eventually get far enough into the U-shaped obstacle that the cost of going around it would be lesser than going into it first because of the distance from the start being incorporated into the cost as well. Once the A\* algorithm reaches the end point, it simply needs to establish the path it took. A way that this could be done is backtracking from the end point along a path of the checked nodes with the next lowest distance from the start point.

Writing this program has given a look at a prevalent and highly debated topic in computer science at the moment: Does polynomial time equal non-deterministic polynomial time? Whether or not it does, finding an answer would be extremely beneficial to the field, either allowing researchers to move on to a different topic or opening up a brand new avenue of solutions and algorithms that could expand possibilities in their respective areas. Greedy and approximation-based algorithms are not always the most accurate, but they are oftentimes the most efficient solution, especially when compared to a brute-force algorithm. So long as they are not applied to anything that needs to be fully accurate, they are a great resolution to the issues at hand for the time being. Heuristics-based algorithms are another effective way of obtaining data when the method is usable, allowing for fast-running programs and simply needs past observations and data to be implemented. In comparison, running a brute-force algorithm with massive amounts of data could take hours, days, months, or even years, and using it simply is not good enough, even if it is always accurate. As shown in the table above, running the program for 37 nodes took 108746 seconds (or 30.21 hours) to run, and although it was entirely accurate, it’s an unrealistic way of gathering the data, whereas running 37 nodes through an approximation algorithm obtained that data in less than a second, and with relative (98.99%) accuracy. Studying and applying heuristics and approximation algorithms is a necessity for the time being, as no one knows for sure whether or not polynomial time and non-deterministic polynomial time are equivalent, and they run at much more acceptable times with relative accuracy rather than algorithms, such as the brute-force algorithm, that check everything.

# 

# **References**

Cohen, J. (2018, July 17). *Path Planning for Self-driving Cars*. Retrieved from Towards Data Science: https://towardsdatascience.com/can-self-driving-car-think-6c9e8d939d60

Novoseltseva, E. (2018, December 11). *Heuristic Programming*. Retrieved from Softjourn: https://softjourn.com/blog/article/heuristic-programming#:~:text=In%20computer%20science%2C%20artificial%20intelligence,to%20find%20any%20exact%20solution.

Team, G. L. (2020, February 17). *Best First Search Algorithm in AI | Concept, Implementation, Advantages, Disadvantages*. Retrieved from Great Learning: http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html