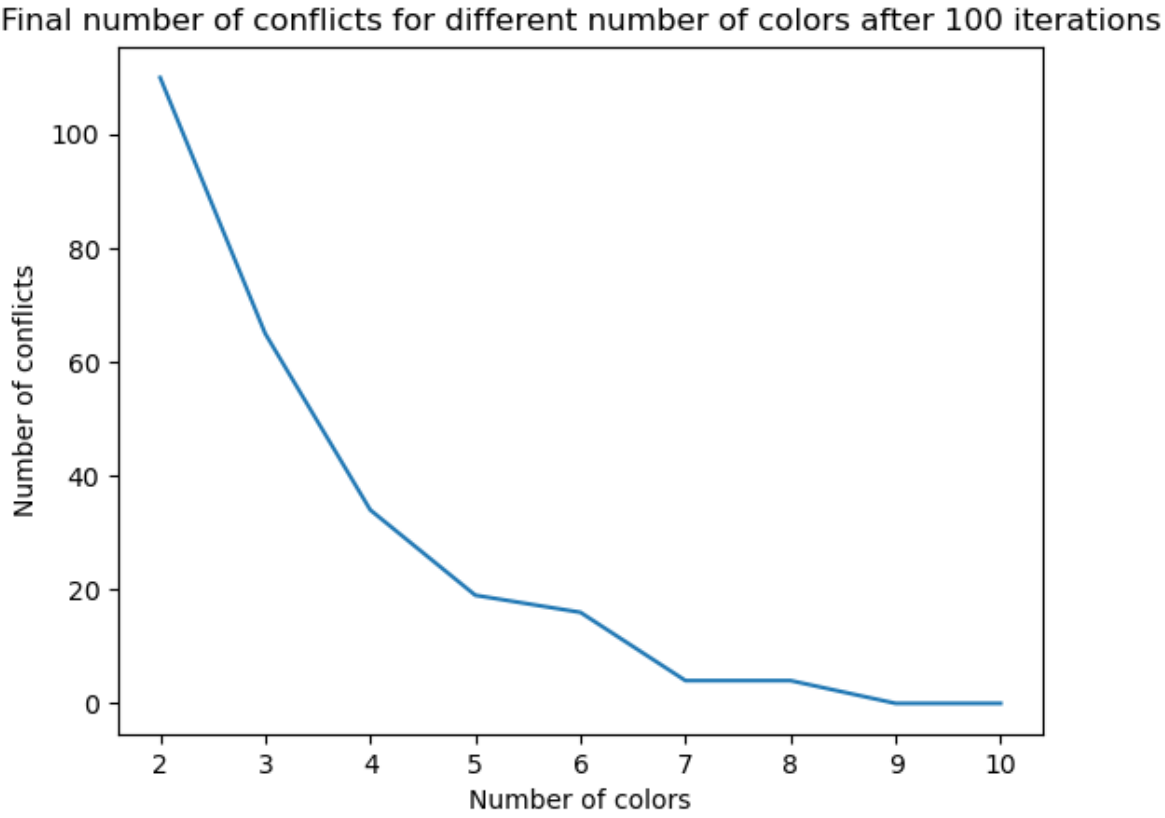


## Decentralized Coloring Algorithm

```python
def decentralized_coloring(graph, colors, num_colors, max_iter=100, hold=20):
    iterations = 0
    nodes = list(graph.nodes)
    num_conflicts = conflict_count(graph, colors)
    elite_conflicts = num_conflicts
    elite_colors = colors
    while num_conflicts > 0 and iterations < max_iter:
        iterations += 1
        random.shuffle(nodes)
        # for each node, try a new color and see if it reduces the number of conflicts
        for node in nodes:
            old_color = colors[node]
            new_color = random.randint(1, num_colors)
            colors[node] = new_color
            new_conflicts = conflict_count(graph, colors)
            # if the new color reduces the number of conflicts, keep it
            if new_conflicts < num_conflicts:
                num_conflicts = new_conflicts
            # otherwise, revert to the old color
            else:
                colors[node] = old_color
        print('Iteration:', iterations, 'Number of conflicts:', num_conflicts)
        # if the new coloring is better than the elite coloring, keep it
        if num_conflicts < elite_conflicts:
            elite_conflicts = num_conflicts
            elite_colors = colors
        # if we have reached the hold iteration, use the elite coloring
        if iterations % hold == 0:
            colors = elite_colors
            num_conflicts = elite_conflicts
    return elite_colors
```

## Explanation

Decentralized coloring is a graph coloring strategy in which, rather than using a central authority to color vertices based on some criteria, in decentralized coloring, vertices make local decisions about their own colors without any coordination with other vertices. In decentralized coloring, each vertex makes decisions independently, based on local information such as the colors of its neighbours, etc. Vertices iteratively adjust their colors based on local rules or algorithms, aiming to minimize the number of conflicts with neighbouring vertices.

Decentralization can lead to more scalable solutions, as the algorithm does not rely on a central authority to make coloring decisions for the entire graph. This can be particularly useful for large-scale graphs where centralized coordination becomes impractical. They are capable of adapting dynamically to changes in the graph structure or requirements. Vertices can adjust their colors in response to local changes, leading to more flexible and adaptive solutions. The are often more resilient to failures or disruptions. Since there is no single point of failure, the algorithm can continue to function even if some vertices or communication links are unavailable.

Decentralized graph coloring algorithms are also relevant in distributed computing environments, where nodes represent computing units or processes. Assigning colors to nodes in a distributed system can help manage resources, schedule tasks, or ensure mutual exclusion without relying on centralized control. They also have applications in communication networks, decentralized coloring can help minimize interference or contention among communication channels or frequency bands. Each node can adjust its channel assignment based on local information, leading to efficient and interference-free communication. Decentralized coloring can also be applied to resource allocation problems, where resources need to be allocated to tasks or entities in a distributed manner. Each entity can select its resources while considering the availability and compatibility of neighbouring resources.

## My Code

My algorithm starts by initializing the coloring of the graph, assigning a random color to each node. The number of conflicts in the initial coloring is computed using the `conflict_count` function. This count represents the total number of edges where both endpoints have the same color. The algorithm then performs a series of iterations, with each iteration attempting to improve the coloring by performing a local search (searching each neighbour). During each iteration, the algorithm randomly shuffles the order of nodes. This randomness helps in avoiding bias towards certain nodes. For each node, the algorithm considers changing its color to a randomly chosen color (different from its current color).

After changing the color of a node, the algorithm calculates the number of conflicts in the modified coloring. If changing the color of a node reduces the number of conflicts, the new color is kept. But if the new coloring doesn't improve the conflict count, the node reverts to its original color.

The algorithm maintains an elite coloring, which represents the best coloring found so far. If a new coloring produced in an iteration results in fewer conflicts than the current elite coloring, the elite coloring is updated to the new coloring. This mechanism helps the algorithm escape local optima by periodically adopting improved colorings. The iteration process continues until the maximum number of iterations is reached, or the number of conflicts is reduced to zero, i.e. a valid coloring has been found.

After a certain number of iterations (hold), the algorithm adopts the elite coloring. This step helps maintain diversity in the search space and prevent stagnation in local optima.

## References

Galán, S.F. Simple decentralized graph coloring. Comput Optim Appl 66, 163–185 (2017). https://doi.org/10.1007/s10589-016-9862-9 https://link.springer.com/article/10.1007/s10589-016-9862-9

Yves Boutellier – Towards Data Science, Graph Coloring with network https://towardsdatascience.com/graph-coloring-with-networkx-88c45f09b8f4