The Program

Excerpt from *The Panex Puzzle*, Mark Manasse, Danny Sleator, Victor K. Wei, and Nick Baxter.

> We have written a program (see source code) that finds the shortest sequence of moves that takes the puzzle from any position to any other position. Obviously the amount of time and space required by the program depends on the initial and final positions, as well as the available hardware (In 1983, the original program running on a VAX 11/750 with two megabytes (with up to six megabytes of virtual memory) was able to solve X_5 in about an hour, and X_6 in under a day; but it could not solve X_7 before running out of memory.)
>
> Imagine a graph in which each node corresponds to a position (or configuration, state) of the puzzle, and there is an edge connecting two nodes if there is a move that takes one position into the other. If only the top *k* tiles of each color are to be exchanged, then the graph needs only to contain a node for each configuration of *2k* tiles. (Moving pieces of size larger than *k* cannot help us solve the puzzle, therefore the graph need not contain nodes involving different positions of them.) This graph is undirected since the moves are reversible.
>
> A straightforward method of finding a way to get from an initial position to a final position is to generate the corresponding graph, and find the shortest path between the initial and final positions in this graph using Dijkstra's algorithm. The technique we used is just a refinement of this method.
>
> The first refinement is based on the observation that all edges have unit length. Dijkstra's algorithm then works as follows. At any given time we have reached and marked all nodes that are at a distance *d* or less from the initial position. We also maintain a list (called the *c-list*) of all those nodes that are at distance exactly *d* from the starting position. One phase of the algorithm has the effect of extending this distance to *d+1*. It works by probing all the nodes adjacent to nodes in the c-list. Those that are already marked are ignored, and those that are not marked are now marked, and placed in the new version of the c-list (called the *n-list*). These newly marked nodes are exactly those that are not at distance *d* or less from the start and for which there is a path of length *d+1* from the start. Therefore, the shortest path to these nodes is of length exactly *d+1*. This reaches all the nodes at distance *d+1* from the start, because a path to a node at distance *d+1* must go through a node at distance *d*. The process starts with only the initial node marked and in the c-list. The process terminates when the final node is marked.
>
> In order to reconstruct the shortest path, one additional data structure must be maintained. When we mark a node at distance *d+1*, we update a pointer in that node to the node at distance *d* that caused it to be marked. At the end we can reconstruct the shortest path by following these pointers from the final node to the initial node.
>
> One way to reduce the storage requirement is to do the computation in such a way that the full graph is never actually generated. The observation that makes this possible is the following: since the graph is undirected, it is the case that when we reach a node that has been reached before (a marked node in the above algorithm), that node is either on the c-list, or it was on the c-list at the previous phase. (We cannot otherwise reach a previously marked node, since such a path would have been traversed immediately after that node was first detected.) Thus, if we keep the old c-list around (we call it the *o-list*), then we can determine if the newly reached node was "marked" simply by determining if it is in the c-list or o-list. The o-list is effectively a boundary that forces the search to go in the right direction.
>
> It is necessary to make the test to determine if a node is in the c-list or o-list more efficient than simply scanning these lists. Hashing is a natural way to do this. We keep a hash table that contains

all the nodes of the o-list, c-list, and n-list. We also keep these nodes linked together to form the three lists. At the end of each phase the o-list is deleted from the hash table, the c-list becomes the o-list, and the n-list becomes the c-list. (We use separate chaining so that deletion from the table is efficient.) After hashing a node, we need to be able to tell which list that node is in. We do this by keeping a generation found in each node. If the nodes in the c-list are generation *g*, then those in the o-list are *g-1* and those in the n-list are *g+1*. (We actually only need to keep this modulo 3.)

A problem with the method of not keeping the whole graph is that it becomes difficult to reconstruct the solution. If time were not a consideration then we could run the algorithm until we found a path from start to finish and remember the node from which we reached the final one. This node is sure to be on a shortest path from start to finish. We then run the whole program again using the second to last node as our finishing node. The whole solution can be found by repeating this process *n* times, where there are *n* nodes on the shortest path.

A much more efficient way to deal with this difficulty is to search from start and finish simultaneously. The node where the two searches meet is sure to be on a shortest path. This procedure is then recursively applied to find the shortest path from the starting position to the middle position, and then again to find the shortest path from the middle to the final position. With this trick, the time to construct a complete solution is at most *log(n)* times the time required to find the middle position, where *n* is the number of moves on the shortest path.

Making the program search in both directions involves very small changes. We initialize the c-list to contain both the starting and finishing nodes, and we maintain a bit in each node indicating whether that node was found as a result of the search from the start or from the finish. The first common node is the midpoint (or one of a pair of midpoints).

One final trick to reduce the storage requirement takes advantage of the unique color and mirror symmetries of the puzzle. If *x* is a position, let *x'* be the position obtained when the left and right columns of *x* are swapped, and *x** is that obtained if the colors of all the pieces are swapped. Note that *x*'* and *x'** are the same. Consider the set of nodes of the c-list that have been reached from the starting position (at a particular moment during the running of the program). This set of positions is closed under the **'* operator, that is, if *x* is in this set, then so is *x*'*. This is true for the following reason. Consider a sequence of moves that takes us form the starting position to a position *x*. If we take every node in this sequence and apply the **'* operator, then we get a legal sequence of positions from the starting position to position *x*'*. (Note that the starting position is invariant under the **'* operator.) Since the set in question contains all positions reachable in exactly *d* steps from the starting position it must contain *x*'*. By only storing one of the two symmetrical positions we can reduce the memory requirement by a factor of two.

A similar redundancy occurs because the final position is the initial position with ' applied. This means that the set of nodes accessible from the starting position in *d* moves is the reflection of the set reachable in *d* moves from the final position. This means that we do not have to store the list of positions reachable from the final position in *d* moves, which reduces the storage requirement by another factor of two.

Here is how we incorporate these ideas into the program to save space. Initially the c-list contains just the starting position. When we generate a position (say *x*) adjacent to one in the c-list, we check if *x*, *x**, *x'*, or *x*'* are in the hash table. If none of them are, then we just insert *x* into the hash table (*x* and *x*'* are accessible from the starting position in *d+1* moves), and continue. If *x'* or *x** is in the hash table then *x* is the middle position of a solution; the program computes the number of moves (based on the generation numbers of *x* and the *x'* or *x** found, either *2d+1* or *2d+2*), and terminates.