ACO Plan

Vibhasnhu

March 2016

1 Approach

As previously discussed the plan is to make a configuration graph of the original graph. Each node of the configuration graph will contain information about the position of obstacles and robots.
For a graph of N nodes with K(¡N) obstacles, the configuration graph will have $\binom{(N-1)}{K}$ nodes, which has an upper bound of $\binom{(N-1)}{(\frac{N-1}{2})}$. This upper bound will occur in the case of full connected graph (i.e. directed edge between any 2 edges in a graph) which is a very unlikely case. More realistic graph configurations have no. of edges E = O(V log V) intuitively. In the program given in graph500.org website, the random graphs generated have even lesser number of edges.

In this configuration graph if we are having an initial configuration C_H^v then to check if robot can reach all the other nodes u of original graph G if for each u $\exists S$ such that $|S| = |K|$ and there is a path from C_H^v to C_K^u in the configuration graph of G.
This problem is a varient of Steiner Tree problem which is extensively studied and there are some solutions which use Ant-Colony Optimization(ACO) meta-heuristic.

2 Current Implementation(Ongoing)

The serial implementation of the above approach is relatively straight-forward. But the only issue is storing the value of pheormone content in the configuration graph and low latency queries over these data structure.
Our implementation is a 3 LAYER approach:
The bottom layer is a map data structure optimized for performance. There is no ordering of nodes in the configuration graph and hence classical hash-table algorithms are unsuitable for the purpose. Therefore, each node of the configuration graph is encoded as an array of ints.
In this array, the first integer will denote the position of the robot in the original graph and rest of the ints will form a bit-set type data structure to store the

position of the obstacles. Therefore, each node of the graph G with N vertices will take $(\frac{N}{sizeof(int)*8} + 1)$ bytes of memory. Let us call this to be KEY for the node.
To store the pheromone content on the configuration graph, we need a data structure which can map a pair of KEYs (i.e. an edge in the configuration graph G) to a float point value which will denote the pheromone content of that edge. In MAX-MIN ACO, the pheromone content is always bounded both from above and below and hence edges having low pheromone content don't need to be stored in this datastructure. It has been observed in MAX-MIN ACO that after few(10) iterations , pheromone content of most of the edges lies very close to its minimum value and hence this data structure can be accordingly optimized by simply discarding such edges.
According to a very conservative estimate, for practical application, the data structure should be able to hold in the range 100 million keys at a time. Implementing this using C++ STL map structure is having atleast 200% overhead when we have graph G having ¡30 nodes when the number of keys get in the range of 100 million.
Above that we have a layer of Graph API over the configuration graph (which will have standard functions expected in a graph). In addition it will have function to retrieve and set the values of pheromone content in the graph.
The topmost layer is the Ant Colony Optimization (ACO) layer for finding the Steiner tree over the configuration graph.
This is the minimal implementation of our approach. The key points are:

- The bottom layer can be implemented on GPU with 0 overhead because in our case the key-value store problem is "embarrassingly parallel".
- It was observed that caching the queries (something like MO's Algorithm) and serving them in parallel dramatically improved the performance of the key-value store of the bottom layer. The C++ 11 standard provides the facility of future and promises which is to be used to avoid programming effort with caching.
- The current cached implementation of this key-value store gives superior performance.
- The middle for Graph API should be thread safe because the top layer will have multiple threads running who will be calling the Graph API operations concurrently. All the operations in this graph API should be lock free to avoid race conditions in the top layer. Infact it is possible to sacrifice accuracy for speed in functions querying the pheromone content.
- Anything more complicated then this organization of program will hit hard on performance even if theorectically it may convergenge in fewer iterations. The reason is similar to that of C10K problem in obsolete web-servers. For any step in the ACO(in top layer), if there will be more then 3 pointer redirections, there will be noticable lag. All the network

communications, context switch of threads, locks etc will be counted as a pointer redirection here.

- The bottleneck in the whole operation is the performance of this key-value store.
- There are a few implementations of this key-value store on GPU. I am still investigating the performance of those and checking if will be useful for our purpose and how to fit it in our implementation. I am looking for a very specific set of features in them and hence can do away with the general operations on those key-value stores.