Parallel Held-Karp Algorithm for the Hamiltonian Cycle Problem

Erik Burton
CME 323 Final Project
June 5th, 2016

Abstract:
In this project we will attempt to parallelize and optimize the classic Held-Karp algorithm for Hamiltonian cycles so that it runs in an efficient parallel fashion on PRAM arbitrary concurrent read, concurrent write (CRCW) machines. The goals are to have a version of Held-Karp which runs faster proportional to the number processors supplied, maintains the worst case bounds on memory and computational time for the problem while having a good expected run time on random graphs with a relatively small number of processors.
We are partially succesful, making an algorithm that given $\sqrt{n}$ processors runs in O(n) expected time for random graphs. It falls a bit short in slightly increasing the worst case bounds.

Introduction:

The Held-Karp Algorithm:

The Held-Karp dynamic programming algorithm is widely held to be the foundational algorithm for the Traveling Salesman Problem and the Hamiltonian Cycle sub-problem. As such it acts as the worst case fallback for many of the most successful randomized Hamiltonian cycle algorithms and in that capacity gives a firm upper bound on computational work on any given graph. The basic idea of the algorithm is simple. We trace a path stepwise, node by node, adding new nodes to our path if the new node is connected to the last node we added. If we end up in a situation where there is nowhere to go, we backtrack down the path we've come until a new alternative way forward presents itself. To prevent redoing work we've done before we store unsuccessful paths attempted in a way that can be quickly checked.

The basic recursion is as follows [i]:

We start with some node, say id 0, when there are n total nodes. We let S be the set of nodes we have yet to incorporate in our path and A is our adjacency matrix. Our recursion function is Cost(set S, previous_node_id k).
Base case:
if |S| == 1: Cost(S, S[0]) = 0 if A[S[0],0] == 1 and 1 otherwise.
Recursive case:
if |S| > 1: Cost(S, k) = min $_{m \text{ in } S-k}$ (Cost(S-k, m) + (if A[k,m] == 1: 0, otherwise 1)).

If we return with a cost of 0, we know we've found a Hamiltonian Cycle.

The upper bound on the time it takes for Held-Karp to run and report whether there is a Hamiltonian cycle (providing the cycle when there is one) is $O(n2^n)$ *[i]*. In the worst case we also store one bit of information for each possible partial path leading to a $O(n2^n)$ bits of storage. We'll go into more depth regarding the summation that gives this bound, when giving the proofs in the Performance section.

A small note: the performance given above is not what is given in Held-Karp's paper. Their solution is general for the Traveling Salesman Problem and is therefore inefficient for the simpler Hamiltonian Cycle problem. The above are the bounds for an optimized Held-Karp made for the purpose of fair comparison. The original runs in $O(n^2 2^n)$

Terminology:

Before proceeding it's useful to establish some notation. We will denote graphs as G = {V,E} where V is the set of nodes and E is the set of edges. Size(V) = n and Size(E) = m.

We will be discussing random graphs. They are, following the standard definition in the literature, graphs with some number of nodes where an edge has a random p in (0,1) chance of being in the graph. p in this case is a constant set during the creation of the graph and identical for all possible edges. The final piece of variable notation is that our PRAM machine has some number L of processors available to the algorithm.

With regards to the algorithm and proofs we consider the notion of a 'level'. Note that Held-Karp style algorithms add single nodes to paths as they build toward a cycle. If the algorithm has k nodes in it's path so far we say it is at level k. The intuition behind this is imagining a factorial tree of nodes connected by edges stretching down from your starter node. Each step down there are fewer possible paths as you cannot return to a node you've been to before. The goal of the algorithm is to find a path to the bottom where it will have completed a Hamiltonian cycle. In this analogy your level is the depth you've gone down the tree.

A related concept is 'step'. The algorithm takes a 'step' forward or backward when it changes level in the tree.

Literature Review:

Beside the aforementioned Held-Karp paper work in this area falls into two broad camps: Improvements to the worst case bound on finding a Hamiltonian cycle on any graph, the most recent and notable being Bjorklund[2010][ii] running in $O(1.657^n)$ time. The other camp are those who try to improve the expected completion time of the algorithm on various kinds of graphs. The most general kind is the random graph described in the Terminology section. The fastest such algorithm I could find for PRAM machines is MacKenzie-Stout[1993][iii], which incredibly runs in O(log*(n)) expected time and O(n) expected work with (n/log*(n)) processors on a CRCW machine. Their approach is completely different from what we are attempting in this project, but is worth mentioning for the sake of comparison.

Both of those papers create novel approaches to the Hamiltonian cycle problem in order to get around the drawbacks of the Held-Karp model. The MacKenzie-Stout algorithm however is probabilistic and in the case where it's main algorithm does not work, it will defer to Held-Karp to give a completion guarantee. The object of this project on the other hand is to improve

upon Held-Karp with parallelization, while maintaining all those desirable guarantees.

The Algorithm:

The parallel algorithm shares most of Held-Karp's basic structure, with three major changes. The first is that, instead of individual processors trying to find their way 'down the tree' processors work together in 'packs'. These packs, in the event that one of them finds a way to the next level will all move to the next level and begin searching from there. Intuitively this makes sense since adding a kth node to the path means there are n-k possible ways forward from this new node, that can be shared between more processors. The second major change is that the parallel algorithm unwraps the recursion. The recursion given in Held-Karp is elegant, but does not lend itself well to situations where many processors have to share information. As the parallel algorithm is iterative, starting at the *lowest level of the recursion and working up*. Thus at level k we already have n-k nodes in our path and need to place the last k. The third major change is a bundle of optimizations, meant to bring the time needed for each step down to O(1). Some of these optimizations also apply to Held-Karp and were added to their algorithm to make the comparison fair. As a small note, we use primes to get a hash number for sets. The set of node ids [v1,v2,...,vk] has hash primes[v1]*...*primes[vk] which is obviously unique to that set.

Variables and Storage:

Below is an overview, in pseudocode, of the variables used in the algorithm and their purpose:

Global variables:

```
# num_nodes:
n = 40
A = adjacency_matrix()

# Nested map. Maps set of connected nodes [hash_id] => current node =>   boolean of whether we have tried the
        paths ahead:
history_map = {}

# vector of first n primes:
primes = get_primes(n)

# start with random bottom node for whole model:
starting_node = random(0,n)
```

Pack global variables (accessible to all processors within pack):

```
num_p = 4 # number of processors available
remaining_nodes = range(0,n) # format [v1,v2,...,vk, 0,...,0]
path_history = [0]*n # format [0,...,0,n-k,...,n], we work bottom up
level = n-1 # level we are in n factorial tree (starting at n-1 [n, 0 indexed])
level_round = [0]*n # on level k we have checked the first level_round[k]*num_p  remaining
        nodes as potential next steps
hash_id = 1 # set of nodes so far hash to use with history_map
last_chosen_p = -1 # last processor to chose the way forward

pack = range(0, num_p) # note you can also run several smaller packs in parallel
```

Algorithm – iterative portion:

```
# Initializing pack:
# Start at a random node connected to our chosen top node.
path_history[level] = starting_node
hash_id *= primes[path_history[level]]
level -= 1

while (True):

        new_path_found = False

        parallel_for processor_id in pack:

                node = remaining_nodes[processor_id]

                if A[path_history[level], node] != 1: continue # no path forward, skip node

                node_hash = hash_id*primes[node] # hash id for adding this node

                # Check if we've tried the paths going forward previously
                # If we have and it didn't work skip potential repeat work
                if node_hash in history_map and node in history_map[node_hash]:
                        if history_map[node_hash][node] == 0: continue

                # Otherwise this node is a possible way forward:
                last_chosen_p = processor_id # CRCW to determine which processor's way
                                             forward is checked first

                if last_chosen_p != processor_id: continue # drop if this processor wasn't chosen

                # This processor's choice is done next:
                new_path_found = True
                hash_id = node_hash
                path_history[level] = node
                remaining_nodes[processor_id] = remaining_nodes[n-level]
                remaining_nodes[n-level] = 0
                level -= 1

        if new_path_found:
                if level != 0:
                        continue # run next cycle on higher level
                elif A[path_history[level+1,starting_node] == 1:
                        return path_history # Hamiltonian cycle has been found
                # If we are at level 0, but there is no final connection we need to backtrack
                # So we just let it hit the conditions below

        # Otherwise no next step found so far:
        level_round[level] += 1 # try the next num_p possible edges

        # If we've tried all the possible edges we need to backtrack:
        # Note, going up we have k possible edges at level k.
        if level_round[n]*num_p >= level:
                level += 1 # move one level down, removing last step
                remaining_nodes[n-level] = path_history[level+1] # add node back
                history_map[hash_id][path_history[level+1]] = 0 # note in map that the
                                                                  attempted path won't work
                hash_id /= primes[path_history[level+1]] # decrement hash_id


        # If there is no Hamiltonian cycle:
        # Happens if we backtrack and there is nowhere to go.
        if level == n-2 and level_round[n]*num_p >= level: return False
```

Proof of Correctness:

First let us bound the number of unique states the algorithm can be in. Functionally the states are characterized by the set of nodes which have not been incorporated into the path, and the last node that was added (our current node). Note that if we fix the set size as k there are (n choose k) * k different states the algorithm can be in as there are n choose k ways to select a set of size k, and each of those k can be the current node. So the total number of states is:

$$O(\sum_{i=1}^{n} i*\binom{n}{i})=O(n2^n)$$

Now consider whether our algorithm halts. Note that whenever it attempts to proceed from a state and does not successfully find a Hamiltonian cycle along any path it backtracks and marks that state as useless in history_map. It will not attempt to proceed from that spot again. Note also that we will exit and return that there is no Hamiltonian cycle when at the bottom level with no-where left to go (either no paths from bottom node, or all paths have been attempted unsuccessfully and marked in history_map). Thus after $O(n2^n)$ round of the while loop we must exit. Thus the algorithm always halts.

Now consider, if it halted and returned a path. In that case we have both reached level 0 and the final node we added connects to the first node in the path, meaning we have a cycle. Note that since we remove the node id of each node we add to the path from the list of remaining nodes, there are no repeated nodes in the path. Also note, that to add a node to the path it must be connected to the previous node. Thus path_history = v where for each i < n-1, v[i] is connected to v[i+1] and v[n-1] is connected to v[0]. Thus we have a Hamiltonian cycle.

Alternately, assume we return that there is no Hamiltonian cycle.
First consider backtracking. Let's prove that if we backtrack from a state there is no Hamiltonian cycle which can be made from that state. We proceed by induction:
Consider the state in which we have selected our path of n nodes. If we backtrack that means the last node was not connected to the first and no Hamiltonian cycle was possible.
Assume for induction that if we backtrack from having k+1 nodes then there was no way to complete the Hamiltonian cycle from there.
The if we are backtracking at a point where we have k nodes in our path that means that from the that kth node every remaining node not yet added to the path was either not connected to that kth node, or if we were to add that node we would get to a state of k+1 nodes that we had backtracked from, and thus could not lead to a cycle. Thus at this k node state there is no path forward to a cycle and we need to backtrack. By induction backtracking when we have k nodes in our path always means there is no way forward to a Hamiltonian cycle.

Finally, consider when we return that there is no Hamiltonian cycle. We are then at the lowest level (n-1) and find that there is no way to add a node which has an edge to our starter node, which will not lead us to a state we backtracked from. Thus there is not Hamiltonian cycle starting from our starter node. As starter nodes don't matter this means there are no Hamiltonian cycles in the graph.

Thus we must always complete, if we return a path it is Hamiltonian path, if we return there is no path, then there actually is no Hamiltonian cycle. Thus we return a Hamiltionian cycle iff one exists and return that there is no cycle iff there is no Hamiltonian cycle.

Performance:

A small note first. Our algorithm requires the first n primes to be available in a list. We can get the all the primes less than some number k in O(k loglog(k)) work and O(loglog(k)) depth[iv]. Since we want the first n primes we can make use of the asymptotic law of distribution of prime numbers[v] which holds that the nth prime is approximately nlog(n) asymptotically. Thus we can get the first n primes in about O(nlog(n) loglog(nlog(n))) work and O(loglog(nlog(n))) depth. Note that we only need to do this once for many graphs if we make the list long enough and that prime lists are readily available online, but it is worth including this cost.

Now, regarding the work being done. Consider each iteration of the while loop. In that while loop O(1) work is being done outside the for loop. Within the for loop each of the processors available to the pack also does O(1) work. Thus the time of one iteration is O(1) and the work is O(|S|) where S is the set of processors available at that step. Obviously this is bounded by L, the number of processors. Now the question is, how many iterations of the while loop have to happen to completion.

The Expected Case:

We'll find the expected completion time on a random graph where each edge has probability p of being present and where a Hamiltonian cycle is present in the graph. If we consider this p fixed, as is the norm since it's a stand-in for relative density of the graph, we will find that the expected completion time given O($\sqrt{n}$) processors is O(n).

We will upperbound the time needed for completion by finding the probability of finding the Hamiltonian cycle in precisely n steps. We will show that this probability is independent of n for fixed p, and thus the expected number of times we would need to restart the algorithm to find the cycle in O(n) is constant. Note that since our algorithm does not restart each time it is more effective than this, since it has a good chance of only having to backtrack a few steps before finding the path forward, rather than restarting completely.

On level k, working from the bottom up, there are k nodes left which are not in our path. Each of these k has a p chance of being connected to our last node. So the probability that at least one is connected is $1 - (1-p)^k$. For simplicity, let (1-p) = 1/c; vitally c > 1. Then the probability is $1-c^{-k}$. Now, assume we only have f(k) <= k processors available at level k. Then the probability that we will immediately find the a next forward is $1 - c^{-f(k)}$. Let f(k) = $\sqrt{k}$. This won't work with lower values.
Now, the chance of completing each level in the first round one after the other with the f(k) processors is:

$$\prod_{i=1}^{n-1} 1-c^{-\sqrt{(i)}} = \exp\left(\sum_{i=1}^{n-1} \ln\left(1-c^{\sqrt{(i)}}\right)\right) \leq \exp\left(\sum_{i=1}^{inf} \ln\left(1-c^{\sqrt{(i)}}\right)\right)$$

Note that this converges for any n when the inner sum converges.
We use the ratio test to check for convergence of the series:

$$\lim_{n\to inf}\left|\frac{\ln\left(1-c^{\sqrt{(n+1)}}\right)}{\ln\left(1-c^{\sqrt{(n)}}\right)}\right| = \lim_{n\to inf}\left|\frac{\sqrt{(n)}\left(c^{\sqrt{(n)}}-1\right)}{\sqrt{(n+1)}\left(c^{\sqrt{(n+1)}}-1\right)}\right| = \lim_{n\to inf}\left|\frac{\sqrt{(n)}}{\sqrt{(n+1)}}\right| * \lim_{n\to inf}\left|\frac{c^{\sqrt{(n)}}-1}{c^{\sqrt{(n+1)}}-1}\right| = \frac{1*1}{\sqrt{(c)}} < 1$$

The last inequality follows from that if c > 1 then $\sqrt{c} > 1$. Thus $1/\sqrt{c} < 1$.
Thus by the ratio test our series converges to some constant C.

Thus our probability of successfully finding the Hamiltonian cycle in O(n) is exp(C), a constant independent of n. Thus we can repeat our O(n) steps an expected 1/exp(C) number of times to successfully find our cycle.

So the expected time to find a cycle in a random graph of size n is O(n). Note that we are using $\sqrt{k}$ processors at level k. So we need $\sqrt{n}$ processors total to make this work. Assume we have L > $\sqrt{n}$ processors. We can upper bound the amount of work we are doing on each step as O(L). Thus we are doing O(Ln) work in expectation. Finally, note that we are only going to O(n) states, so our total storage is also O(n) (including arrays and history, but not the adjacency matrix which is $O(n^2)$).

The Worst Case:

Let's consider the absolute worst case. In this instance we visit every possible state. We've already established that there are $O(n2^n)$ different states. Note that our while loop cannot enter a state it's already backtracked from. We also can only save each state once so, like Held-Karp, our upper bound on the data we store is also $O(n2^n)$. So what is the most work our algorithm can do in each state? Well, if we have to backtrack from every state, and we have the misfortunate of having to backtrack from all posssible neighbors each processor is doing O(1) work every time we backtrack. As we're returning to a state we randomly picked a forward path from before we may be repeating O(L) work. So the amount of work we are doing can be as high as $O(Ln2^n)$. Note though, that since the time on each iteration is O(1) and we, just like the original Held-Karp traversing every state, in $O(n2^n)$.

Evaluation:

Optimality and Shortcomings:

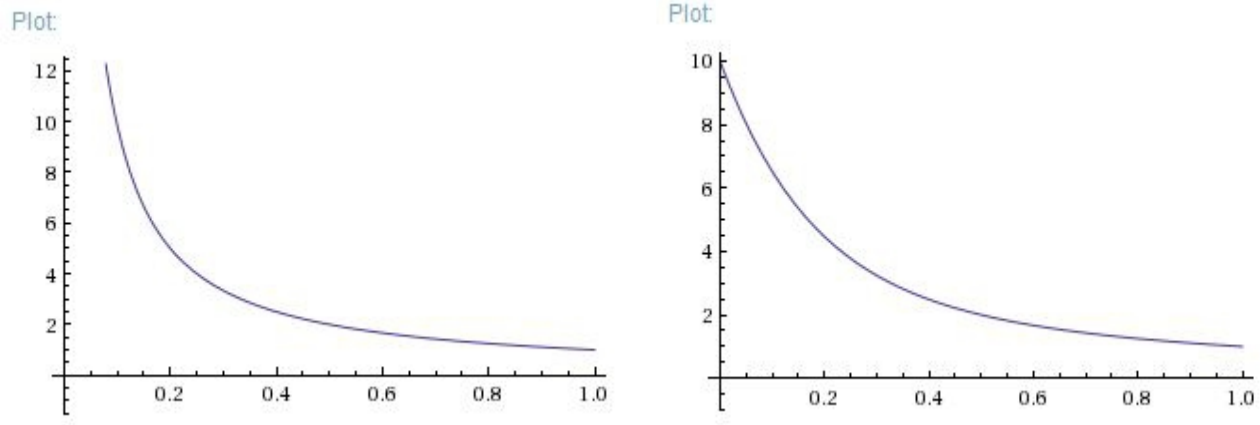
While not optimal in general, as MacKenzie-Stout O(log*(n)) expected time attests to, within the paradigm of adding one node at a time, both the expected time and expected storage of O(n) are optimal. Similarly, while not optimal in general, if we take Ω(n) steps with L processors working O(Ln) work is optimal. We'll explore below how the number of processors impact the completion time more generally, but given $\sqrt{n}$ processors and within the paradigm of adding one node, as started by Held-Karp, the algorithm acts optimally.

Unfortunately, the worst case bounds are not optimal, and in fact worse than the serial solution in terms of work. The reason for this is that in the worst case we backtrack from every possible state, and on each backtrack we redo min(L,k) work where L is the number of processors and k is the level we are on. The reason for this is that when we backtrack, our L processors reinspect the first L possible ways forward. While they have the advantage of finding a way forward among the first L possibilities in O(1) if such a path exists, running these simultaneously in O(1) also necessitates not being able to save your work in a way that can be read faster than O(1). Thus we end up redoing work. We can mitigate this in practice, but swapping ways that don't work in the array for new ones every time (not implemented in code for simplicity), but asymptotically it won't make a difference. Note that, although we are doing more work , we still complete in $O(n2^n)$ time. With regards to the next section it is worth noting that the worst case is a very dense graph, with many edges, but with only one Hamiltonian cycles to find. As we will see in a moment, denser graphs do not give us a

worthwhile speedup when adding processors.

The acceleration for sparse graphs:

Let us get an idea of the speedup we can expect on a random graph for some number of processors. Let p again be the probability of an edge being in the graph. Then, for the serial algorithm we would expect to spend 1/p time on any level to find the next possible way forward. Given L processors that work in parallel this is $1/(1-(1-p)^L)$. So we can get a rough idea of the speedup of L processors by finding the ratio of time we need to spend on each level: $(1-(1-p)^L)/p$. Note first that as our graph gets sparser (p -> 0) this value goes to m and that for p = 1 this value is 1. So for a perfectly dense graph we save no time, while for a very sparse graph our savings approach the number of processors we use. This makes sense, as for a very dense graph nearly any path you pick will be adequate, while for a very sparse graph, if there are k edges that do not exist, and appear consecutively (in our list of possible edges) it will take O(k) time to work through those for our serial algorithm, but only O(k/L) time for the parallel one.



The two graphs above are both speedup (y-axis) and probability p (x-axis). On the left for L=40 and for L=10 on the right.

It is easy to prove that the algorithm will always work through a level faster. Assume for contradiction that it doesn't. Then $(1-(1-p)^L)/p < 1$ which implies $p + (1-p)^L < p + (1-p)$. This is impossibly by 1-p < 1 and m >= 1. Thus the lower bound for our speedup is 1. Unfortunately, as seen above, the speedup will be less than linear.

Conclusion:

Future Work:
There are two main avenues of future work I would have liked to have given more time. The first is finding an accurate lower bound on the expected work for the Held-Karp algorithm. The calculation involves a tricky recurrence dependent on p and n; it is in a way a Geometric series nested in a Markov Chain. I strongly suspect that for p < 1/2 the time is bounded by $\Omega(n^2)$, but the proof I had to that effect was flawed and I was unable to resolve the flaw in time.

The second avenue is getting accurate time estimates for my own algorithm and Held-Karp in random graphs where the Hamiltonian cycle *does not exist.* This is a little tricker since we would need to estimate when we can expect to have to backtrack and where over all the possible paths in the graph, or something similar. This is somewhat of a missing half in evaluating my algorithm. It won't be worse than the worst case bound of course, and the

above estimations of speedup with more processors still applies, so it would certainly be faster or at least as fast as the serial algorithm.

Closing Remarks:

As discussed above the algorithm brushes up against the optimal efficiency within the paradigm of adding one node at a time. Unfortunately, this is no where near the optimal efficiency as discussed in the literature review. While we were successful in bringing down the depth by up to a factor of L, the number of processors, from the work on sparse graphs, we are way off the optimal O(1) depth and also not at the optimal linear speedup for adding processors. So in sweeping terms, we are fairly successful within the Held-Karp paradigm and successfully parallized the original algorithm, but the paradigm itself is weak for this task compared to other approaches. One upside though is that while MacKenzie-Stout need $O(n/\log^*(n)) \sim O(n)$ processors to get to expected time of $O(\log^*(n))$, this algorithm can do a respectable O(n) expected time with a mere $\sqrt{n}$ processors. While not a small number by any means it is almost within the reach even for very large (from a single machine's perspective) graphs of 1000 nodes. Given that all the operations are very simple arithmetic and reference only a handful of O(n) arrays along with the adjacency matrix, I suspect the algorithm could do fairly well if implemented on a GPU, even for massive single machine calculations.

i 'A dynamic programming approach to sequencing problems', Michael Held and Richard M. Karp, *Journal for the Society for Industrial and Applied Mathematics* 1:10. 1962
ii 'Determinant Sums for Undirected Hamiltonicity', Andreas Bjorklund, https://arxiv.org/abs/1008.0541, 2010
iii 'Optimal Parallel Construction of Hamiltonian Cycles and Spanning Trees in Random Graphs', Philip MacKenzie and Quentin Stout, https://web.eecs.umich.edu/~qstout/pap/SPAA93ultra.pdf, 1993.
iv 'Programming Parallel Algorithms', Guy Blelloch, https://www.cs.cmu.edu/~scandal/cacm/node8.html, 1996
v https://en.wikipedia.org/wiki/Prime_number_theorem