Below are results for the distance-*.txt files included with my Program B submission; every distance file contains only the first * cities from the distances-31.txt file, which is sourced from the data found at http://www.mapcrow.info/united_states.html.

Number of Cities	Runtime in Milliseconds	% Increase
5	6.8ms	
6	10.3ms	51%
7	16.1ms	56%
8	23.6ms	46%
9	34.4ms	46%
10	45ms	30%
15	7.9×10^2ms (790ms)	77% (average)
20	5.2×10^4ms (52,000ms, or 52s)	230% (average)

The runtime itself was measured using System.nanoTime(); for all results of n=15 and under, I ran the program around a dozen times, recording the approximate average of the times run (typically only recording 2 or 3 significant figures).

In order to minimise noise, the test was run with output forwarded to /dev/null (as can be seen in the sourcecode). The program was run on the first virtual console (though Xorg remained running on the seventh). It was launched with “chrt -f 99” (that is, with the highest priority FIFO scheduler, such that it should have preempted all other tasks) and no Java-specific runtime options.

So let's reason this out. Right off the bat, the theoretical (as in, I read it on Wikipedia) running time of dynamic Held Karp is $O(2^n n^2)$[1]. If true, this implies that for every nth step, the running time should double (and then a little bit more -- n^2 -- though this becomes marginal compared to 2^n as the input size increases).

In our actual execution, however, we beat this fairly handily; where 100%+ increases should be expected, we increase only be around 50%. The exception is as we get to 15 and 20, where the increases suddenly explode, arguably increasing faster than even the worst case should be; without actually investigating the cause of this, I believe it can likely be attributed to hashmap resizes that are occurring, in part because my code doesn’t prune them itself.

[1] Organic analysis of dynamic programming problems is stupidly hard, at least for me, since I’m unsure how to amortize the dynamically-solved subproblems. Supposing we just ignored those, my program has four loops: the first two run for the entire permutation of the input size, so O(n!). The next two both run O(n). So the total of the four loops is $(n^2 n!)$. I assume that by caching subproblems, we turn that O(n!) into $O(2^n)$, I just don’t know how.