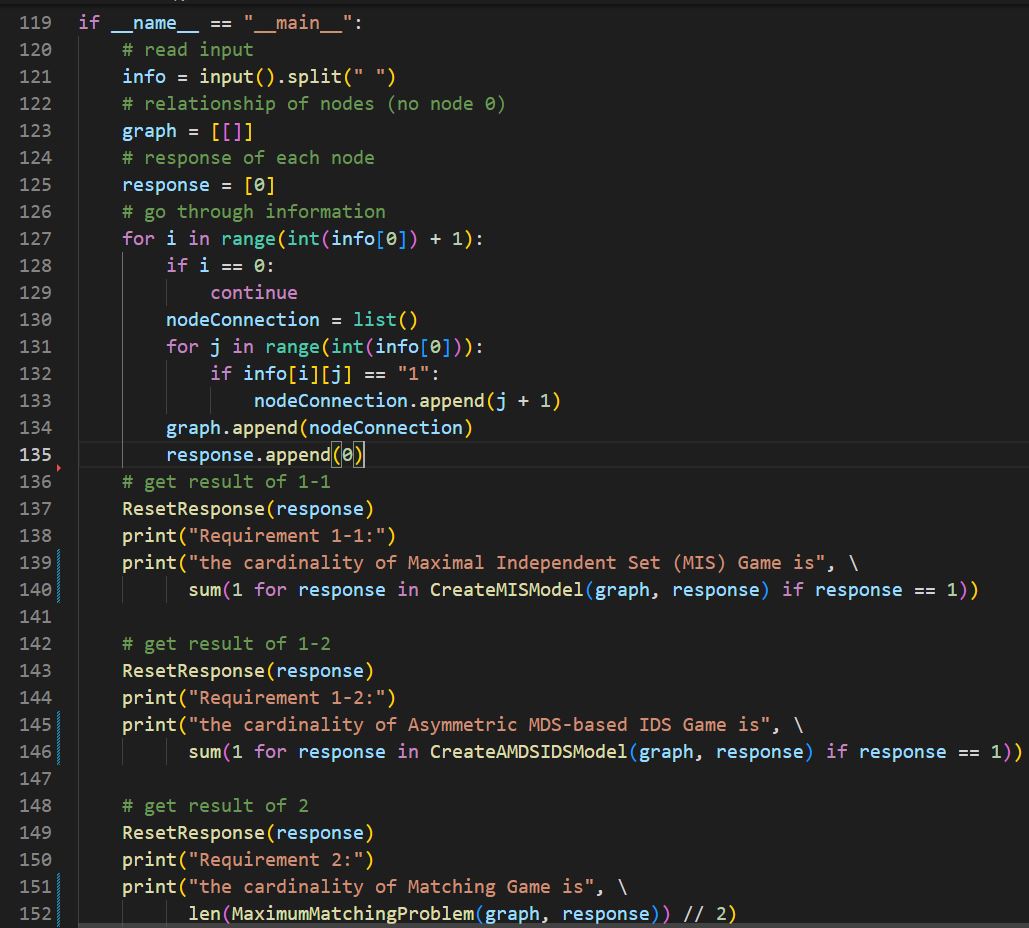
賽局理論與應用 Game Theory and Its Applications Assignment 2

Student ID: 312551077

Department: 資科工碩

Name: 薛祖恩

* **Main**



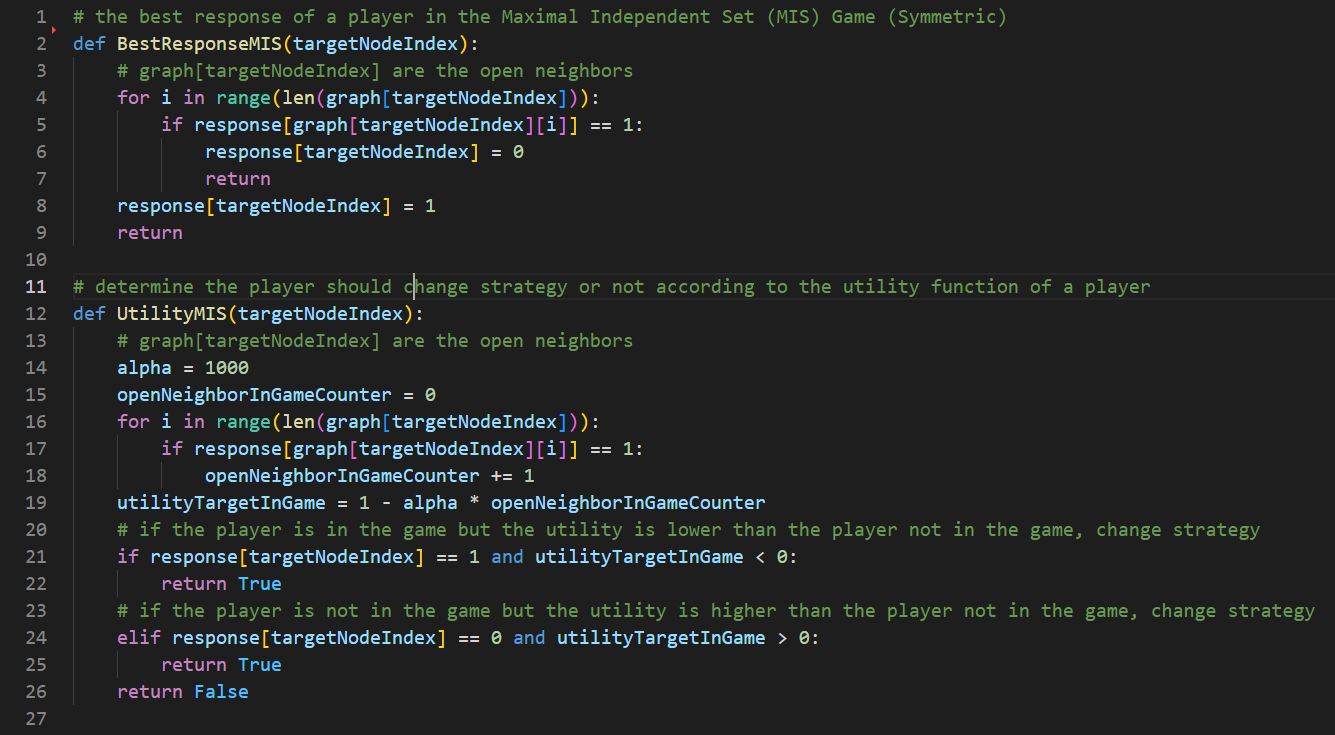
First, the code reads the input, stored into the variable *info*, then I declare two lists *graph* and *response* to store the open neighbors and the response of the node respectively. Note that there is n+1 elements in *graph* and *response* by setting the first element as empty list and 0, respectively. Since I want the index to be as the same as human’s perspective. *graph* is a 2-dimensional list, and *response* is a 1-dimensional list.

The next step is to deal with the user input, first, we split the input by a space. By doing this, the first element becomes the number of the nodes, the rest of the elements are the connection information of each node. For each connection information, if the input is 1, then the index (human perspective) of the connection information will be added to the variable *nodeConnection*, and after the iteration, *nodeConnection* will be added to *graph*, which stores the open neighbors for all nodes.

The rest of the code is just resetting *response* and calling the other methods, and showing the results.

For example: if the input is “6 010000 101100 010010 010010 001101 000010” as mentioned on the website, the outer for loop will run 7 times, ignoring the first element, the inner loop will iterate 6 times, the first connection information is “010000”, for node 1, it is only connected to node 2, thus “[2]” will be the first *nodeConnection* and stored into *graph*; for node 2, “[1 3 4]” will be the next *nodeConnection* and stored into *graph*, and so on. The last node, that is, node 6’s *nodeConnection* will be “[5]”.

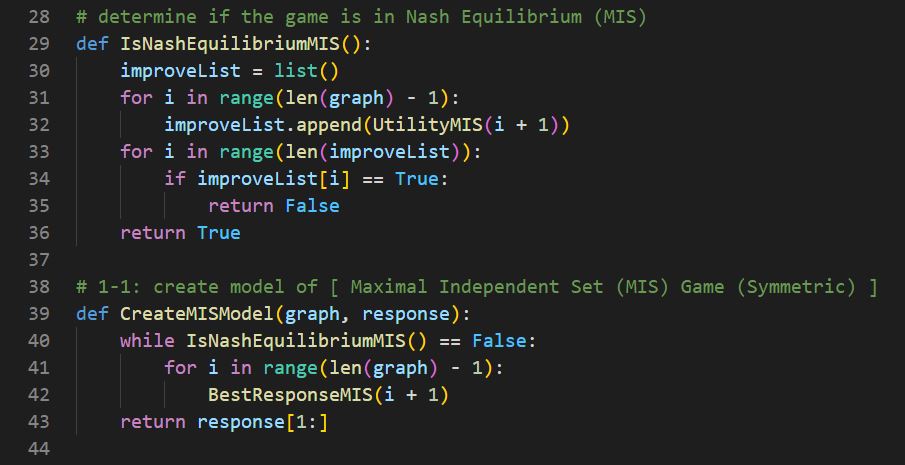
* **Requirement 1-1 (Maximal Independent Set (MIS) Game (Symmetric))**



Best response: for all open neighbors of node *targetNodeIndex*, I search the response and see if the response of the open neighbor is 1 or not. If there exists at least 1 node that is in the set, the response *targetNodeIndex* will become 0, which means the node decided not to be in the set. Otherwise, the node should decide to be in the set.

Utility: for all open neighbors of node *targetNodeIndex*, I search the response and see if the response of the open neighbor is 1 or not. If the open neighbor’s response is 1, then the utility of the current node will be decreased by α, if there are k open neighbors decided to be in the game, then the utility of the node will be:

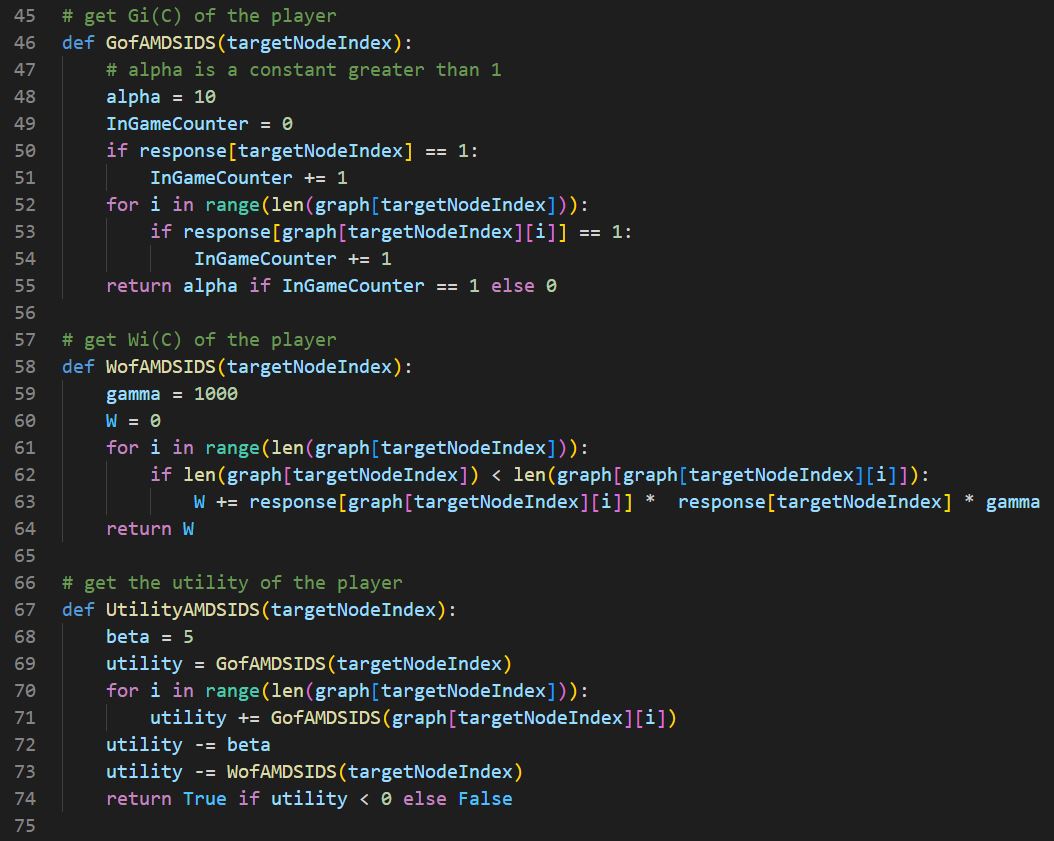
Since α is greater than 1, in my code, 1000. Thus, if there exists an open neighbor that decides to be in the set. The utility of the node to be in the set will become negative, which is much smaller than the utility to not be in the set, under this situation, the function will return true, indicate that the utility can be improved. We can apply the same logic to the opposite condition that I just mentioned. If the two conditions are not met, then the function returns false, means that the utility cannot improve for this node.



Determine Nash Equilibrium: first, I declare a list called *improveList*, if there exists a true in the list, that means the current set does not reach an NE. To complete this list, we must iterate through each node using the function *UtilityMIS*.

Run MIS Model: while the current set is does not reach an NE, find the best response for all nodes. Then return *response[1:]*, since the first element, that is, *response[0]*, is meaningless.

* **Requirement 1-2 (Asymmetric MDS-based IDS Game)**



gi(C): I declare a variable *InGameCounter*, that stores the count of the nodes that decide to be in the game (node *targetNodeIndex*’s open neighbors and itself). If the value of the variable is 1, return alpha, in this case, 10, otherwise, return 0.

Wi(C): for all open neighbors that has higher degree than node *targetNodeIndex*, we sum the return value up by gamma if the two nodes, both node *targetNodeIndex* and node *targetNodeIndex’s* higher degree neighbor decides to be in the set.

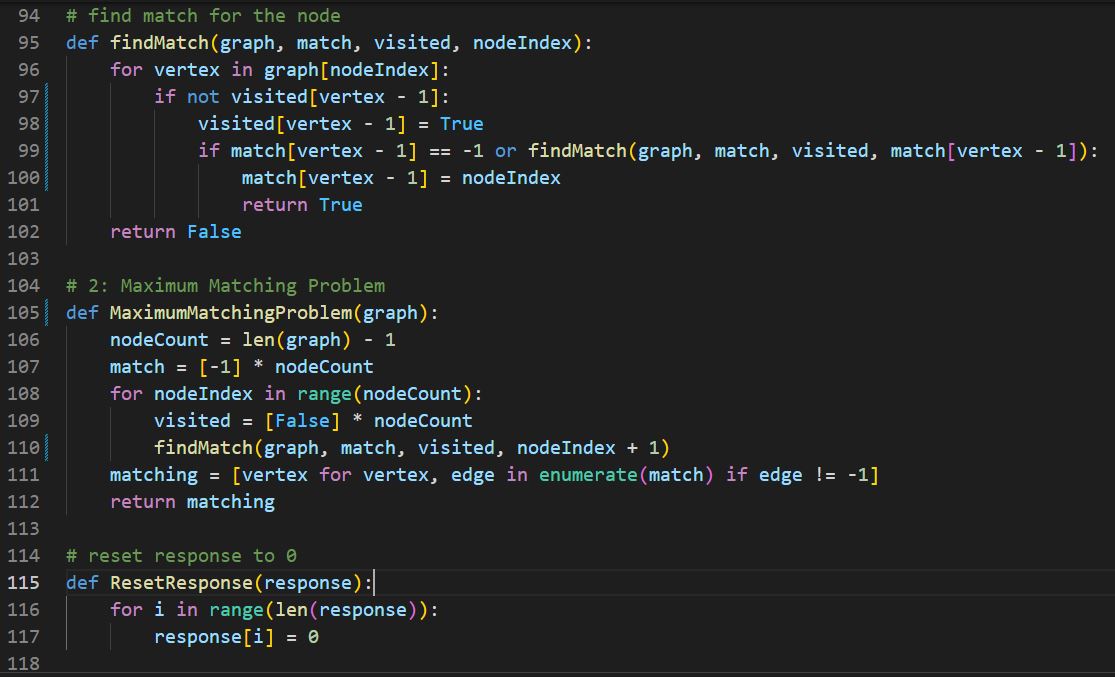
Utility: we sum up the g(C) function, C are the node *targetNodeIndex’s* open neighbors and node *targetNodeIndex* itself, then we decrease this value by W(C) and beta, in my case, 5. If the function returns true, that means the utility can be improved, else return 0.



Determine Nash Equilibrium: The same logic as requirement 1-1. I declare a list called *improveList*, if there exists a true in the list, that means the current set does not reach an NE. To complete this list, we must iterate through each node using the function *UtilityAMDSIDS*.

Run Model: If it does not reach an NE, we change the response from 0 to 1 or from 1 to 0 for all utility is smaller than 0 (which returns false in the *UtilityAMDSIDS* function).

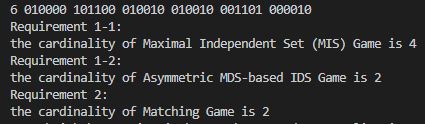
* **Requirement 2 (Maximum Matching Problem)**



We solve this problem by using DFS, there are two new lists, *match* and *visited*. *match* is the list that stores the matching node for each node. *visited* is the list that records if the node is visited or not. If there is an edge and the node is unmatched, we match those two nodes.

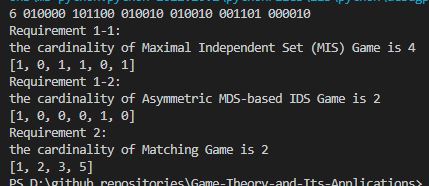
*ResetResponse* is a function that sets all elements in the list *response* to 0.

* Results



You can also print out the *response* for requirement 1-1, 1-2, and the result of the maximum matching problem.





For requirement 1-1, node 1, 3, 4, 6 are in the set, thus the cardinality is 4.

For requirement 1-2, node 1, 5 are in the set, thus the cardinality is 2.

For requirement 2, match nodes are (1, 2) and (3, 5), thus the cardinality is 2.