Team Dino: Truly Johnson, Andy Ko, Daven Liu, Xiaoning Wang
CSCI 350
Project Report and Experimental Evaluation

Project Report:

Survey of background reading

We started by trying out the strategies of the four baselines. The background reading we did was vital in helping us code all four baselines, which we then built our own strategies upon to create a successful solver. Full citations for all readings are at the end of this paper in the Works Cited section on page 17. For the first three baselines, we had two strategies to exhaustively enumerate all possibilities. The first strategy is to simulate addition. We got this from Marie desJardins and Tim Oates from the University of Maryland (desJardins and Oates 2011). In this strategy, we imagined that we are doing a N digits addition in N decimals (where N is the number of colors. 1≤ N ≤ 26). Add “one" to the first color, "carrying" to the next column when the sum is becoming N+1. If the last index is already at the highest color, then there are no more guesses and guess is set to nil. The second strategy is to generate every possible permutation. This can only be used if you already know the color distribution - which colors and how many instances of each are present in the secret code. To do this, we made use of Donald E. Knuth's Algorithm L (Knuth, 2004), which starts with the colors in a defined order, than swaps indices systematically to generate every permutation until the colors are in the reverse of the original order.

We used desJardins and Oates’ recursive solution (desJardins and Oates 2011) in Baseline 1 and Baseline 2 and used Knuth’s non-recursive solution (Knuth 2004) in Baseline 3. In Baseline 1 we just exhaustively enumerated all possibilities. Baseline 2’s strategy is as same as baseline 1 except eliminating the guesses which got score of (0 0). Baseline 3 guessed the color distribution by first guessing a guess with only one color for each color. Once the algorithm found the color distribution it used Knuth’s algorithm to generate all possible permutations of that distribution. We developed it to a general solver using local search which will be specified in the next section. Finally, we implemented T. Mahadeva Rao’s algorithm for Baseline 4 (Rao 1982) which will be also be further explained in the next section.

Discussion of strategy

The general strategy for our solver is to use the parameters of the tournament to select a specific function tailored to those parameters. Many of the SCSAs have a specific function tailored to them, using the knowledge of what that SCSA does. Then there is a general solver for all other cases. Initially, our plan was to have one general solver using local search. We created this solver to perform significantly better than baseline 3, which it did. However, once we were able to create a working baseline 4, we realized it performed better than the local search solver for large problems. Thus, we decided to use baseline-4 instead of dino-local-search to solve any SCSA we didn't have a specific function for.

Our final solver uses the following functions: dino-two-color-alternating, dino-first-and-last, and baseline-4-Dino. dino-two-color-alternating uses the strategy of baseline 3 to figure out which two colors are in the code, then generates both guesses where those two colors are alternating. dino-first-and-last uses the strategy of baseline 4 but if the first position is fixed to a certain color, it fixes the last position to that same color, and vice versa.

Our general solver, baseline-4-Dino, constructs guesses according to Rao's algorithm (Rao 1982) with a few tweaks. Rao's algorithm uses the terms 'being-fixed' to denote a color which is known to be present and whose position is being determined, and 'being-considered' to denote the next color in the increasing order of colors. Baseline 4 begins with constructing a guess with the being-considered color and will learn from the response that it will get from that guess and store it in a knowledge base called 'inferences' from Rao's algorithm which is a list that contains several sublists, each with the correct color as the header and a list of the possible positions the color may belong to. In order to determine if the being-considered color is correct and the number of times it appears, we find the gain which is to subtract the number of colors in the guess that has a position in inferences (this is different from Rao's algorithm, but the result is the same) from the sum of the correctly placed pegs and the incorrectly placed pegs. If the gain is less than or equal to 0, then the color does not belong in the code. If the gain is greater than 0, then we will add a sublist, which will contain the color as the header with a list of possible positions, (equal in number to gain) to inferences.

To determine if the being-fixed color is in the correct position, we will use the incorrectly placed pegs. If there are 0 incorrectly placed pegs, then that means the being-fixed color is in the correct position and we will fix the being-fixed color to that position and we will update being-fixed to the color in the next sublist of inferences. If there are 1 incorrectly placed pegs, then that means both being-fixed and being-considered color are not in the position of being-fixed and we would remove that position from all sublist that has either colors in inferences. If there are 2 incorrectly placed pegs, then that means being-fixed is not in the correct color, but being-considered is supposed to be in the position of Being-Fixed so we would fix the position of being-considered color to the position of being-fixed. With Rao’s algorithm, it is not possible to have more than 2 incorrectly placed pegs. After each guess, it would update being-considered to the next color in the colors list and clean up the inferences list, which will remove any fixed positions from all other unfixed color sublists. After updating the inferences list, we build the next guess. The next guess will use inferences to see whether any position is tied to a color, in which it set that position to the tied color. Then it will set the being-fixed color to it’s next position. If we find all the colors, which means the length of inference is equal to the number of positions, then we set any position that does not have a color to the second color which is not yet fixed in the Inferences list, otherwise set any position that does not have a color to the being- considered color (Rao 1982).

In addition to the functions used in the final version of the solver, our code also contains the other three baselines and a section at the end for functions we developed but decided not to use in the final product because they were outperformed by other functions. These include dino-ab-color, dino-only-once, dino-mystery-2, and dino-localsearch. Dino-ab-color was based off of using baseline 3 with only A and B as colors, but was later outperformed on ab-color by simply passing the list (A B) as colors to baseline 4. dino-only-once attempted to add the knowledge that each color was only used once to baseline 4, but unmodified baseline 4 still outperformed it. Dino-mystery-2 was designed to generate codes with three colors alternating, as we believed this was the pattern for mystery-2. But baseline 4 outperformed it on mystery-2. And, as mentioned, dino-local-search was supposed to be a general solver for all SCSAs we had no specific code for. But baseline 4 outperformed it too. Still, it is worth describing the strategy used for it, as it could potentially be modified and improved in the future.

The local search function, dino-local-search, was initially supposed to be a modified local beam search, but it turned out that it was quicker as an ordinary local search (i.e. with beam width set to 1) so in this paper we will just look at the case where beam width = 1, because that gave the local search its best performance. dino-local-search constructs guesses using a similar method to baseline 3. First, it constructs guesses of only one color, to figure out how many pegs of each color there are in the code. Once it figures out the color distribution of the code, it generates an initial guess- the colors in order, and guesses it. This is just like baseline 3. But after this, instead of simply generating every possible permutation as in baseline 3, dino-local-search makes a random swap between two different elements to generate a guess that has not been tried before and guesses it. Then, the solver compares the result of this guess to that of the previous guess. With this information, the solver can see what effect the swap had, and learn some information about the secret code. If the swap led to an increase of two correctly placed pegs, that means the two pegs that were swapped are now in the correct positions. Conversely, if the swap led to an decrease of two correctly placed pegs, that means the two pegs that were swapped were in the correct positions before being swapped. If the swap led to an increase of one correctly placed peg, that means the two pegs that were swapped may not both be in the correct positions now, but they were certainly both in the wrong position before. And if the swap led to an decrease of one correctly placed peg, the swapped pegs are certainly in the wrong position now.[1]

Based on the difference in correctly placed pegs before the swap and after the swap, information is added to the list of known-pegs or the list of wrong-pegs for each position. Then, a new swap is made, randomly, using all the known-pegs and avoiding the wrong-pegs (except 1/50 of the time, when a suboptimal guess can be made. This prevents the algorithm from getting stuck). The algorithm continues like this until the correct guess is made.

Theoretical Analysis:
First we look at a simple case- two color alternating. The worst case number of guesses for this one is (M-1) + 2 = M+1. The color distribution can always be guessed with M-1 guesses (not M, because once you have the number of each of M-1 colors, you can figure out the number of the last color using subtraction), and then there are two additional guesses to guess both ways the two colors could alternate. To generate a guess takes at most O(N) time - to loop through a list of N elements and set them to alternating colors. But the other algorithms are much more complicated.

Baseline 4
Baseline 4 is the heart of our code which runs on SCSAs we have no specific functions for and is also the basis for some of our other SCSA functions including dino-first-and-last. Because of this, in most practical cases, the complexity of our code will be equivalent to the complexity of Baseline 4. Baseline 4 has two main functions, the Update function which updates the inferences and the Getnext function which builds the guess using inferences. For N pegs and M colors, generating the first guess is O(N). The Getnext function assigns each position of guess a color, which is N times. If we denote L for the length of inferences then finding if a position is tied to a color (computational complexity of O(L)) and finding the tied color for that position

[1] If the swap led to an increase of one correctly placed peg, one peg (P1) must have been in the wrong position before, but is now in the correct one, causing the +1. The other peg (P2) must either have been correct before and still correct, or wrong before and still wrong. Either way, the position P2 used to occupy is now occupied by P1, so P1 must be correct in the position that P2 occupied before the swap. If P2 was correct before the swap and still is, that would mean both P1 and P2 are correct in the same position, and are thus the same color. But the algorithm never swaps two pegs of the same color. This means that both pegs must have been in the wrong position before the swap. A symmetric proof can be used to show that a decrease of 1 after the swap means both pegs must be in the wrong position after the swap.

(computational complexity of O(L)) is O(2L). Finding if the position is the position of the being-fixed color is O(L) and if L is equal to N then finding the color for that position in inferences is O(N). The worst case computational complexity for Getnext function is $O(3LN) = O(N^2)$ when L is equal to N. The Update function consists of adding to the inferences list, handling the case of 0, 1 and 2 incorrectly placed pegs and updating inferences list, being-fixed and being-considered colors. If we denote G for the number of correct times the being-considered appears in the code, then finding G is O(LN) and if making a sublist takes O(N), then adding to the inferences list is $O(GLN^2)$. Handling the case of 0 incorrectly placed pegs is O(2L). Worst case for handling the case of 1 incorrectly placed pegs is O(5L). Worst case for handling the case of 2 incorrectly placed pegs is O(5L). Updating the inferences list is $O(L^2)$, being-fixed is O(L), and being-considered is O(1). The worst case computational complexity of Update function is $O(GLN^2+L^2+ 5L+ 1) \Rightarrow O(N^2)$. Altogether, the worst case computational complexity of Baseline 4 is $O(GLN^2 + L^2 + 3LN + 5L + 1) = O(N^3)$ when L is equal to N.

According to Rao, given N pegs and M colors, the worst case number of guess is M + N attempts (Rao 1982), but this is actually incorrect. With insert-colors and 10000 rounds shown in the table below, we found out that expected number of guess is between O(N + M) and O(N*M). Since the possibilities for N are reduced by a fraction on every guess, we think there would be approximately O(logN) guesses made for each color, so the expected number of guesses is O(M*logN).

Table 1: Average number of guesses for Rao's algorithm (Baseline 4)

# of pegs and colors	Worst guess	Average guess
4 pegs 6 colors	9	6.2316
6 pegs 10 colors	17	10.9155
8 pegs 10 colors	23	14.1988
15 pegs 20 colors	56	37.651
20 pegs 26 colors	95	59.8937

Dino Local Search

Now, let's look at the local search. In the beginning, generating each guess to figure out the color distribution is not very complex- the program just generates a single list of one color. The next step is generating the first guess, which is trivial in terms of complexity.

Then there is the generate-successor function, which makes a random swap in the previous guess to generate a new guess. In the beginning of the process, this is relatively simple- basically any swap between two pegs of different colors is acceptable. If the two random indices chosen contain the same color, two new random indices are chosen. If the secret code contains only two colors, the indices have to be regenerated about N/2 times in the case where there are N-1 pegs of color A and 1 peg of color B. Each index selection has only a 1/N chance of choosing color B, but with two indices, there's a 1/N + 1/N = 2/N chance that one or both choose B, and an approximately 2/N chance that only one chooses B (since the case where both choose B is rare). So you would have to generate indices about N/2 times to get a valid swap, which can be simplified to O(N).

Most of the time though, early swaps will be much faster, especially if there are more than 2 colors in the code. If every peg is a different color, this will barely be a problem. But as more and more guesses are made, new swaps must also not result in a guess that was already made. Now worst case is generating indices about (N/2)*previous_guesses times. Since the amount of guesses is capped at 100, this is at worst O(99(N/2)) which can still be simplified to O(N). Finally, once knowledge is added, it can potentially take a long time to find a swap that fits with the knowledge base, or it could even be impossible until another swap is made. This is why we have epsilon (which is set to 50). So after many guesses, when it becomes harder and harder to generate guesses that fit the knowledge base, we may have to generate (N/2)*previous_guesses*50 indices. Again, since 50 is a constant, this is still O(N). Thus, generating indices to swap is in general an O(N) task in the worst case.

Adding knowledge does not require much complexity, since it's just adding information to a list. Iterating through knowledge has a complexity of O(2N) = O(N), because you're iterating through two lists (known-pegs and wrong-pegs) of length N. Total worst-case complexity is thus O(N) for generating indices to swap * O(N) for iterating through knowledge lists = O(N^2). Because there is randomness involved, there is a lot of variance here- in many cases complexity will likely be better, but if you get unlucky it could be very bad.

But in order to get the full picture, we need to look at the total number of guesses made. In the worst case, for N pegs and M colors, it takes M-1 guesses to figure out the color distribution, as we already discussed. But how many guesses are needed after that? To figure this out we have to look at how much each guess narrows down the possible results. Once we have the color distribution, there are N! possibilities for the correct guess in the worst case, if each peg has a different color (Weisstein).There are 3 possible cases for a swap: +2 or -2 correctly positioned pegs, which adds the most knowledge, +1 or -1 correctly positioned pegs, which adds some knowledge, or no change in correctly positioned pegs, which adds no knowledge. The worst case is starting with all pegs in the wrong position, and having each peg be a different color.

The chance that any one element is put in its proper position is 1/N. The chance that both elements are swapped into their correct positions is approximately $1/N*1/N = 1/N^2$. The chance that at least one of the two elements is put in its proper position is thus about $2/N - 1/N^2$. The chance that neither element is put in its correct position is approximately $1 - (2/N - 1/N^2) = 1 - 2/N + 1/N^2$

No change in properly positioned elements after a swap only subtracts 1 from the list of possible guesses, +1/-1 properly positioned element results in approximately 2/N of the guesses being eliminated (Each specific assignment for one peg exists 1/N of the guesses, and we're eliminating 2 assignments), and +2/2 properly positioned elements results in approximately 1- 2/N of the guesses being eliminated (Each specific assignment for one peg exists 1/N of the guesses, and we're confirming 2 assignments. This means 2/N of the guesses remain, so 1 - 2/N were eliminated). Now we can take a sum of each knowledge gain possibility weighted by the probability that possibility will occur (assuming there are g possible guesses with the known color distribution).

$(1 - 2/N + 1/N^2)(1) + (2/N - 1/N^2)(g*2/N) + (1/N^2)(g*(1 - 2/N))$
$(1 - 2/N + 1/N^2)(1) + (2/N - 1/N^2)(2g/N) + (1/N^2)(g - 2g/N)$
$(1 - 2/N + 1/N^2) + (4g/N^2 - 2g/N^3) + (g/N^2 - 2g/N^3)$
$(1 - 2/N + 1/N^2) + (4g/N^2 - 2g/N^3) + (g/N^2 - 2g/N^3)$
$1 - 2/N + 1/N^2 + 5g/N^2 - 4g/N^3$

This can be written as $O(g/N^2)$ guesses being removed each time. In other words about $1/N^2$ of the g possible guesses are removed after each guess. Let's say the number of guesses made is x:

$g * (1 - 1/N^2)^x = 1$
Since there are N! possible guesses to start, we can substitute g for N!

$N! * ((N^2 - 1)/N^2)^x = 1$
$((N^2 - 1)/N^2)^x = 1/N!$
log base $((N^2 - 1)/N^2)$ of $1/N! = x$
log base $N^2/(N^2 - 1)$ of $N! = x$
$O(\log N!) = x$

This means it would take on average O(log N!) guesses to get the correct answer after figuring out the distribution. This is a total worst case complexity of O((M-1) + log N!) = O(M + log N!). Again, this is variable - the search could get very lucky or unlucky and take much shorter or longer.

Experimental Evaluation:

We tested our solver in tournament of 100 rounds for 8, 10 problem on insert-colors, two color, ab-color, two-color-alternating, only-once, first-and-last. Our algorithm for two-color-alternating is generated from baseline 3. Algorithms for the other SCSAs are generated from baseline 4. Our general results are in Table 2. We analysed CPU time and number of guesses in Table 3.

Table 2: General result of 8, 10 problem on six SCSAs

	insert-colors	two-color	ab-color	two-color-alternating	only-once	first-and-last
peg 8 color 10	(SCORE 130.40373) (26.080746 0 0)	(SCORE 145.2613) (28.698708 0 0)	(SCORE 193.54883) (38.301517 0 0)	(SCORE 154.58192) (30.916384 0 0)	(SCORE 129.81845) (25.69643 0 0)	(SCORE 140.67484) (27.857616 0 0)

Each cell in Table 2 contains the result of running corresponding SCSA on 8,10 problem. And, each result is constructed with a Score#, followed by a list of (Wins, losses, failures). As shown in the table, our solver worked well. We don’t have losses or failures in each tournament.

Table 3: CPU time (in microseconds) and number of guesses of 8, 10 problem on six SCSAs

	insert-colors	two-color	ab-color	two-color-alternating	only-once	first-and-last
number of guesses to win	12	10	7	10	14	10
Time for the first guess	85	65	85	85	54	92
AVE CPU time for guess afterwards	48.9167	47	50.5714	41	53.3571	48.3
time for 1 round	587.0004	470	353.9998	410	746.9994	483
time for 100 rounds	58700.04	47000	35399.98	41000	74699.94	48300

As shown in Table 3, our solver works better on two-color, ab-color, two-color-alternating, first-and-last than on insert-colors. Works not better than insert-colors on only-once. The reason is that only-once is dependent on a random chooser like insert-colors. Generating a strategy for random code generation is not reasonable. It sometimes just depends on how lucky you are. only-once in particular is even more difficult than insert-colors because it has no color duplicates. Duplicates of the same color tend to make baseline 4 more efficient, as described by Rao (Rao 1982). With only-once, there is no chance of this occurring.

Scalability:

We tested our program in a series of tournaments of increasing difficulty (pegs × colors). We chose (peg 4, color 6), (peg 6, color 10), (peg 8, color 10),(peg 15, color 20), and we also challenged (peg 20, color 26). We tested all the known SCSAs. The general running result is in Table 4. Then we analyzed the CPU time and number of guesses to win on “two-color-alternating” and “first-and-last”. The results of these two SCSAs are in Table 5 and Table 6.

Table 4: Our results at a glance

	insert-colors	two-color	ab-color	two-color-alternating	only-once	first-and-last
peg 4 color 6	(SCORE 204.98224) (40.99645 0 0)	(SCORE 218.69473) (43.738945 0 0)	(SCORE 278.28067) (55.65613 0 0)	(SCORE 196.55318) (39.310635 0 0)	(SCORE 200.97339) (39.78643 0 0)	(SCORE 201.39159) (40.278316 0 0)
peg 6 color 10	(SCORE 149.33899) (29.867798 0 0)	(SCORE 170.81734) (34.163467 0 0)	(SCORE 205.45203) (41.090405 0 0)	(SCORE 154.5819) (30.91638 0 0)	(SCORE 151.26634) (29.937042 0 0)	(SCORE 154.74637) (30.949272 0 0)
peg 8 color 10	(SCORE 130.40373) (26.080746 0 0)	(SCORE 141.07857) (28.215714 0 0)	(SCORE 167.11516) (33.42303 0 0)	(SCORE 154.58192) (30.916384 0 0)	(SCORE 131.07928) (26.215858 0 0)	(SCORE 131.30908) (26.261816 0 0)
peg 15 color 20	(SCORE 77.728714) (15.545742 0 0)	(SCORE 88.15413) (17.630825 0 0)	(SCORE 89.92088) (17.984177 0 0)	(SCORE 110.51003) (22.102007 0 0)	(SCORE 77.79538) (15.559076 0 0)	(SCORE 77.52694) (15.505387 0 0)
peg 20 color 26	(SCORE 64.99981) (12.874962 0 0)	(SCORE 70.45442) (14.090885 0 0)	(SCORE 69.99224) (13.998448 0 0)	(SCORE 97.10491) (19.420982 0 0)	(SCORE 60.542862) (12.108572 0 0)	(SCORE 67.594154) (13.518831 0 0)

Table 4 is shows our results tested on all the known SCSAs. Each cell in Table 4 contains the result of running corresponding SCSA and the chosen (#peg, #color). Each result contains a score followed by a list of (wins, losses, failures). The results come from playing a tournament of 100 rounds. As shown in the table, our solver worked well. We had no losses or failures so far. Our solver was even able to solve the 20,26 problem.

Table 5: CPU time and number of guesses comparison on “two-color-alternating”

	CPU time		Guesses to Win	Time for 1 round	Time for 100 rounds
	Time for first guess	AVE Time for each guess			
peg4 color6	115	31.6	6	189.6	18960
peg6 color10	53	34.7	11	208.2	20820
peg8 color10	129	39.8	11	238.8	23880
peg15 color20	314	57	20	342	34200
peg20 color26	242	67.8	26	406.8	40680
Mean	170.6	46.18	14.8	277.08	27708
Stardard Deviation	105.2535035	15.56894345	8.043631021	93.41366067	9341.366067

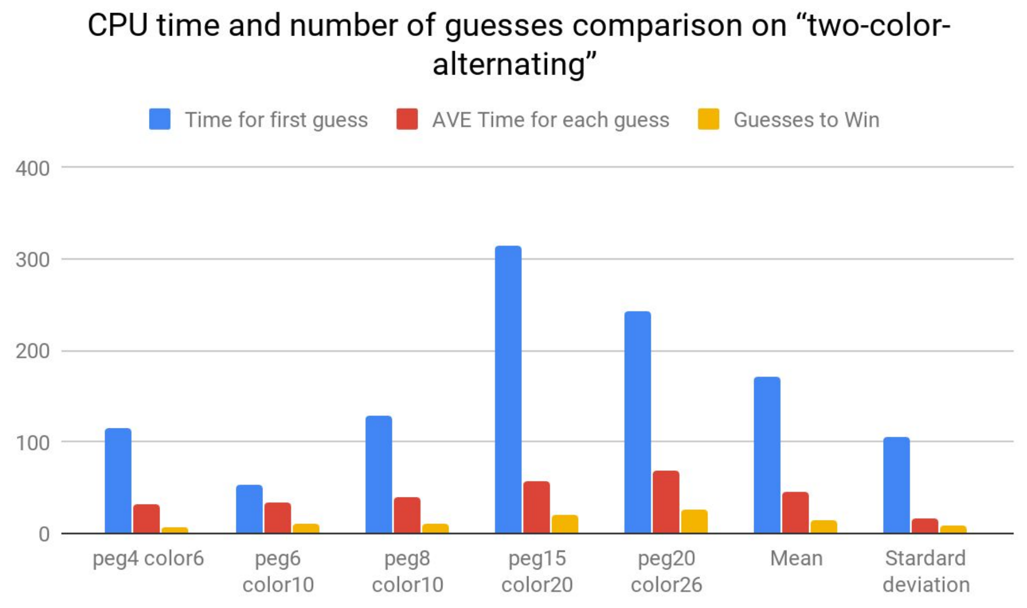


Table 6: CPU time and number of guesses comparison on on “first-and-last”

	CPU time		Guesses to Win	Time for 1 round	Time for 100 rounds
	Time for first guess	AVE time for each guess			
peg4 color6	70	64.5	6	387	38700
peg6 color10	86	43.4167	12	521.0004	52100.04
peg8 color10	81	53.5833	12	642.9996	64299.96
peg15 color20	96	72.82051	39	2839.99989	283999.989
peg20 color26	77	86.5	44	3806	380600
Mean	82	64.164102	22.6	1639.399978	163939.9978

Stardard Deviation	9.772410143	16.70588689	17.51570724	1577.001335	157700.1335

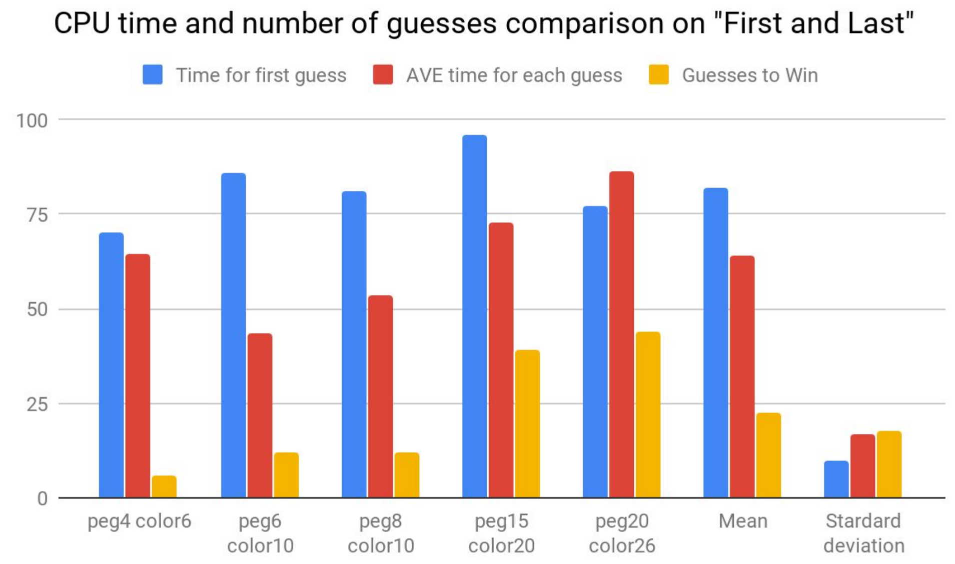


Table 5 and Table 6 are the CPU time and number of guesses on “two-color-alternating” and “first and last”. We used function “get-internal-run-time” to get the current run time in internal time units (microseconds). The time for guesses are from play tournament of 1 round about 5 times and take the average result. We used two formulas as below to evaluate the CPU time and number of guesses performance.

Formula 1 : Get the time taken for each guess:

Time took for taken guess = (get-internal-run-time) after guess - (get-internal-run-time) before guess”[2]

Formula 2 : Calculate the time for playing a 100 round tournament:

Time for 100 round = Time for one round * 100
= Average time for each guess * number of guesses to win*100

We listed CPU time for the first guess in the table. The reason is that it requires more time than other guesses in general. However, we observed that time for the first guess is not always the longest time in each round. Sometime, the second last guess or guess in middle took surprisingly longer time than other guesses.

[2] The difference between the values of two calls to function “get-internal-run-time” is the amount of time between the two calls during which computational effort was expended on behalf of the executing program.(http://clhs.lisp.se/Body/f_get__1.htm)

As shown in the three tables above, we observed that as the number of pegs and colors increase, the score and number of wins decrease. At the same time, CPU time for each guess and number of guesses to win increase. To show our player's scalability, we compared score, number of wins, number of losses and number of failures of playing "two-color-alternating" and "first-and-last" on 4,6; 6,10; 8,10;15,20 and 20,26 problems. We omitted number of losses and number of failures in charts because they are all 0s.

Table 7: Score and Number of wins on playing "two-color-alternating"

	Score	Number of wins	Number of losses	Number of failures
peg 4 color 6	196.55318	39.310635	0	0
peg 6 color 10	154.5819	30.91638	0	0
peg 8 color 10	154.58192	30.916384	0	0
peg 15 color 20	110.51003	22.102007	0	0
peg 20 color 26	97.10491	19.420982	0	0

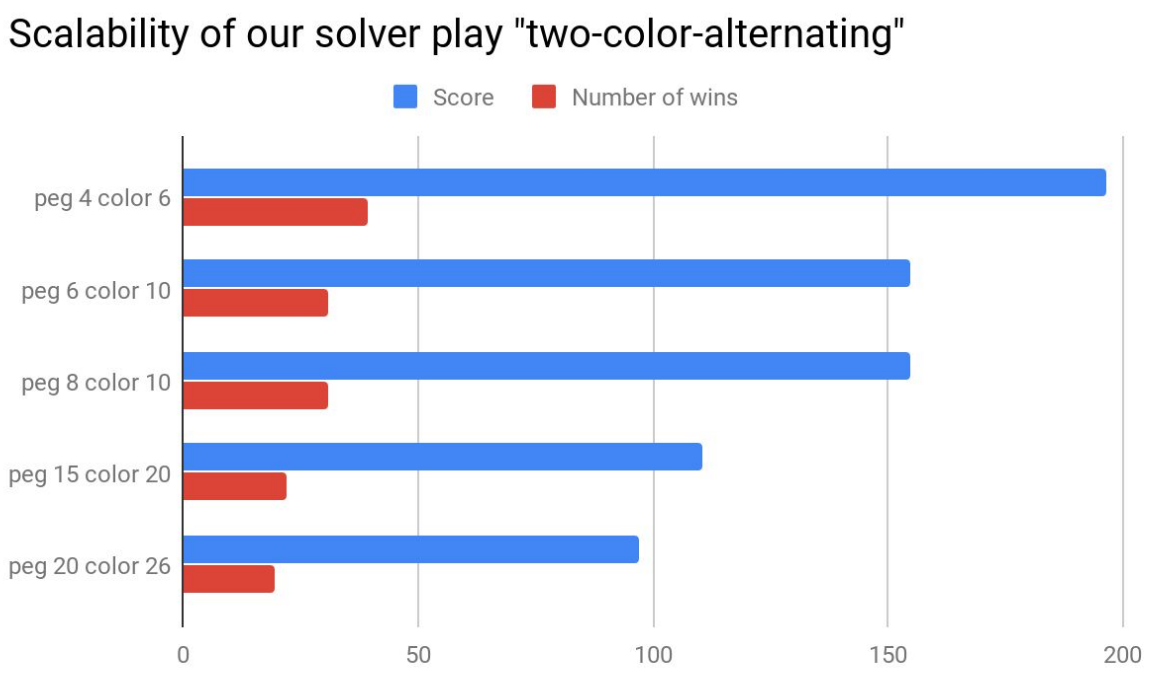
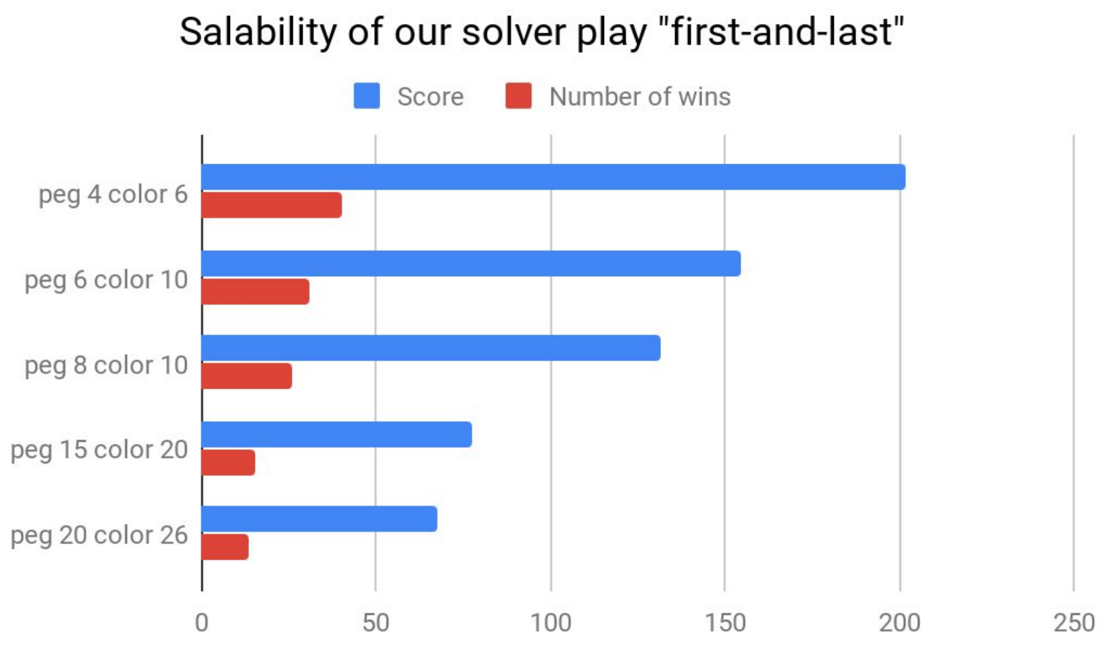


Table 8: Score and Number of wins on playing "first-and-last"

	Score	Number of wins	Number of losses	Number of failures
peg 4 color 6	201.39159	40.278316	0	0
peg 6 color 10	154.74637	30.949272	0	0
peg 8 color 10	131.30908	26.261816	0	0
peg 15 color 20	77.52694	15.505387	0	0
peg 20 color 26	67.594154	13.518831	0	0



The Learning Challenge

To measure how well we achieved our goals of learning based on knowledge of the SCSA or based on responses to previous guesses, we compared baselines 3 and 4 to dino-local-search and our main solver Dino, which chooses a function accordingly based on the SCSA it is playing against. With this data, we are able to see how much better Dino performs using the SCSA information than the baselines or dino-local-search do. We can also see how the use of knowledge collected from previous guesses allows local-search to outperform baseline 3 and baseline 4 to (most of the time) outperform baseline 3 and local search.

Table 9: CPU time and number of guesses on “two-color-alternating” for 8,10

	CPU time (in microseconds)		Number of guesses to win	Time for 1 round (in microseconds)	Time for 100 rounds (in microseconds)
	Time for first guess	Average time for each guess			
baseline-3-Dino	142	111.28814	59	6596.00026	659600.026
dino-local-search	142	143.89473	19	2733.99987	273399.987
baseline-4-Dino	132	92.8125	16	1485	148500
Dino	140	126.09091	11	1387.00001	138700.001

Table 10: CPU time and number of guesses comparison on “ab-color” for 8,10

	CPU time (in microseconds)		Number of guesses to win	Time for 1 round (in microseconds)	Time for 100 rounds (in microseconds)
	Time for first guess	Average time for each guess			
baseline-3-Dino	318	137.27777	18	2470.99986	247099.986
dino-local-search	143	131	9	1179	117900
baseline-4-Dino	136	130.41667	12	1565.00004	156500.004
Dino	143	140.77777	9	1266.99993	126699.993

Table 11: CPU time and number of guesses comparison on “first-and-last” for 8,10

	CPU time (in microseconds)		Number of guesses to win	Time for 1 round (in microseconds)	Time for 100 rounds (in microseconds)
	Time for first guess	Average time for each guess			
baseline-3-Dino	138	115.11	>100 (does not win)	11511	1151100
dino-local-search	139	128.8421	19	2447.9999	244799.99
baseline-4-Dino	141	115.64286	14	1619.00004	161900.004
Dino	142	134	11	1474	147400

For the previous three known SCSAs, the main solver Dino uses knowledge about the SCSA to select a function tailored for that SCSA. In all of these cases, Dino outperforms baseline-4-Dino because baseline 4 completely ignores which SCSA it’s playing against. We can also see that local-search and baseline 4 always significantly outperform baseline-3 in terms of number of guesses, because they gather knowledge from previous guesses and make use of it throughout the round, while baseline-3 stops collecting knowledge once it knows the color distribution. In terms of time, baseline 4 tends to be the fastest but Dino is not dramatically slower, and is quicker overall in one round since it makes fewer guesses. One interesting thing to note is that for ab-color, shown in table 11, dino-local-search actually takes the same amount of guesses as Dino, and fewer guesses than baseline 4. This just goes to show that different situations require different learning approaches. Even though on every other SCSA shown in tables 9-10 and 12-14, baseline 4 takes fewer guesses than dino-local-search, we cannot conclude that baseline 4 is always better. As they say in machine learning, there is no free lunch. This is why the best approach is to select different functions for different SCSAs - different learning approaches will work better in different cases.

Table 12: CPU time and number of guesses comparison on “only-once” for 8,10

	CPU time (in microseconds)		Number of guesses to win	Time for 1 round (in microseconds)	Time for 100 rounds (in microseconds)
	Time for first guess	Average time for each guess			
baseline-3-Dino	139	110.59	>100 (does not win)	11059	1105900
dino-local-search	139	124.206894	29	3601.99993	360199.993
Dino (using baseline-4)	140	132.9	10	1329	132900
dino-only-once	139	116.56	25	2914	291400

Table 12 shows an example of an unsuccessful learning strategy. Although dino-only-once takes fewer guesses than baseline-3 or dino-local-search for the only-once SCSA, baseline-4 still outperforms it. For this reason, we chose to use baseline-4 for only-once in the main Dino program, even though it was not created specifically for only once. Still, as mentioned earlier, only-once is the known SCSA we perform worst on so there is room for improvement here. If we had more time, we would attempt to create another function specifically for only-once that could actually outperform baseline-4.

Table 13: CPU time and number of guesses comparison on “mystery-2” for 7,5

	CPU time (in microseconds)		Number of guesses to win	Time for 1 round (in microseconds)	Time for 100 rounds (in microseconds)
	Time for first guess	Average time for each guess			
baseline-3-Dino	140	92.69512	82	7600.99984	760099.984
dino-local-search	147	136.57143	14	1912.00002	191200.002
Dino (using baseline-4)	107	101.55556	9	914.00004	91400.004
dino-mystery-2	148	112.3	10	1123	112300

Table 14: CPU time and number of guesses comparison on “mystery-5” for 7,5

	CPU time (in microseconds)		Number of guesses to win	Time for 1 round (in microseconds)	Time for 100 rounds (in microseconds)
	Time for first guess	Average time for each guess			
baseline-3-Dino	139	134.92	25	3373	337300
dino-local-search	145	117.1	10	1171	117100
baseline-4-Dino	137	127.44444	9	1146.99996	114699.996
Dino	141	117.6	5	588	58800

The two mystery functions we had a chance to create functions for were mystery-2 and mystery-5. We ran these on 7,5 instead of 8,10 because we were only given samples for the 7 peg, 5 color case. The results of running the samples for these two mystery functions are shown in tables 13 and 14. As you can see, in the case of mystery-2 (Table 13), again, the function designed specifically to play the SCSA could not beat the number of guesses baseline-4 took to solve it. The performance between the two functions does not differ immensely, but we still chose to use baseline 4 over dino-mystery-2 because it was a little better, and also because we do not know exactly how mystery-2 will scale up, since we only have 7,5 samples. Because we know that baseline-4 can scale relatively well even on random SCSAs, it is a safe bet to use baseline-4 on a mystery SCSA if we do not have anything that can beat baseline-4’s performance on it for the 7,5 case.

Meanwhile, for mystery-5 (Table 14), our specific function called by Dino (dino-two-color-alternating, since mystery-5 looks to also generate codes wits two colors alternating) took fewer guesses than baseline-4. In this case, even though we don’t know exactly what mystery-5 will look like scaled up, since we have proven better results than baseline 4 on the 7,5 case it’s worth it to at least try to take knowledge of the SCSA into account.

Overall, there are many situations where our solver learns, either from the name of the SCSA it’s going up against or from the results of previous guesses. We have done our best to optimize our code so it uses whatever strategy is best for a particular problem.

Works Cited

desJardins, Marie and Tim Oates, April 2011, University of Maryland Baltimore County
http://modelai.gettysburg.edu/2011/mastermind/index.html

Knuth, Donald E. "Generating All Combinations", June 2004, Stanford University,
http://www.kcats.org/csci/464/doc/knuth/fascicles/fasc3a.pdf.

Rao, T. Mahadeva. “An Algorithm to Play the Game of Mastermind”, October 1982,
SUNY College at Brockport,
https://dl.acm.org/citation.cfm?id=1056607&dl=ACM&coll=DL.

Weisstein, Eric W. "Permutation." From MathWorld--A Wolfram Web Resource.
http://mathworld.wolfram.com/Permutation.html