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. In the paper they use a clique of size 3 which is very quick to find, but we could potentially do larger cliques with a backtracking algorithm.

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

This is a simple heuristic that I think will definitely help a lot. Any vertex fewer than k neighbors is guaranteed to have a color available to it, and generally this will be the case.

However, the vertices that are removed will have to be added back in a LIFO manner. This makes it difficult to parallelize the addition of the vertices back into the graph as it seems to be sequential in nature (constantly pinging a central chare for the next vertex to add would both be slow and potentially cause race conditions). Coming up with a good parallel solution to this addition problem will most likely give us a huge speedup. However, I’m not entirely sure that an elegant solution to this exists.

* 1. Reapplication of the heuristic during the search

During the search we can go back and apply the previous heuristic and remove vertices if a vertex with U uncolored neighbors and C colors used to color those neighbors has a U+C <= k.

The main question with this is when do we decide to reapply this heuristic? If we can find a fast parallel solution to the re-addition of the vertices back into the graph then we might be able to use this heuristic after generating a new state by coloring a vertex. 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.

What is important to note is that this heuristic can be applied recursively to obtain any number of independent graphs that can be colored and the composed. Furthermore this heuristic can also be combined with the vertex removal heuristic during the actual search to potentially get great speedups. The one problem is that this kind of a splitting generates an AND-OR space search tree instead of just an OR tree, and dealing with such a tree in a parallel context is harder (see 5).

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. It’s pretty simple in nature should be incredibly easy to implement.

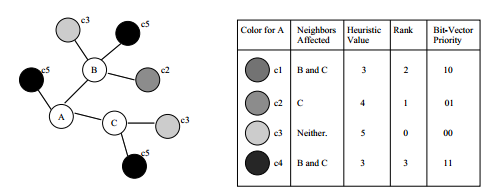
* 1. Forced Move

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.

The main question when it comes to this is how deep the recursion might get. It is possible that this kind of testing leads to an O(V) operation where V is the number of vertices. Therefore, I think it might be worth only performing this when the number of uncolored vertices is below some kind of computationally feasible threshold. Additionally we could do selective forced moves at every step by coloring a vertex v and then forcibly coloring its immediate neighbors only if these neighbors have neighbors that are all colored. By changing the depth (checking neighbor’s neighbor’s neighbors, etc…) to which we perform a search like this we might be able to find the computationally optimal point for a search like this.

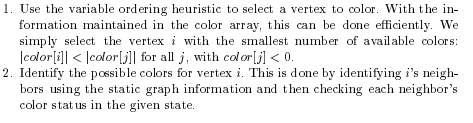
1. Variable and Value Ordering

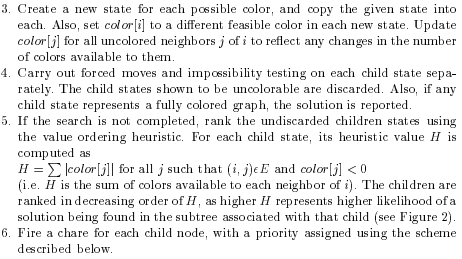
This heuristic is based on choosing a color for a particular vertex v by checking which colors are also not shared by the neighbors of the neighbors of v. Consult the below diagram for a clear picture.



This would generally be a fast thing to figure out as at most it would be an O(k^2) operation and k is not generally very large. Furthermore, doing this after the forced move heuristic would further reduce the number of potential color choices. This is a simple heuristic that I think we should implement soon.

**Overall Node algorithm using the above Heuristics**





Now, this algorithm is very close to what we might use. The changes that that we would make to this would potentially be in the pre-processing of the vertices (1, with particular emphasis on the parallelization of the LIFO re-addition problem) and limiting the recursive depth of forced move testing (2b, with emphasis on small sequential forced-move checking with constraints to prevent necessary recursion). I also think that we might be able to add some additional input information to the state-space child chares to allow for faster execution, but we’ll have to discuss that.

1. Independent Subgraphs (AND-OR trees)

This is a complex topic, but the computational details are as follows. If we choose to split the graph into independent subgraphs by removing articulation points then we need to have the child chares that are solving the coloring 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. 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. 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?