Project:
Algorithmic complexity and graphs

Group: Johannes Timmreck, Alexander Prosch, Jonas Conrad

Table of Contents

2. Choosing binoms in a company (matching)

2.1)

The two chosen heuristics are rather simple.
The first heuristic is a greedy algorithm which sorts the nodes at the beginning regarding the number of their neighbours. It then starts matching the nodes by beginning with the node with the least neighbours. It then matches the node with the first unmatched node in its list of neighbours.
It will be referred to as “Sorted Greedy”.

The second used heuristic runs after a similar concept, but instead of just sorting once in the beginning, the list of nodes is sorted every time a matched pair is removed. This way the next matched node will always be one, with a minimal amount of possible matches left. The node with which it will be matched will be similar to the first heuristic chosen greedily to be the first one. After the 2 nodes are matched they are eliminated from the list of nodes and all neighbour node lists. This allows for a new sort by the least number of unmatched neighbours before continuing the matching process.
It will be referred to as “Multiple Sort”.

To compare the two heuristics the main of the algorithms.py file will generate a random graph and run the two heuristics. It will output matching size and time in the console and put images of the graph and matches into the images directory. Additionally, the algorithm of the networkx library will also be run and its statistics output for comparison.
It will be referred to as “Networkx”.

The generated graphs will be dumped into the data directory in python readable form. The graphs can later be imported using the pickle and networkx libraries (like in the `import_graph` function in graph_helper_functions.py).

The timed version of this run will be significantly faster, than in the evaluation step in 2.3, as the visualization step adds a significant amount of time, which will be skipped in the evaluation step.

2.2)

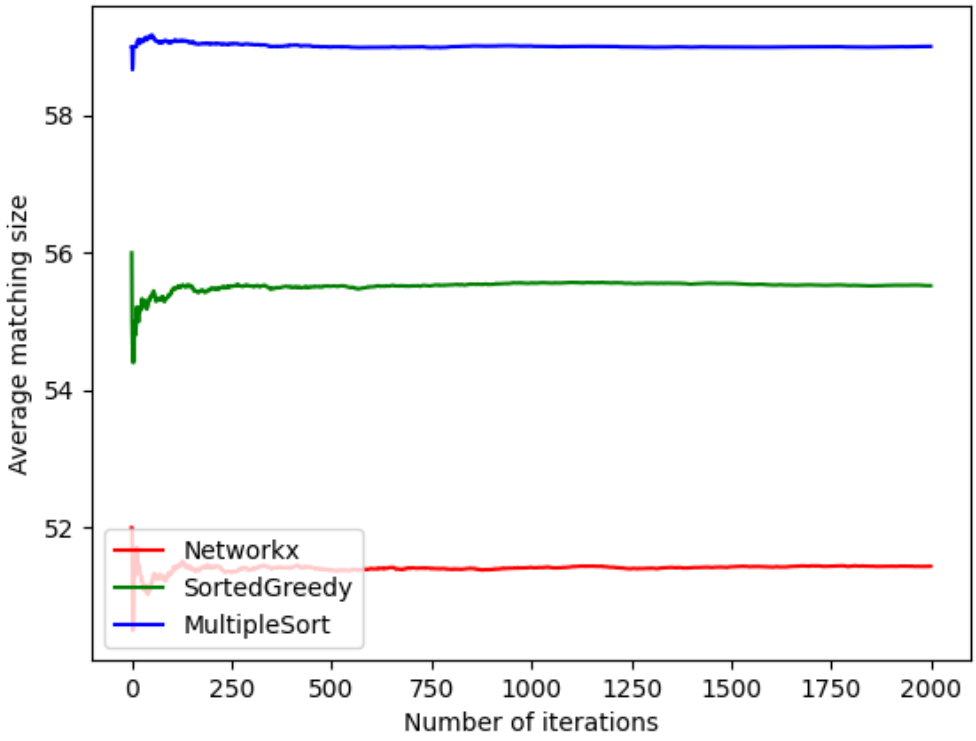
The test_matching.py file contains a function `test_matching`. It requires a networkx Graph object and a list of matchings and checks if the list contains a valid matching. Additionally, it checks, if the matching is a Maximal or a Maximum matching.
This function will always be invoked in the algorithm functions to check if the found matching is valid. If the matching is not valid, the algorithm will output an empty list instead of the invalid one. If the algorithms are run in visualize mode (they will always be when executing the algorithms.py).

To validate the matchings the Networkx library functions `is_matching`, `is_maximal_matching and is_perfect_matching` were used.

2.3)

To have a data pool which is large enough to compare the different heuristics the evaluation.py generates a specified amount of random graphs and then executes all 3 algorithms for each graph. The evaluation cuts out the visualization step of each graph to significantly reduce the measured runtime of the algorithms. The measured values can be found in the evaluation directory.
The number of iterations for this evaluation was set to 2000.

Matching



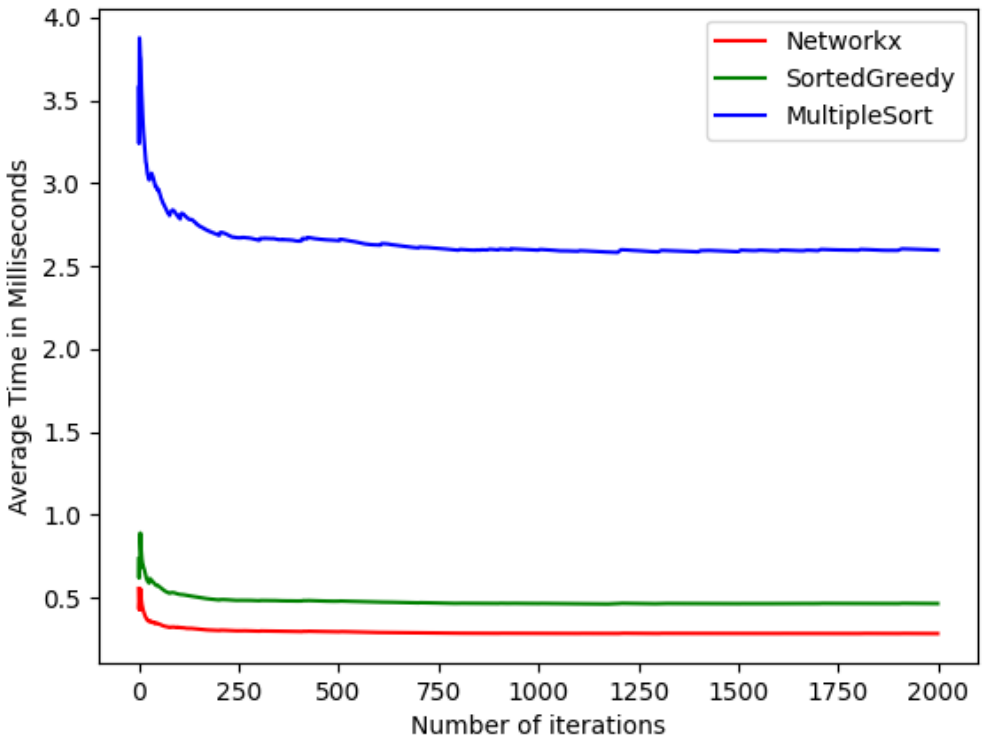
This plot shows the average amount of found matches after each iteration. As can be seen, the Multiple Sort algorithm performs much better than both the Networkx algorithm and the Sorted Greedy algorithm. As well as all algorithms converging to their respective average after about 250 to 500 iterations.

Algorithm	Average Matchings	Largest Matching (occurrences)	Smallest Matching (occurrences)
Networkx	51.437	57 (1)	45 (1)
SortedGreedy	55.453	59 (11)	50 (1)
MultipleSort	59.0055	60 (445)	56 (2)

This deduction is also supported when looking at largest and smallest Matching and their respective occurrences.
While the Networkx and Sorted Greedy algorithms have quite small occurrences of their largest and smallest amounts, it is visible, that the Multiple Sort has a high occurrence of the value 60, which is the largest possible value of matchings in a graph of 120 nodes.
This amount is so large, that the average matching of the Multiple Sort algorithm is higher than the largest Matching of the other 2 algorithms.

But this high Matching-output comes at a cost in runtime.

Time



As can be seen the Multiple Sort algorithm runs significantly longer than the Sorted Greedy and NetworkX algorithms. This is most likely because the Multiple Sort algorithm iterates through all neighbour nodes to remove the already matched nodes.

Algorithm	Average Runtime in ms	Longest Runtime in ms	Shortest Runtime in ms
Networkx	0.282811	1.054453	0.235588
SortedGreedy	0.463941	1.711578	0.388222
MultipleSort	2.596983	7.915431	2.143273

This can also be seen in the Average Runtimes and the longest and shortest runtimes. It shows that Networkx takes on average around 60% of the time of the Sorted Greedy algorithm and the Multiple Sort algorithm runs 9 times as long. Both the Networkx and SortedGreedy longest runtimes as still faster than the shortest runtime of the Multiple Sort. For all three algorithms, it can be observed that the average runtime is way closer to the shortest runtime compared to the distance between the average runtime and the longest runtime. This shows, that the worst-case complexity for all algorithms is much higher than the average complexity.

2.4)

The Sorted Greedy algorithm's worst-case complexity is in case every node has every other node as a neighbour.
The reading of all neighbour nodes iterates through all nodes and then through each of its neighbours, this leads to a complexity of $O(n^2)$.
The initial sorting of the list has a complexity of $O(n)$.
While iterating through this list of nodes, the algorithm will iterate through all its neighbours until it reaches one, that is not yet matched. While looking up if the current node or the neighbour node is in matched_nodes has an average complexity of O(1), the worst case complexity is O(n)
This inner loop will have in its worst case to run through n-1 nodes to match the nodes. This means the inner loop has a worst-case complexity of $O(n(n-1)) \Rightarrow O(n^2 - n)$.
This leaves the outer loop with a worst-case complexity of $O(n(n(n^2-n))) \Rightarrow O(n^4-n^3)$, but in practice, it will be much smaller, as a graph with all edges present is very rare and the lookup in lists has an average complexity of O(1).
List operations such as append have a complexity of O(1) and can therefore be ignored.
The total worst-case complexity, therefore, is about $O(n^4-n^3+n^2 + n)$.
Simplified this would be a worst-case complexity of $O(n^4)$ which is polynomial.

The Multiple Sort algorithm takes much longer to run, but it also runs in polynomial time.
The reading of all neighbours leaves us again with a complexity of $O(n^2)$ while the initial sorting has a complexity of $O(n)$.

The next loop has a complexity of O(n) in its worst-case scenario, which would be no connections at all, as for every connection 2 nodes are removed from the unmatched_nodes list. For the first inner loop, this condition stays the same, as we are iterating all still unmatched nodes, to remove the newly matched nodes, giving again a complexity of O(n). The second inner loop again iterates over all unmatched nodes but has a third level of loops, which iterates over all neighbour nodes. This means that its worst-case scenario would be a node with connections to all other nodes. This leaves the second inner loop with a worst-case complexity of $O(n^2)$.
Additionally in the loop, the neighbour node has to be removed from the list and the list gets sorted again, both list operations have a worst-case complexity of O(n).
Appending to a list in contrast can be ignored, as it has a complexity of O(1).
This means the total worst-case complexity would be $O(n^2 + n + n(3n+n^2)) \Rightarrow O(n^3 + 4n^2 + n)$.
Simplified it has a worst-case complexity of $O(n^3)$

So even though we see that the worst-case scenario of the Sorted Greedy algorithm is higher than the complexity of the Multiple Sort algorithm it runs consistently faster. This is most likely because the lookup of nodes in the unmatched_nodes list adds $O(n^2)$ while relying on all nodes having all possible edges and the nodes being placed at the end of the list.