Heuristics for State-Space Search

1. Pre-Coloring and Vertex Removal
   1. **Coloring a clique**

By searching for a clique of a given size and coloring it arbitrarily we can reduce the number of states that need to be searched, as the number of child chares a chare would need to spawn would be reduced. We would consider a clique size of 3 since larger cliques can be formed from combining size-3 cliques.

* 1. **Removal of vertices with fewer neighbors than k, the number of colors**

Any vertex fewer than k neighbors is guaranteed to have a color available to it. These are removed at the starting chare and placed in a local stack for latter LIFO merging. Also, at every stage of the state space search, a local algorithm tries to find if the number of colors available to a vertex exceeds the sum of its uncolored vertices and different colorings to colored neighbors. If that is the case, that vertex is removed since it is guaranteed to have a color available to it, and is stored locally.

The total coloring could then be determined by traversing back up the state tree from a leaf that found the solution.

1. Detection of Independent Subgraphs

This is a simple heuristic would allow us to color smaller graphs independently. However, in the case where a graph is connected we can find an articulation point (a vertex that once removed would split the graph into one or more independent graphs), color that vertex, and then color the subgraphs. This heuristic can be applied recursively to obtain any number of independent graphs that can be colored and the composed. This kind of a splitting generates an AND-OR space search tree instead of just an OR tree.

The child chares that are coloring the subgraph maintain proxies to their parents. If a child finds a solution then it reports to its parents and the message continues up the tree. If a solution is not found then the child will report up to its parent and the important point is that the parent needs to recursively kill all of the child chares that it spawned. This is the important distinction about the AND-OR tree, namely that a failure at an AND node (where we split the graph at an articulation point) leads to failure for all child nodes of the AND node. Furthermore, to speed up killing child chares, all chares wishing to spawn a child must ask their parent whether or not they can spawn a child, thus reducing the potential depth a kill command might have to travel down to complete.

1. Impossibility Testing and Forced Moves
   1. Impossibility testing

If we find that by coloring a vertex that one of its neighbors is reduced to 0 possible colors then we don’t generate that state. This helps to reduce total chare count.

* 1. Forced Move

If by coloring something we find that one of its neighbors is now forced to be a certain color because it only has one color available then we perform that move. However, this might then lead to another vertex not having any possible colors left, which means that the forced move heuristic has to be recursive in nature.

1. Value Ordering and Prioritization

When a chare selects the most constraint vertex to color, and spawns off ‘m’ child chares, where ‘m’ is the number of possible colors for the vertex, then we assign bitvector priorities to the child chares. The chare in which the assigned color least impacts the color possibilities of the neighboring vertices is given the highest priority. In this way, we sweep the state space search in a left to right manner.

1. Independent Subgraphs (AND-OR trees)

This is a complex topic, but the computational details are as follows

1. Adaptive grainsize control for Chare creation

As we had discussed earlier when only a certain number of vertices remain to be colored we can just have a chare do the coloring sequentially instead of having to spawn children for each vertex coloring. However, apparently using the number of remaining vertices to be colored leads to less than optimal execution time.

A solution to this is to have a chare start evaluating the entire state-space tree below it in a sequential manner and to add each new space to a stack. When the stack fills up to a certain point then the chare fires off children to explore these spaces and empties the stack to start adding to it again. This might be a simple change to incorporate into our design by simply adding a stack push whenever we would fire off a new chare and popping when it fills up to a certain point.

**Grain Size control**: The number of vertices colored at each level of the state space search.

If each node selects 1 vertex (most constraint) and spawns K children (k=color possibilities for that vertex, max K=chromatic number), then the number of spawned chares grows as:

1 + k + k2 + k3 + k4 … kd (d = depth)

= O(Kd-1)

This is not an exact estimate since the branching factor would reduce (be much less than k as we go down the state space since the coloring possibilities reduce)

Let the grain size be G. This means that each node would select the G most constraint vertices and then spawn kG children. Total chares spawned in this case is:

1 + kG + k2G + k3G + …. kaG (aG = d since the number of leaf nodes remain the same)

= O(kaG - G) = O(kd-G)

Comparing the two, we get a reduction of a factor of kG-1 for a grain size of G.

A more aggressive way of controlling grain size, which would lead to a less optimal solution, but with much less number of chares: (we may not do this, but just for completeness):

* At each level of state space, a chare colors (permanent coloring) G vertices, then finds the next most constraint vertex, and spawns off K children. The permanent coloring of G vertices could be done using some sequential graph-coloring algorithm.
* The number of chares in this case grows as:

1 + k + k2 + k3 + …. km (m=d/G, d = original depth as in previous cases, G = as mentioned in above bullet)

= O (kd/G-1)

Here, we get a reduction of a factor of kd-d/G for a grain size of G. This is much more than the previous reduction of kG-1, and hence we would find a solution much faster, using much less memory. But the downside is that we **won’t get the most optimal coloring**.

1. Supersteps

Supersteps in principle involve performing more than just one coloring at a node in the state-space tree, and in some ways we are doing that with the forced-move heuristic. However, parts of our current set of heuristics don’t pair well with the idea of doing a lot of sequential work at each node in the state-space tree. This is an area I think that we should explore some more, but it will require some discussion into changing the design of our approach.

1. Other ideas
   1. Dividing the chares into arrays/sections/groups/nodegroups for efficient multicasts. What is a good strategy?
   2. Setting priorities in messages? Also, do we stick with marshaled parameters or switch to messages?