Ian Schwer
Shaun McThomas CS 165: Lempel Ziv compression June 1, 2016

Building:

We took advantage of the GNU Make tool chain for our project. Running "make" or "make –f Makefile" in the project directory will build two targets: LZ and EXPAND. Alternatively, you could run make LZ and make EXPAND separately.

Data Structures:

We decided too implement this using a hash table, specifically std::unordered_map. The reason we choose too use this data structure was because we could have constant find and insertion times, and those were the only two operations we ended up needing to do our compression. The major disadvantage of this data structure was the space and in worst case we have linear lookup size equal to the container size.

The approach we took was too convert the input from a character array too a string of bits. The advantage of this was we could string operations such as substring and string length with still maintaining constant accesses too the string. For each character in the current window, then for some look ahead amount descending too 2 bytes, we attempt too find a matching string in the map. If we do find it, then we add a character double. If we don't find a match, then we insert the current string into the map, including all possible permutations of the string.

Before we write the tuplets out to stdout, we also combined character references that are next too each other into one tuplet, increasing the length. These combination tuplets we limit too be of size $2^S - 1$. After we have done this consolidation step, we iterate over all the tuplets, and encode them properly according too the instructions.

Decode was a rather simple implementation. We first go over the input string and build recover out tuplets, then for each tuplet, we either append the new addition or we do a repetition of existing characters based off the information from the tuplet.

Analysis:

Compress and decompress have two different complexities. Compression first has too iterate over each window, W. In the worst case, over each window we end up doing the entire look ahead amount, meaning that we always find unique bit sequences. This would end up having a complexity of $8wF$ time. However, we find that in the average case we have roughly had 50% repetition that appear exponentially distributed. This means that we get out of the method with $2wF$ time with a hash table of $(\mathrm{F}-2)^2 * \mathrm{w}$ space.

Decompression will have a better case when we have just have new strings since string addition is roughly linear up too the length of the total content. In that case, we have Nw complexity. In the case where there are mostly string references, we have to do an additional iteration of length offset. Offset in max can be F, arriving at NFw time.

Optimal Parameters:

Ian Schwer
Shaun McThomas CS 165: Lempel Ziv compression June 1, 2016

Too find the optimal parameters for LZ, we simply wrote a shell script too iterate over all the possible parameters, ran compression, and sorted by the compression savings output. This lead too N=11, S=3 and L=3. These parameters work because we are able to find the perfect balance between character literal matches and reference tuplets. Each window will be 2048 bits. From that window, we can make a string literal of 7 bytes, or 56 bits, which is roughly 3 percent of the total window size. If we found roughly half of the windows were new characters, then we would have roughly 18 new character tuplets, which gives half of them as repeats. After multiple runs we see that these are roughly at 50% repeats. Below is our table generated for the book1 file and for the xls sheet.

Book1

N	S	L	%	time
9	3	1	8.357	0m4.238s
9	3	2	7.941	0m4.132s
9	3	3	13.03	0m4.158s
9	3	4	14.63	0m4.127s
9	3	5	14.58	0m4.112s
9	4	1	16.97	0m10.265s
9	4	2	4.107	0m10.220s
9	4	3	10.53	0m10.278s
9	4	4	12.59	0m10.280s
9	4	5	12.68	0m10.867s
10	3	1	3.365	0m3.586s
10	3	2	15.85	0m3.481s
10	3	3	19.39	0m3.448s
10	3	4	20.03	0m3.399s
10	3	5	19.44	0m3.508s
10	4	1	3.916	0m9.689s
10	4	2	12.27	0m9.267s
10	4	3	16.82	0m9.418s
10	4	4	17.78	0m9.500s
10	4	5	17.27	0m9.589s
11	3	1	13.83	0m3.020s
11	3	2	22.85	0m1.831s
11	3	3	25.04	0m2.868s
11	3	4	24.99	0m2.928s
11	3	5	24.12	0m2.858s
11	4	1	7.828	0m8.499s
11	4	2	19.59	0m8.341s
11	4	3	22.49	0m8.698s
11	4	4	22.63	0m8.629s

11	4	5	21.82	0m8.726s
12	3	1	22.75	0m2.706s
12	3	2	28.84	0m2.580s
12	3	3	29.95	0m2.545s
12	3	4	29.48	0m2.483s
12	3	5	28.55	0m2.467s
12	4	1	17.94	0m7.377s
12	4	2	25.95	0m7.279s
12	4	3	27.53	0m7.262s
12	4	4	27.18	0m7.548s
12	4	5	26.28	0m7.554s
13	3	1	29.72	0m2.317s
13	3	2	33.61	0m2.295s
13	3	3	34.02	0m2.322s
13	3	4	33.41	0m2.319s
13	3	5	32.55	0m2.294s
13	4	1	26.04	0m6.768s
13	4	2	31.2	0m6.722s
13	4	3	31.91	0m6.519s
13	4	4	31.37	0m6.514s
13	4	5	30.54	0m6.445s
14	3	1	34.99	0m2.048s
14	3	2	37.35	0m2.023s
14	3	3	37.37	0m1.996s
14	3	4	36.79	0m2.010s
14	3	5	36.08	0m1.987s
14	4	1	32.38	0m6.155s
14	4	2	35.56	0m6.077s
14	4	3	35.76	0m6.153s
14	4	4	35.22	0m5.971s
14	4	5	34.53	0m5.889s

Kennedy.xls

9	3	1	58.87	0m2.098s
9	3	2	59.04	0m2.060s
9	3	3	58.34	0m2.073s
9	3	4	57.41	0m2.269s
9	3	5	56.33	0m2.076s
9	4	1	65.1	0m4.020s
9	4	2	65.62	0m4.038s
9	4	3	65.01	0m4.123s
9	4	4	64.1	0m4.138s

9	4	5	63.03	0m4.083s
10	3	1	59.06	0m1.808s
10	3	2	58.81	0m1.829s
10	3	3	57.97	0m1.857s
10	3	4	56.98	0m1.813s
10	3	5	55.92	0m1.780s
10	4	1	66.58	0m3.617s
10	4	2	66.64	0m3.556s
10	4	3	65.84	0m3.530s
10	4	4	64.87	0m3.614s
10	4	5	63.81	0m3.513s
11	3	1	58.4	0m1.595s
11	3	2	57.93	0m1.638s
11	3	3	57.03	0m1.611s
11	3	4	56.02	0m1.606s
11	3	5	54.98	0m1.610s
11	4	1	67.16	0m3.340s
11	4	2		0m3.262s
11	4	3	66.13	0m3.279s
11	4	4	65.14	0m3.286s
11	4	5	64.1	0m3.280s
12	3	1	59.69	0m1.422s
12	3	2	59.31	0m1.447s
12	3	3	58.58	0m1.506s
12	3	4	57.79	0m1.517s
12	3	5	56.97	0m1.513s
12	4	1	64.82	0m3.630s
12	4	2	65.45	0m3.616s
12	4	3	64.58	0m3.641s
12	4	4	63.63	0m3.631s
12	4	5	62.66	0m3.622s
13	3	1	61.91	0m1.244s
13	3	2	61.72	0m1.175s
13	3	3	61.26	0m1.180s
13	3	4	60.76	0m1.213s
13	3	5	60.25	0m1.184s
13	4	1	63.63	0m3.663s
13	4	2	64.52	0m3.570s
13	4	3	63.67	0m3.613s
13	4	4	62.78	0m3.638s
13	4	5	61.88	0m3.632s
14	3	1	62.65	0m1.052s

14	3	2	62.55	0m1.048s
14	3	3	62.26	0m1.039s
14	3	4	61.95	0m1.049s
14	3	5	61.63	0m1.057s
14	4	1	64.91	0m3.550s
14	4	2	65.05	0m3.732s
14	4	3	64.19	0m3.637s
14	4	4	63.31	0m3.653s
14	4	5	62.42	0m3.668s