Scrambling

Most simulations would need a scrambling function, to mix up the puzzle for the user to solve. A good scrambling method should make all possible positions roughly equally probable. The easy way to scramble is just generate a list of random moves and perform them starting from a solved puzzle. There are some notable issues however that you have to keep in mind:

1. It is not always clear how many moves you need to give a reasonably fair shuffle. The speed of computers generally makes this a moot point as in practice you can usually do enough moves for a reasonable mix. This is only really an issue for old programmable calculators.
2. Some puzzles such as the Rubik's Cube have parity effects, and then using a fixed number of random moves will always give positions with a particular parity. Randomly varying the number of random moves alleviates that problem though.
3. A random move may undo the previous random move. It is best to avoid this to ensure the puzzle is mixed well, by remembering the previous move and never moving the same face twice. If you decide not to bother with this, you will have to do a larger number of random moves. Note that moves can cancel out even if they are not immediately consecutive, such as RLR' on the cube. To also avoid this you have to allow a pair of commuting moves only in one order (e.g. allow RL but not LR).
4. With some types of puzzles - bandaged puzzles in particular - there is no number of moves you can do to make all possible positions roughly equally probable. This is because some positions are easier to reach than others. If a position on a bandaged puzzle (e.g. Square-1) has more moves available, then it will be more likely, because more moves can reach that position.
5. Puzzles with moves that are one-way (i.e. cannot be directly undone) may not be solvable after mixing in this way. This can be fixed by instead scrambling the puzzle by undoing random moves beginning with start.

For many people these issues are not a problem, and this method is easy to implement. If each facelet has its own array entry instead of just each piece, then this method is much easier than the ones that follow below.

There is another way to scramble however that are indisputably fair (in the sense that all positions are equally probable, assuming a fair random number generator). This is usually done by selecting a random permutation. In essence you are shuffling the puzzle just like a deck of cards - the pieces are reordered in a random fashion as if the puzzle was taken apart and reassembled. If all n! permutations of a set of puzzle pieces is solvable, then this is a perfect method. The pseudo code for this is as follows:

```
function mix ( array p )
  for i from 1 to N-1
    r = random number between i and N inclusive
    if i not equal to r then
      swap values of p[i] and p[r]
    endif
  endfor
endfunction
```

This routine assumes p is an array of piece locations 1 to N, with values indicating the pieces at those locations which are to be mixed. Unfortunately, there are many puzzles with parity restrictions. If you wish to randomly choose an even permutation of the pieces, then you could mix it as above, check the parity, and if the parity is odd do a single swap. A better method is shown here:

```
function mixEven ( array p )
  for i from 1 to N-2
    r = random number between i and N inclusive
    if i not equal to r then
      swap values of p[i] and p[r]
      swap values of p[N] and p[N-1]
    endif
  endfor
endfunction
```

Note that in the first routine the *if* statement was not really necessary (swapping p[i] and p[r] when r=i does nothing anyway), but here it is essential in order to preserve the even parity. The change from N-1 to N-2 as the loop range is not important though - if you keep it as N-1 the last loop iteration just does nothing.

On the Rubik's Cube the edges and the corners have the same parity, either both odd or both even. The way to achieve that is simple. Use the mixEven method above to mix the edges, and then again to mix the corners. This makes both parities even. Now randomly choose even or odd, both 50% probability. If even was chosen you are done already. If odd was chosen just swap the first two corner locations, and swap the first two edge locations. This makes both permutations odd as required.

Scrambling the pieces orientations if the puzzle has them is a much easier task. If there is no total twist restriction, then every piece can simply be assigned a random orientation. If there is a twist restriction, you can just choose randomly the first N-1 orientations, and adjust the Nth orientation as you do so to keep the total orientation zero.

If your program will have routines for indexing permutations (explained below) then there is a simpler method. Just choose a random number between 1 and n!, and convert that index to a permutation. If that index conversion routine is already written (or will have to be written anyway), this is the easiest method of all.

Calculation of God's Algorithm

God's algorithm is the name for the method of solving a puzzle which always uses the fewest possible moves. There are a few puzzles for which this already completely known (e.g. Pyraminx, 2×2×2 cube, Skewb), but many puzzles have just too many possible positions to be able to calculate God's Algorithm for all possible positions. As the cost of memory and processing power comes down more and more, puzzles that were previously out of reach become possible. Yet there are many puzzles which have such astronomical numbers of positions that it is hard to imagine that they will ever be fully solved.

For any position, God's algorithm can give a move that brings the puzzle one step closer to being solved. It is essentially just a large database, in which a position can be looked up, and from which you can extract the move to perform. To fully solve a position, just keep looking up the current position and doing the move the database tells you to, and eventually the puzzle will be solved. To find God's algorithm you therefore have to generate such a database only once, after which it can be reused forever.

The database is generated in reverse, beginning with the starting positions, in the following manner:

1. Make an empty database that is large enough to store all the data.
2. Put the solved position (or positions) of the puzzle in the database. For all these solved positions perform the following step.
3. For each of these positions, and each possible move do this:
   a. perform the move, and see what position is generated.
   b. If this position is not yet in the database, then store it together with the move that solves it (the inverse of the move that was just done).
4. For each of the positions that have just been generated, repeat the previous step. If no new positions were generated, then the database is finished.

This is often called a 'breadth first' search. This means that it visits all positions at distance 1 before visiting all positions at distance 2, and so on.

Reducing memory use, Indexing

The database used can be huge. It is therefore essential to reduce the storage space needed per position. The most common way to do this is by indexing. By using indexing, the positions of the puzzle are not themselves stored, but are given by the index in the database or table. Suppose that you have found a way to associate with each position of the puzzle a unique number in the range 1 to N. You can then use a table with N entries, and the index in the table determines which puzzle position the data pertains to.

Another useful point is that you don't have to store the moves, but instead you can store only the distance from solved. The table still allows you to know what the best moves are in each position; simply try out all available moves and see if the new position is closer to being solved by looking it up in the table. Most puzzles can be solved in fewer than 256 moves, so you need no more than a single byte per puzzle position.

For very large puzzles, it might be necessary to use even less than a byte per position. If you know that the puzzle diameter is less than 15, you need only 4 bits to store the distance. In fact, you can go even further. Instead of storing the distance, store the distance modulo 3 (i.e. the remainder from division of the distance by 3). This is still enough to give God's algorithm, because it is still possible to check whether a move brings it a step closer to start or not. Note that 2 bits have 4 possible values, so while generating the table we can use the fourth value to indicate entries that have not yet been filled.

Many puzzles have symmetry. It may be possible to use such symmetry to reduce the size of the tables. For example if every position also has a mirror image position, and none of the positions themselves are mirror symmetric, then the table can be cut to half size. Similarly for rotational symmetries. Indexing this reduced set of positions may be a bit more tricky however.

Many puzzles are still too large to be fully calculated, even when only 2 bits per position are used. In such cases, it is only possible to calculate all positions which lie within a certain distance from solved. In that case you will have to store the positions that have been visited in the database. The techniques used for indexing are still useful, because they allow you to encode a position in a unique number (or list of numbers) which is easy to store and doesn't take up as many bits as most other ways of storing a position.

Click this link for more detailed pseudo code examples of database generation and indexing.

Thistlethwaite's algorithm

In the early 1980s Morwen B. Thistlethwaite came up with a very clever method of solving the Rubik's cube. It is clearly not (yet) possible to calculate God's algorithm for the whole cube, so he decided to split up the problem into 4 sub-problems, each of which are just like solving a simpler puzzle and which can be done using a table.

To achieve this he used a sequence of nested groups (see the Theory page). The whole cube group is generated by the moves {U, D, R, L, F, B}, and this is the first group G_0 in the sequence. Successive groups restrict various moves to half turns only as follows:

G_0 = <U, D, R, L, F, B>
G_1 = <U, D, R2, L2, F, B>
G_2 = <U, D, R2, L2, F2, B2>
G_3 = <U2, D2, R2, L2, F2, B2>
G_4 = I

The first sub-problem is how to use any moves to get a position which lies in G_1, i.e. to get a position that can be solved using only {U, D, R2, L2, F, B}. From that new position, the second sub-problem is how to use only those restricted moves to get to a position that can be solved by using only {U, D, R2, L2, F2, B2}. This continues, each stage uses fewer types of moves to reach a position that can be solve with even fewer. Eventually in the last stage it is solved.

Note how each stage can be considered a standalone puzzle in itself because the elements of G_i form a subgroup. This is different from the usual methods of solving a cube, because usually previously solved pieces have to be temporarily moved while solving later pieces. Here, any progress made during earlier stages remains intact throughout.

In the first stage, the orientation of the edges is solved. It is impossible to flip an edge piece once the R and L faces are restricted to half turns. Any position that has no edges flipped lies in G_1 (because it can be shown that it can be solved using those restricted moves). Therefore the size of the lookup table for the first stage is only 2^{11} = 2048. It can easily be generated by the same methods as for God's Algorithm. To solve a cube position with Thistlethwaite's algorithm, the first stage is solved simply by looking only at the flip of the edges and ignoring all else.

The other stages work exactly the same way, though they are not all quite so easy. You have to build some kind of index into the table, which only encapsulates those aspects of the cube which are solved at that stage.

Thistlethwaite's algorithm solves the cube in at most 7+10+13+15= 45 moves, with an average of 4.6+7.8+8.8+10.1= 31.3 moves.

Note that if you take a different shortest route through stage 1, the route through stage 2 may be shorter or longer than before. This in turn influences the length of stage 3 or 4. The only guarantee is that it will not exceed 45. Another interesting fact is that some positions in G_3, the square group, can be solved in fewer moves when quarter turns are allowed.

Tree Searching

This is a method of solving a particular position by trying out all combinations of moves until it is solved. In a tree search of depth 1, every move is tried out from the scrambled position, to see if it becomes solved. If it doesn't, you know that it is more than 1 move away from start. Deeper searches can be done by recursion; for a search of depth n try out each single move followed by a tree search of depth n-1. In pseudo code it looks something like this:

```
function Treesearch( position p; depth d )
  if p is solved then
    Hooray!
  elseif d>0 then
    for each available move m
      Treesearch( result of m applied to p; d-1 )
    endfor
  endif
endfunction
```

The biggest advantage of a tree search is that it needs very little memory to run as it has no databases or tables.

Iterative deepening

If you have a scrambled puzzle, and you are sure that its solution will be at most n moves, then a tree search of depth n will find it. If, unbeknownst to you, the position of the scrambled puzzle is actually only a few moves from solved, then the search may take much longer than necessary. It may have to search many positions at depth n before finding a shorter solution. Tree searches are called 'depth first' searches for this reason, as it reaches out to the maximum depth before visiting all positions that are closer by. Such a tree search might also find a long alternative solution instead of the short one you actually would want.

To overcome these problems, use iterative deepening. This simply means that you do a tree search of depth 1, then one of depth 2, and so on, until a solution is found. This gives a hybrid of a breadth-first and depth-first search. It does repeat some work each time, but overall it is quicker because deep searches take a long time. Furthermore it will always find the shortest possible solution. This algorithm is known by the abbreviation IDA.

Pruning

Tree searching is still slow. It would be much quicker if it doesn't need to search as deeply. Suppose we have done 5 moves so far in a depth 12 search. Only very few (if any) of the branches in any tree search reach a solution, so if we could somehow know that all branches that continue from this position are dead ends, i.e. that the current position cannot be solved within 7 moves, then we need not try any further.

A pruning table gives just this kind of information. For each position, it gives a lower bound on the number of moves needed to solve it. At first glance it seems that it is ridiculous because if you had such a table, you might as well have used it to store God's Algorithm. This is not the case as a pruning table does not have separate entries for each puzzle position. Each entry in a pruning table represents many puzzle positions. All these positions are further away from solved than the distance given by the table. A pruning table is calculated using exactly the same methods as were used for God's algorithm, except that instead only part of the position is used to index the database.

It is easiest to explain by giving an example. Consider the 2×2×2 Rubik's Cube. There are $N_1 = 7!$ possible piece permutations, and $N_2 = 3^6$ possible piece orientations. When calculating God's algorithm, we would convert a puzzle position into a permutation number n_1, an orientation number n_2, and then use these as indices into a large two-dimensional table of N_1 by N_2. For pruning tables we could use one small table of size N_1 for the permutation and one small table of size N_2 for the orientation. These two tables contain the minimum values of the distances in the columns and rows of the God's Algorithm table.

If it takes 5 moves for example to solve only the orientations of the pieces, without even bothering about the permutations, then it will take at least that many to solve the position fully. At each step in the tree search you can check whether you still have enough moves left to solve the orientation, and similarly if there are enough moves left to solve the permutation. If so, then continue searching, but if not then you don't have to look any further from this position and have to backtrack and try something else.

This algorithm is sometimes given the name IDA*. The code for this algorithm is as follows:

```
function IDA ( position p )
  for depth d from 0 to maxLength
    Treesearch( p; d )
  endfor
endfunction

function Treesearch( position p; depth d )
  if d=0 then
    if p is solved then
      Hooray!
    endif
  elseif d>0 then
    if prune1[p]<=d and prune2[p]<=d then
      for each available move m
        Treesearch( result of m applied to p; d-1 )
      endfor
    endif
  endif
endfunction
```

In this example code I have assumed that there are two pruning tables, but it can obviously be changed for any number of them.

Increasing speed, Transition tables

The pruning tables improve the speed of a tree search significantly, more so if they are carefully chosen. It is best if every essential part of the puzzle position is used in some way in a pruning table, the pruning tables are as large as possible, and also independent of each other. Independent in this sense means that for any indices n_1 and n_2 into two pruning tables, there is some puzzle position that has those two indices.

One aspect that I have not clarified is the phrase 'available move' when performing a tree search. It is clearly undesirable to undo the move that has just been performed at the previous level of the tree. Often moves should not be repeated either because instead of doing on a Rubik's Cube for example R followed by R, we could have done R2 straight away (assuming that is considered a single move). Simply put, any R, R2, or R' move should not be followed by R, R2, or R' because otherwise the search checks many positions several times. Also, since RL=LR we can also assume that R (or R2, R') never follows L (or L2, L'). Most puzzles have similar simple identities that can be checked.

Another significant way of speeding up the search is reducing the overhead at each node in the tree. At most nodes, the current position has to be looked up in pruning tables, changed by performing a move and then passed on to the next level of the tree search. To look up a position in the pruning tables, it needs to be converted to one or more index numbers. It is therefore best if that conversion is unnecessary because the current position is already given by such a set of indices. To do a move on a position given by a set of indices, you can use transition tables. These are tables that give the indices for the new position resulting from applying a move to the previous position.

Again an example is in order. On the 2×2×2 cube, each position is completely determined by the orientation of the pieces, and the permutation of the pieces. The position is therefore encoded in two indices. We can have a pruning table for each index as explained before. We also have a transition table of size $N_1 \times M$, which gives the effect on the first index of any move. Here M is the number of possible moves. Similarly there is a transition table for the second index too, of size $N_2 \times M$.

A puzzle of which the positions are given by two indices can be searched by using the code below. Extending to more indices if necessary is trivial.

```
function Treesearch( Index1 p1; Index2 p2; depth d )
  if d=0 then
    if position p1;p2 is solved then
      Hooray!
    endif
  elseif d>0 then
    if prune1[p1]<=d and prune2[p2]<=d then
      for each available move m
        Treesearch( trans1[p1,m]; trans2[p2,m]; d-1 )
      endfor
    endif
  endif
endfunction
```

In the above I have assumed that the transition tables give all possible moves. On the 2×2×2 cube this would mean that M=9, because the moves are {R, R2, R', F, F2, F', U, U2, U'} (and we consider piece BLD fixed). This is not necessary, as we only need to have M=3 to encode the transitions by moves {R, F, U} and the others can then be deduced by repeatedly applying the transition table. This can save on memory storage which can be better applied by having larger pruning tables.

Kociemba's Algorithm

All the discussion above assumes that you actually want to find the shortest solution. A tree search might take a long time even with good pruning tables. Herbert Kociemba managed to combine several ideas into a very effective new algorithm which will give good sub-optimal solutions very quickly. It may well find an optimal solution to a position fairly soon, but it may take a long time for it to actually prove the solution is optimal by trying out all shorter sequences.

The first idea was based on Thistlethwaite's work. Kociemba uses only two phases however. His first phase moves from G_0 to G_1 = <U, D, R2, L2, F2, B2>, and the second phase from G_1 to G_2 = I. So each of Kociemba's phases combines two of Thistlethwaite's stages together. Clearly these phases are too large to build a full table. Nevertheless, it is quite straightforward to make pruning tables and use IDA* to solve each phase. Already this two-phase algorithm will clearly lead to some improvement on Thistlethwaite, because the first phase of this algorithm finds the shortest possible way of going through the first two parts of Thistlethwaite, and the second phase the shortest way to go through the last two parts. The phases have actually been fully calculated and their maximum lengths were found to be 12 and 18 moves, so the algorithm's first solution is at most 30 moves long (using the Half Turn Metric). In fact it has been proven that the 29 and 30 move cases can be avoided, so that every cube position can be solved in at most 28 moves. For a while this was the best upper bound for God's Algorithm until different methods reduced to 22 moves.

The second major idea was to continue searching after the first solution is found. As was noted with Thistlethwaite's algorithm, a different phase 1 solution may well have a better possible phase 2 solution. After one solution is found, other phase 1 solutions are tried to see if these give a better phase 2 and hence a better overall solution.

This is not the end of it though. Suppose we have found a solution of length 7+8, and all the other phase 1 solutions of length 7 have been tried. By continuing the phase 1 IDA* even further, it tries to find solutions of length 8, followed by a phase 2 solution of at most 6 (so that the sum 8+6 is less than the solution we have so far). Eventually, as the phase 1 search depth increases, the length of allowed phase 2 solutions decreases until phase 1 has reached the length of the best solution found so far. This means that it is an optimal solution.

The basic outline of the code is listed below. It is essentially two copies of the IDA code we saw previously, except for the parts in bold type.

```
maxLength=9999

function Kociemba ( position p)
  for depth d from 0 to maxLength
    Phase1search( p; d )
  endfor
endfunction

function Phase1search( position p; depth d )
  if d=0 then
    if subgoal reached and last move was a quarter turn of R, L, F, or B then
      Phase2start( p )
    endif
  elseif d>0 then
    if prune1[p]<=d then
      for each available move m
        Phase1search( result of m applied to p; d-1 )
      endfor
    endif
  endif
endfunction
```

```
function Phase2start ( position p)
  for depth d from 0 to maxLength - currentDepth
    Phase2search( p; d )
  endfor
endfunction

function Phase2search( position p; depth d )
  if d=0 then
    if solved then
      Found a solution!
      maxLength = currentDepth-1
    endif
  elseif d>0 then
    if prune2[p]<=d then
      for each available move m
        Phase2search( result of m applied to p; d-1 )
      endfor
    endif
  endif
endfunction
```

One important subtlety is that phase 1 should end on a move that does not occur in the move set of phase 2 (i.e. F, F', B, B', R, R', L, or L'). If you do not do this, then all the same move sequences will be searched again after the phase 1 length is increased.

This algorithm is extremely effective. It very quickly finds solutions which are not far from being optimal. To see just how powerful this algorithm is, you can download Herbert Kociemba's incredible **Cube Explorer** from his page.

Although Kociemba originally designed his algorithm for solving the normal Rubik's cube, it is possible to apply it to other puzzles. My first Square-1 solver uses it for example, and I also once made a Siamese Cube solver. Any puzzle for which you can find a subgroup (generated by single moves) that makes the two phases roughly equal, and which is small enough to be calculated by IDA*, can have the Kociemba algorithm applied to it.

Multi-phase Algorithms

Let's reconsider the Thistlethwaite algorithm. It has four stages, where each stage is a sub-problem for which God's Algorithm has been calculated. In its original form each stage is solved once optimally, resulting in a solution to the puzzle. As we have already seen, alternative optimal solutions to an early stage may lead to positions that give shorter solutions in the later stages. Therefore an obvious improvement is to do a search in each stage that visits every alternative optimal solution.

This adaptation of Thistlethwaite works well, and will result in somewhat shorter overall solutions. It is also quite fast since there are usually not that many alternative solutions in a stage, and they can be generated very quickly from the stage's God's Algorithm database. To be really effective however, we would also have to try non-optimal solutions in one or more stages, just like Kociemba's algorithm.

The difficulty with this approach lies in bounding the depth of the search in each phase. Suppose there are n phases, and that we were to naively copy the Kociemba approach exactly. Then phase n-1 keeps getting extended until it consumes phase n, and gives an optimal solution for those two phases combined. At this point phase n-2 is searched for another solution, and then phases n-1 and n-2 are again searched from scratch. This continues until eventually the phase n-2 search has extended so far as to consume phases n-1 and n, thus giving an optimal solution for the last three phases.
This clearly will not work very well. The algorithm spends all its time in the later phases, optimising small parts of the solution, without even trying alternative phase 1 solutions.

It is clearly necessary to limit the search depths of later phases if we ever want to extend the searches of earlier phases. The naive implementation above obviously fails, but there is a better way. First each phase is done optimally, as in the Thistlethwaite adaptation above. Once all those alternatives have been exhaustively searched, introduce one move of slackness which is to be used in any of the first n-1 phases. In other words, any one of those first n-1 phases is allowed to use one extra move, while the other n-2 phases remain optimal. Once all these possibilities have been searched, do the same with two moves of slackness, to be taken up by any one or two of the first n-1 phases. This continues, theoretically until the slackness plus the minimal length of the first phase equals the length of the best solution found. In practice, you would break off the algorithm before then when the solutions are good enough.

I have successfully applied this algorithm to Peter's Black hole, to find a solution of 150 moves in about 90 minutes. I also ran straightforward IDA* for about 24 hours until it exhaustively searched all sequences of up to 118 moves without success. It would take about 30 times as long to get through depth 120.