CALIFORNIA STATE UNIVERSITY, NORTHRIDGE

GETTING TO EIGHTY CUBES: COMBINATORIAL SEARCH
HEURISTICS APPLIED TO N-CUBE INSTANT INSANITY™

A graduate project submitted in partial fulfillment of the requirements
For the degree of Master of Science in Computer Science

By

Chris Reeser

May 2015
The graduate project of Chris Reeser is approved:

_________________________________________  ____________________________
Jeff Wiegley, Ph.D.                         Date

_________________________________________  ____________________________
Adam Kaplan, Ph.D.                          Date

_________________________________________  ____________________________
John Noga, Ph.D., Chair                     Date

California State University, Northridge
Acknowledgements

If it were not for the encouragement and generous support of numerous faculty and friends none of this would have been possible. Thanks especially to John Dye, PhD., for challenging me to solve a 60-cube Instant Insanity problem in under an hour, and for taking the time to discuss my crazy efforts in attempting to do so. Credit goes to professors Gloria Melara, PhD., and Peter Gabrovski, PhD., for encouraging me to enroll in the graduate program in Computer Science at CSUN. Much appreciation is due Chris and Lisa Everett, Ian and Karen Kroop, Mark Lewis, Martin Parks, Joe and Rosemary Ragone, Brenda and Takashi Saito, and John Whitford for their generous support. Thanks also goes to Robyn McGinnis for providing a quiet table in the back of the restaurant to work on this project and for all the coffee. Recognition is well deserved by Bahar Hajiaghamohammadi, Gladys Garcia, and Joel Rarang, who gave of their enthusiasm, creativity, and expertise as software engineers, computer scientists, and lab partners.
# Table of Contents

Copyright .................................................. ii
Signature page ............................................. iii
Acknowledgements .......................................... iv
Abstract ..................................................... viii

1 Introduction .............................................. 1
   1.1 N-Cube Instant Insanity (I2N) .......................... 1
   1.2 Combinatorial Explosion ............................... 2

2 Applied Heuristics ...................................... 4
   2.1 Avenues of Attack ..................................... 4
      2.1.1 Reduction by Symmetry .......................... 4
      2.1.2 Reduction by Equivalence ....................... 4
      2.1.3 Reduction by Logic .............................. 6
      2.1.4 Reduction by Patterns ......................... 6
      2.1.5 Reduction by Elimination ...................... 6
   2.2 Combinatorial Search ................................. 7
      2.2.1 Constraint Satisfaction Problem (CSP) ......... 7
      2.2.2 Search Optimization ............................ 8

3 I2N Optimization ....................................... 11
   3.1 Data Structures ...................................... 11
      3.1.1 Typed Arrays .................................. 11
      3.1.2 Compressed Stack .............................. 11
      3.1.3 Indexing ....................................... 12
   3.2 Forward Checking .................................... 12
      3.2.1 Look Ahead to Avoid Dead Ends ............... 12
      3.2.2 Tabu Table to Avoid Repeated Work ........... 12
   3.3 Constraint Propagation ............................... 13
      3.3.1 Local Consistency .............................. 13
      3.3.2 Arc Consistency ............................... 13
   3.4 Variable Ordering .................................... 13
      3.4.1 Fail Early, Fail Often ........................ 14
      3.4.2 Branching Depth, Branching Factor ........... 14
      3.4.3 Lasts vs Spans Visualization ................. 14
      3.4.4 Global Optimum of N-Factorial Orderings .... 14
      3.4.5 Fitness Function ............................... 15
   3.5 Algorithm Selection .................................. 16
      3.5.1 Cost vs Benefit ................................. 17
6.1.2 Minimizing Movement Helps ........................................... 33

6.2 Overhead Costs ................................................................. 33
  6.2.1 Avoiding Work requires Work ........................................ 33
  6.2.2 Synergistic Heuristics .................................................... 33
  6.2.3 Non-overlapping Coverage ............................................. 34
  6.2.4 Order of Execution Matters .......................................... 34

6.3 Exponential Complexity Limits Scalability ............................. 34

A Data: Inputs, Solutions, Times ............................................ 35
  A.1 Completion times for 60, 64, 68, 72, 76, and 80-cube problems .... 35
  A.2 Data and time for one 80-cube problem ................................ 36
  A.3 Data, solutions, and times for one 100-cube problem .............. 37

References .............................................................................. 42
ABSTRACT

GETTING TO EIGHTY CUBES: COMBINATORIAL SEARCH HEURISTICS

APPLIED TO N-CUBE INSTANT INSANITY™

By

Chris Reeser

Master of Science in Computer Science

The natural generalization of the Instant Insanity puzzle to n cubes has been shown to be NP-complete. An algorithm to solve the problem therefore requires exponential time unless P=NP. This fact, however, does not mean that the intelligent use of combinatorial search techniques cannot dramatically improve performance over the most basic brute force attack. This paper describes the implementation and performance testing of a number of such techniques. These techniques allow a modest computer (circa 2015) to solve typical instances in time \((4/3)^n/10^8\) minutes for \(n < 80\).
Chapter 1

Introduction

The cube based puzzle Instant Insanity was formerly known as the Great Tantalizer, Katzenjammer, and Groceries. The puzzle was re-conceived by Franz Owen Armbruster as a device to teach combinatorial complexity. Marketed by Parker Brothers in 1967 as Instant Insanity, over 20 million of the puzzles were sold [2].

![Two sides of an Instant Insanity™ puzzle.](image)

The deceptively simple puzzle consists of four-cubes of which each face is solidly colored by one of four colors. A valid solution is to stack the four cubes such that each of the four sides of the stack display all four colors. Of the 41,472, or $(1/8)24^4$, possible alignments, only one satisfies the criteria for a correct solution.

1.1 N-Cube Instant Insanity (I2N)

An n-cube generalization of the four cube Instant Insanity problem was shown in 1978, by Roberson and Munro, to be NP-complete[2]. As such, it is easy to verify a solution, yet finding all possible solutions to a particular puzzle, or proving none exist, becomes exponentially more difficult as the number of cubes increases. Since each cube added increases the number of possible arrangements by a factor of 24, the explosive combinatorial growth quickly makes even puzzles of modest length appear to be intractable.

The generalization to n cubes is straightforward. There are n cubes, n colors, and each side of each cube is a single color. A solution is a way of stacking the cubes so that for each of the four sides of the stack (front, back, right, left) no color is repeated. Throughout the remainder of this paper, N-Cube Instant Insanity will most often be referred to simply as I2N.

With regard to the first cube, each additional cube can be oriented 24 ways. For convenience, integers are used for labels in place of colors. Consider a first cube, c1, with a top, bottom, left, right, front and back, labeled 0, 1, 2, 3, 4, 5, respectively. A second cube, c2, with faces labeled 6, 7, 8, 9, 10, and 11, is stacked neatly on top of the first cube. Now given that the top of c1 is 0, and the front of c1 is 4, c2 could be oriented so as to have any of its 6 faces touching c1’s face 0, and be rotated about its vertical axis so that any of one of those 4 sides is showing in front, directly above c1’s face 4. A bit of simple JSON notation can help keep things straight. Suppose we describe a cube as an array of pairs of opposite
Table 1.1: Setting each of c2’s 6 faces against c1’s top-0, front-4, with 4 rotations on vertical axis.

1.2 Combinatorial Explosion

Before tackling this problem it is worth considering how large a problem might possibly be solved. Assuming NP is not P, an exponential complexity is expected. Because even a mere 20-cube puzzle has more than $5 \times 10^{26}$ possible arrangements, checking them all one-by-one would take too long. An 80-cube puzzle has over $3 \times 10^{109}$ possible arrangements.

Use of modern computers is assumed, but their limits circa 2015 are considered. It has been estimated that in 2007, all the general-purpose computers in the world computed $6.4e18$ instructions per second [8]. That amounts to $2.02e26$ instructions for the year, 2007. If that capacity doubled every year from 2007 to 2015, then in 2015, the world capacity would be $2.56e2 \times 2.02e26 = 5.17e28$ instructions. At that rate, using the entire world’s computing power, it would still take over 3 years to find all solutions to a single 20-cube problem using a brute force method.

Brute force simply by its name doesn’t sound like a very efficient way of cracking this problem. However, a brute force algorithm does have the advantage that it checks each possibility only once. This is in distinction to a typical human initial attempt which quite likely would begin by stacking the cubes rather arbitrarily, checking whether a solution was achieved, possibly trying small modifications (e.g. reorienting one or two cubes), knocking
Table 1.2: Paths to check for various branching factors as $n$ increases.

<table>
<thead>
<tr>
<th>$n$</th>
<th>4</th>
<th>10</th>
<th>20</th>
<th>40</th>
<th>60</th>
<th>80</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(1/8)2^n$</td>
<td>41,472</td>
<td>7,925,422,620,672</td>
<td>5.025e26</td>
<td>2.020e54</td>
<td>8.121e81</td>
<td>3.264e109</td>
</tr>
<tr>
<td>$(2/3)3^n$</td>
<td>54</td>
<td>39,366</td>
<td>2,324,522,934</td>
<td>8.105e18</td>
<td>2.826e28</td>
<td>9.854e37</td>
</tr>
<tr>
<td>$(2/3)1.5^n$</td>
<td>4</td>
<td>58</td>
<td>2,217</td>
<td>7.372e6</td>
<td>2.451e10</td>
<td>8.151e13</td>
</tr>
<tr>
<td>$(2/3)1.3^n$</td>
<td>3</td>
<td>10</td>
<td>127</td>
<td>2.408e4</td>
<td>4.576e6</td>
<td>8.697e8</td>
</tr>
</tbody>
</table>

the stack down, and repeating. It is clear that some means of quickly and easily tracking
time results, and detecting and eliminating a major portion of unfruitful arrangements is needed. Fortunately, combinatorial search algorithms are designed to address this very problem.
Heuristics are often considered to be rules of thumb, concepts that help solve a particular kind of problem[5]. Here they derive from insights into I2N in particular and combinatorial search in general.

If it were possible to incorporate such heuristics into an algorithm so that it could find all the solutions to an 80-cube puzzle in a manner that required checking only \((2/3) (4/3)^{80}\) potential solutions, then perhaps a typical desktop computer could do so in a matter of a few hours.

2.1 Avenues of Attack

2.1.1 Reduction by Symmetry

First, to eliminate an obvious symmetry, consider ways the entire solution could be oriented with regard to three axes. Since for any solution the whole stack could easily be rotated so that any of its four long sides becomes the front side, it will not count as four separate solutions, only one. Also because every cube in the stack could be rotated 180 degrees such that the stack’s two end cubes (top and bottom) expose their hidden faces and hide the formerly exposed stack bottom and stack top, and such that all other hidden faces remain hidden, rather than counting as two separate solutions, these two trivial variations are treated as one. So by eliminating this symmetry we reduce the total by seven-eighths. Otherwise, each cube relative to the first cube can be oriented 24 ways.

Second to avoid over-counting solutions, different orderings of the cubes themselves will not be considered as separate solutions. In fact we have already been making this assumption. Otherwise the \(24^n\) would actually be \(n! 24^n\). Therefore, a given puzzle has a single implied ordering of the cubes.

Another instance of symmetry occurs whenever a pair of opposite sides has the same color, or whenever two or three pairs on a single cube have the same color(s). Because multiple orientations of the cube could lead to indistinguishable solutions, there is no point in counting them more than once.

2.1.2 Reduction by Equivalence

Sometimes the essence of a difficult problem can be expressed as a simpler problem and having solved the simpler problem, enough information is gained to then more easily arrive at a solution to the original difficult problem. Fortunately, I2N has this characteristic.

By looking at the problem only slightly differently, we can see that it can be broken down into first finding a run of opposite side pairs where each color occurs exactly twice. Another way of saying this is that the conditions for a solution are relaxed to have no requirements about the left and right sides at all and only require the front and back together
to have exactly two of each color instead of each of the 4 sides having exactly 1. This can be done with a simple brute force attack on the $3^n$ ways to choose the front/back pair for each cube.

The number of such partial solutions $p$ is at most $3^n$ each taking time $n$. While this can occur (e.g. in the case where each cube has all 6 sides the same color), cases of this type are easy to identify using ideas from other sections in this chapter. For uncontrived instances, it is extremely rare for $p > n$. For each partial solution found, the front and back faces can be fixed and the process can be repeated for the left and right sides. Since the front and back are fixed, there are $2^n$ cases to check. As before, the number of such partial solutions $q$ which is certainly no more than $p$ could be as large as $2^n$, it is exceedingly rare and easily detected. Once two complimentary runs (which makes for a complete solution) are found, a final step is required to determine front vs back and right vs left. This can be done in linear time. So this technique has a time complexity no worse than $O(n3^n + pn2^n + pqn)$. Given the limits on $p$ and $q$ noted above, this technique could require time proportional to $n6^n$, but almost always completes in time proportional to $3^n$.

One might argue that this relaxation process could be carried further by first looking for partial runs of the pairs - finding just one face on each cube such that exactly one of every color is counted. But this will in fact take even more work because it generates too many partial solutions and must select from $6^n$ permutations.

On the other hand, attempting to verify all four sides at once doubles the amount of work at each step with little gain.

The middle ground here proves to be the best and is related to earlier work on a graph-based approach to Instant Insanity[6].

Figure 2.1: Graph based solution[6].

Figure 2.1 shows a graph of 4-cube Instant Insanity, where nodes represent colors and edges the cubes on which two nodes form a pair of sides opposite one another.

However, as one can imagine, this can get a bit messy with $n > 4$. The same technique sans drawing is shown in table 2.1 using highlighted JSON notation. The work of untangling such graphs so as to find two separate sub-graphs, each having one node for every color can be represented as two paths through a search tree where each path yields a color.
<table>
<thead>
<tr>
<th>Cubes</th>
<th>Opposite Side Pairs</th>
<th>Pair 0</th>
<th>Pair 1</th>
<th>Pair 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cube 0</td>
<td>[0,1]</td>
<td>[1,3]</td>
<td>[2,2]</td>
<td></td>
</tr>
<tr>
<td>Cube 1</td>
<td>[0,1]</td>
<td>[3,3]</td>
<td>[2,3]</td>
<td></td>
</tr>
<tr>
<td>Cube 2</td>
<td>[0,2]</td>
<td>[1,3]</td>
<td>[0,3]</td>
<td></td>
</tr>
<tr>
<td>Cube 3</td>
<td>[0,1]</td>
<td>[0,2]</td>
<td>[2,3]</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.1: Two paths through 4-Cube Instant Insanity.

count of two for each color and no sub-path is duplicated.

The path highlighted in pink represents a partial solution as a run of opposite sides, say front-back, and the light blue path represents the complementary partial solution, which together make a complete solution. A search space tree representing the problem specified this way has an initial branching factor of 3, leading to a search space of $3^n$ again assuming that only a small number of initial paths are feasible.

### 2.1.3 Reduction by Logic

Since every partial solution requires a complementary partial solution, only two of the first cube’s pairs need to be considered while searching for partial solutions as the complementary partial solution will surface if it exists on the $2^n$ branching complementary search space. Thus reducing the work required to only $(2/3)3^n$.

Now we could use a brute force algorithm like that shown in listing 1, but there are several more improvements which can be made.

### 2.1.4 Reduction by Patterns

There are certain easily identifiable patterns which can yield an immediate solution. For instance, if any particular color occurs less than four times there can be no solution. If the number of colors is less than the number of cubes, the puzzle is also invalid and cannot be solved. Several such patterns may be found which yield a null result and require little effort. Others are more complex and take a bit of work.

If it can be discovered that a puzzle is in fact the amalgamation of two or more separate smaller puzzles, such that each smaller puzzle shares no color in common with the others, then the smaller puzzles can be solved separately much faster than would be suggested by the total number of cubes.

Because this analysis requires relatively little work it will be referred to in this paper as low-hanging fruit.

### 2.1.5 Reduction by Elimination

Although brute force is a systematic way to guarantee every possible solution is found, it is not very efficient. One obvious improvement is to stop checking subsequent cubes as soon as any color has occurred more than twice. No matter what follows, the permutation will fail to be part of a solution.
Even the ancients understood at least one improvement over brute force search. In Greek mythology, Ariadne was a weaver who found a way out of the labyrinth by unwinding a trail of thread from her spindle. This enabled her both to find her way back from a dead-end and to avoid repeating a path already tried. This concept is the basis of backtracking search, which is the fundamental form of informed or heuristic search.

Figure 2.2: Backtracking with Pruning[7]

One can think of the collection of all possible permutations of pairs of opposite sides of the cubes in I2N organized as a tree where each node, representing a cube, branches three ways, representing the three possible choices for a pair of opposite sides. Pruning refers to the elimination of an entire sub-tree whenever it is discovered that no solution is possible on any path beyond a given node.

Pruning is mainly how Combinatorial Search can greatly reduce the amount of work in finding solutions to I2N as well as many other difficult problems.

2.2 Combinatorial Search

Typically concerned with NP-hard problems where the search space is exponential in regard to the input, combinatorial search algorithms attempt to use heuristics to greatly reduce the unproductive portions of the search search space or otherwise effectively explore the characteristically large domain.

2.2.1 Constraint Satisfaction Problem (CSP)

Finding the solution to some combinatorial search problems requires finding an optimal path where the number of nodes may be unbounded. However, there is another type of combinatorial search problem where the number of nodes, or variables, is fixed, and a solution entails satisfying a number of constraints such that all variables comply with rules about local consistency. These are known as Constraint Satisfaction Problems or CSPs.
I2N can easily be seen as a CSP. Many other problems that can be stated as a CSP include: Sudoku, N-Queens, planning and allocation, map coloring, boolean satisfiability problems (SAT), and a wide variety of logic puzzles.

### 2.2.1.1 Formal Definition

Because many concepts related to constraint satisfaction are integral to the application of the heuristics applied to I2N in this paper, a formal definition of a Constraint Satisfaction Problem seems in order.

A CSP specifies a triple \( \{X, D, C\} \), where \( X \) is a set of variables \( \{X_1, ..., X_n\} \), \( D \) is a set of respective domain values \( \{D_1, ..., D_n\} \), and \( C \) is a set of constraints \( \{C_1, ..., C_m\} \).

Each variable \( X_i \) can take on values in the nonempty domain \( D_i \). Every constraint \( C_j \in C \) is a pair \( (t_j, R_j) \), where \( t_j \subseteq X \) is a subset of \( k \) variables and \( R_j \) is a \( k \)-ary relation on the corresponding subset of domains \( D_j \). An evaluation of the variables is a function from a subset of variables to a particular set of values in the corresponding subset of domains. An evaluation \( v \) satisfies a constraint \( (t_j, R_j) \) if the values assigned to the variables \( t_j \) satisfies the relation \( R_j \).

An evaluation is consistent if it does not violate any of the constraints. An evaluation is complete if it includes all variables. An evaluation is a solution if it is consistent and complete[8].

### 2.2.1.2 Constraints on I2N

For I2N, the set of variables is the set of cubes, the set of domains is made up of the pairs of opposite side colors for each cube, and the set of constraints restricts the values (pairs) at each cube to those that satisfy certain rules such as: Every color must have a count of exactly two for a partial solution. A complete solution is a set of two compatible partial solutions such that no two pairs are used twice on a single cube.

From these simple rules, more elaborate conditions can be applied to more efficiently enforce the above rules. Such is the basis of heuristics used to optimize the search for solutions to the CSP.

### 2.2.2 Search Optimization

Given that combinatorial search normally deals with especially large search spaces it is critical that the search proceeds in as efficient manner as possible. Various techniques have been found to improve search performance to varying degrees. A number of techniques apply to improving the performance of I2N, some general and some specific.
function verify(puzzle, path) {
    zero(count) // set array count to n zeroes
    for (i = 0; i < n; i++) {
        // the pair of opposite sides to use for this cube
        pi = path[i]
        // front or left or top, say...
        color0 = puzzle[i][pi][0];
        // back or right or bottom
        color1 = puzzle[i][pi][1];
        count[color0]++
        count[color1]++
        // Improvement: could exit loop early if either count > 2
    }
    for (i = 0; i < n; i++) {
        // check that every color shows exactly twice
        if (count[i] !== 2) {
            // no need to keep checking
            return false
        }
    }
    return true // everything looks ok
}

// nextPath iterates through all possible paths once
while (path = nextPath()) {
    if (verify(puzzle, path)) {
        // getCopartial returns co-path if it exists
        if (path2 = getCopartial(puzzle, path)) {
            // add to collection of solutions
            solution.push([path, path2])
        }
    }
}

Listing 1: Brute Force
state = [];
i = 0;

while (i < len) {
    path[i] = nextPath(path[i], state);
    if (path[i] > 2) { // backtracking
        do {
            path[i--] = 0;
            path[i]++; // cascade backward
        } while (path[i] > 2);
    } else { // not backtracking
        // checks constraints
        result = constrain(len, i, path, state);
        // if any constraint fails, skip all subtrees
        // below this branch, move on to the next branch
        if (result & FAIL) {
            // current node in path leads to violation of constraints
            path[i]++;
            // try next branch, if no more branches,
            // backtrack to first untried ancestor branch
        } else if (result & PASS) {
            // i = len - 1, path is allowed for entire length
            accept(len, path, output);
            // processes solution candidates found
            path[i]++;
        } else { // result & CONTINUE
            i++;
        }
    }
}
} // end while-loop
Chapter 3  
I2N Optimization

A number of techniques exist which can improve the average performance of an exhaustive search for solutions to a set of I2N problems. All are used in the author’s experimental implementation, I2NX.

3.1 Data Structures

Arrays are used throughout I2NX as a convenient way to organize related data, to allow direct indexing into a set of values, and to pass a collection by reference. A stack is used to keep a snapshot of previously calculated values on hand so that upon backtracking, those values need not be recalculated, only updated with a few newly changed values. A queue is used to maintain the pool of problem fragments distributed to multiple concurrent web workers.

3.1.1 Typed Arrays

Because ordinary JavaScript arrays are dynamically typed, they need to allow for various value types such as integers, floats, and strings. This flexibility comes at the price of greater complexity and diminished performance unless the JavaScript engine or just-in-time compiler is able to deduce a simpler and consistent usage, as is sometimes the case with Chrome’s V8 engine. However, Typed Arrays are views into a sequential memory buffer. They can be treated as statically typed and often give much better performance. Multiple views can be mapped onto the same buffer giving some additional flexibility not readily available with the ordinary arrays.

I2NX uses Typed Arrays for its stack of tabu tables and color counts. Compared to previous use of ordinary arrays, performance more than doubled.

3.1.2 Compressed Stack

Moving data from memory to registers of the CPU takes time. This is mitigated somewhat by the use of memory cache as determined by architecture of the CPU circuitry. This motivates the application designer to seek ways to minimize the swapping of data between main memory and cache by limiting the application’s memory usage. One way I2NX reduces memory requirements is by compressing the data it frequently uses and by passing references to data buffers rather than copying.

Using an unsigned 8-bit integer typed array for the main I2NX stack allowed packing both the tabu table and color count into the same array. (Currently, no 4-bit integer type is available). Using the lower 3-bits for a color count that ranged in value from 0 to at most 6 leaves the high bits available for use by the three tabu table bit flags. Even though the index for color count is by color and the index for tabu table is by cube, this still works because I2N requires the number of colors to be equal to the number of cubes. Compared to using
two separate arrays, performance is nearly twice as good. Even better, by piggy-backing
onto the same array, the benefit of tabu tables came at practically no additional cost, making
possible even greater performance gains by way of constraint propagation.

3.1.3 Indexing

Besides the advantages of being able to reference values by means of some integer
result of a calculation, indexes can themselves be manipulated while serving as a proxy for
the values they represent. In the case of I2NX’s variable ordering app, the many potential
orderings generated and mutated by the genetic algorithm are simply arrays of indexes.
The values pointed to by the indexes are not moved from the original position in the array
where they are stored. Rather, the indexes themselves are rearranged. Access to the values
then takes place via the reordered array of indexes. Only once the best ordering has been
determined, is the original array of values rewritten.

3.2 Forward Checking

Rather than limiting branching decisions to the accumulated state up to and including
the current node, consideration is given to information that may be obtained from nodes
lower in the search tree. Often the limited effort required to obtain this additional informa-
tion saves considerable work resulting in a net performance gain.

3.2.1 Look Ahead to Avoid Dead Ends

In I2N the constraint that the count of each color in a partial solution must be exactly
two leads to the consideration of whether any subsequent cubes can satisfy the two-count
of a color not yet having reached a two-count. By looking only at subsequent cubes hav-
ing that color, some of which may be several nodes away, it may be discovered that it is
impossible to obtain the required two-count without causing some other color to exceed its
two-count. Without having to perform all the work in reaching that point in the search, it
can be immediately determined that the current branching decision under consideration is
destined to fail.

3.2.2 Tabu Table to Avoid Repeated Work

Keeping track of which cube’s opposite color pairs are in conflict with pairs forming
the partial path to the current node can readily be managed with a structure known as a
Tabu Table. For I2NX, this is kept in the high bits of the stack so that on backtracking no
recalculation must be done other than for the arrived at cube. Whenever it is discovered
that a subsequent cube has a pair that would create a constraint violation if it were included
in the path, the tabu table is marked for that cube for that pair. Since there are three pairs
for each cube a bit flag for each pair can be set or cleared accordingly. Thus, only three bits
are needed to represent the availability of pairs for any single cube. If all three bits become
set for any cube it is clear that backtracking must occur.
3.3 Constraint Propagation

Various constraint propagation techniques might be used to make a CSP easier to solve by enforcing some form of local consistency. Sometimes the original problem is changed into an equivalent problem that can even prove satisfiability or unsatisfiability as a result of the local consistency itself. Otherwise it may improve the effectiveness of other heuristics such as backtracking.

3.3.1 Local Consistency

In constraint satisfaction problems, local consistency refers to the logical consistency of subsets of constraints or variables. A consistent variable assignment is one that can be consistently extended to another variable. Two types of local consistency are used in I2N. Directional consistency limits the domain of a subsequent cube having some color in common with the current cube’s pair of opposite sides. Relational consistency extends this to all subsequent cubes having either color of the pair with regard to the constraint that all colors of a partial solution have a color count of exactly two.

3.3.2 Arc Consistency

If a variable is not arc consistent with another one, it can be made consistent by removing some values from its domain, since they would not be part of a valid solution anyway. To do this for all pairs of variables, the process may have to be iterated a few times because removing values from the domain of one variable may affect its arc consistency with other variables. Eventually, the propagation of constraints in this manner can make the whole problem arc consistent.[8]

Reducing domain of variables, which is to say the available pairs at each cube, arc consistency in I2NX is maintained by keeping a count of each color exposed and marking the tabu table for subsequent cubes so that no pair remains unmarked that could result in a color count exceeding or falling short of two.

3.4 Variable Ordering

There are two main approaches to variable ordering. Dynamic ordering changes the ordering of variables while the search algorithm runs. This allows the ordering to adapt to changing conditions during the search and can result in the greatest reduction in branching. However it may consume more time than it saves in the bookkeeping required to manage the continuously changing variable order. Static ordering analyzes the variables prior to running the search algorithm and makes certain statistical assumptions about the ordering which would best reduce branching throughout the subsequent search. While this avoids costly search overhead, static ordering attempts to find the globally optimal ordering in n factorial possible orderings for a particular problem. Furthermore, it may make one part of the search more efficient at the cost of making another part less so. A possible hybrid approach may be to generate multiple static orderings for different partitions of the problem, each tuned to the particular characteristics of the partition.
3.4.1 Fail Early, Fail Often

Pruning away areas sure to have no solution early in the sequence of cubes saves later work and keeping the search tree narrow throughout reduces average branching factor. Because pruning can be so effective at reducing subsequent search effort, it may seem that doing so as much as possible as early as possible is the best strategy. However, this often proves to be at odds with the effectiveness of pruning throughout the search tree to keep it narrow. Pruning early but balanced with pruning throughout seems to give the best performance over various I2N problems. A similar finding is reported by Smith and Grant.

“The fail-first principle states that ‘To succeed, try first where you are most likely to fail.’ For constraint satisfaction problems (CSPs), Haralick and Elliott realized this principle by minimizing branch depth. Smith and Grant hypothesized that if failing first is a good thing, then more of it should be better. They devised a range of variable ordering heuristics, each in turn trying harder to fail first. They showed that trying harder to fail-first does not guarantee a reduction in search effort[1].”

3.4.2 Branching Depth, Branching Factor

Branching depth in I2N can be limited by minimizing span of cubes having a given color, whereas branching factor can be limited by maximizing elimination of variable domain values which happens as color last occurrences are approached. Maintaining arc consistency in I2N seems to make up for some of the loss of effectiveness which can occur in trying to balance reduced spans and lasts. Apparently others have found this to be the case in working with certain other CSPs.

“For variable ordering there does not appear to be a simple mapping between minimization of branch depth and search efficiency. We show that this is due to the fact that minimizing branch depth can be associated with an increase in the average branching factor, and the latter effect can be large enough to impair search overall. This occurs with forward checking, but if we use maintained arc-consistency (MAC) then this trade off is mitigated, so that minimizing branch length correlates with reduced search effort. We suggest that the radical fail-first principle can be replaced with an alternative failure principle that takes both the branching and the branch depth factors into account[1].”

3.4.3 Lasts vs Spans Visualization

A visualization showing colors along the y-axis and cubes along the x-axis provides a more intuitive sense of variable ordering for I2N. By coloring in the cubes between the first and last occurrences of a color, it is easy to see the span which affects the branching depth and potential backtracking. Sorting the colors by last occurrence helps accent the effect of a fitness function which biases toward a lower branching factor.

3.4.4 Global Optimum of N-Factorial Orderings

I2NX uses a genetic algorithm to quickly arrive at a beneficial ordering. Guaranteeing a globally optimal ordering among the n! possible orderings seems especially impracti-
cal considering the search space is at worst $3^n$. However because a really good ordering can reduce the search effort by factors of 10, 20, even 100 or more, over orderings with higher entropy, spending some limited time producing a good ordering nearly always proves worthwhile. However, there remains a trade-off between the rate of convergence on a good ordering and the improvement seen by extending the search. In I2NX, the rate of convergence is tuned to stop before the additional time spent exceeds the reduction in search time offered by a slightly better ordering. On problem lengths greater than 64, this tends to be many factors less than the search time.

3.4.5 Fitness Function

Genetic algorithms typically have a way of scoring the fitness of individuals which, in turn, may influence the next generation. As one generation of fit individuals leads to an even more fit generation, individuals of successive generations tend to score better, up to a point. If only the best individuals are allowed to contribute, the algorithm is said to be heavily biased toward elitism. This may lead to a rapid convergence on a less than globally optimal solution. One way to escape becoming stuck with local optima is to include some members which score poorly, thereby maintaining greater variation. Another way is to introduce some random mutations from time to time. Yet another way is to introduce a randomly generated individual into the gene pool. The degree of variation is used to control the rate of convergence and thereby, hopefully, to find an ordering which scores better than others and ultimately leads to a much faster search.

The usefulness of the genetic algorithm for I2N lies in its flexibility to produce orderings that score well according to some fitness function. Because orderings that minimize the span of colors among cubes and distribute the last occurrence of colors as early as possible yield quicker average search times, these measures become the basis of scoring in the
I2NX fitness function. Listing 3 shows one example of weightings assigned in I2NX. A lower score leads to a better fitness. A visualization of the results is seen in figure 3.1.

```plaintext
function score (s1, s2, s3, s4) {
  // s1 is the sum of (lasts)
  // cube positions where each color occurs last

  // s2 is the sum of (spans)
  // the average of cube positions
  // where a color occurs next to last with
  // where that color occurs next to next to last
  // less the position of the first occurrence
  // of the respective color

  // s3 is the sum of cube positions
  // where each color occurs next to next to last

  // s4 is the sum of cube positions
  // where each color occurs next to last

  // w[0], w[1], w[2], w[3] are weighting factors

  // n is the number of cubes in the problem

  var nn = n*n, w = [4/3, 6/5, 2/3, 3/4];

  return ((s1*s1/nn)*w[0] + s2*w[1] + s3*w[2] + s4*w[3])/2/nn;
}
```

Listing 3: Fitness Function

### 3.5 Algorithm Selection

By definition, solutions to N-cube Instant Insanity must provide exactly one of each of n colors per side of a stack of cubes. I2NX treats this problem as a CSP, searching for all consistent and complete solutions by means of a backtracking algorithm. To improve performance various heuristics are applied to derive additional constraints that benefit from forward checking and arc consistency. Those constraints are implemented as a collection of rules applied conditionally at each node in the search space, resulting in a decision to backtrack (eliminating another sub-tree) or to continue along the current path forward toward an acceptable state.

Selection of which rules, or algorithms, to consider at any point within the search determines to a great extent the performance of the search. Even the ordering of the rules can make a difference in performance.
3.5.1 Cost vs Benefit

Besides the requirement to provide a correct and complete set of solutions, if any exist, an important goal for I2NX is to do so as quickly as possible for a collection of many randomly generated problems. Optimizing for just one problem is not enough, particularly if it means that search for other problems is made much slower. Therefore, each rule used to improve the performance of I2N search must on average more than pay for itself. It is possible that a rule requires more time consumed in overhead than it contributes to reduced search time. In this way some rules are more effective than others. One way to rate a rule is to consider its benefit to cost ratio. Rules with a ratio greater than one are those which offer a net gain.

3.5.2 Synthesis

Sometimes the overhead of one rule is mitigated by its pairing with another rule. Together they provide a much greater net gain in performance, while alone, each contributes only a modest gain or even a net loss. In I2NX, packing tabu table onto the same stack with color counts leads to nearly a tripling of performance, whereas using a separate array reduces performance by half. Because overhead is conserved via reuse of an existing data structure, the benefits of several rules are achieved without additional cost.

3.5.3 Ordering

Typically rules with the greatest benefit to cost ratio are applied first. Of the conditions that satisfy these rules, further testing is done by rules with a lower ratio. A familiar case where this is done is by the Sieve of Eratosthenes. Multiples of lower valued primes are removed first before those of increasingly higher primes. Likewise, ordering of rules in I2NX results in demonstrably better performance.

3.5.4 Exclusion

The contribution a particular rule may make to performance might be dependent on the node at which it is used. One way I2N reduces some rules’ overhead is simply to exclude certain rules at nodes where they contribute little. While this could be determined dynamically during the search, that dynamism itself introduces a certain overhead. I2NX analyzes each problem in advance, statically determining which nodes will use which rules.

3.5.5 Alternation

One set or ordering of rules might also have a different degree of effectiveness at a different node. Providing alternate orderings depending upon the node’s characteristics can sometimes improve performance. I2NX determines this statically in advance of search.

3.5.6 Tipping Point

In combination with other heuristics, an otherwise expensive rule may lead not only to a net gain for itself, but for many others as well. In this case its inclusion creates a ripple effect increasing the effectiveness of existing rules and even makes otherwise low
performing rules useful. In I2N, tuning the fitness function used by the variable ordering app to account for the next-to-next-to-last color occurrence forward checking rule improved not only the performance of that rule, but also allowed additional arc consistency rules to be used effectively.

3.6 Avoiding Unnecessary Work

It has been said that “there is no code faster than no code.”[3]. Although one might reasonably expect as well that no code does very little, sometimes that is exactly what we want. Some of the performance improvements to I2NX involved simply finding and removing code that proved unnecessary. A classic example in JavaScript is a for loop that is bounded by the length of an array. Given an array a, it is better to assign its length, a.length, to an integer variable, say alen, than to look up a’s length on each iteration of the loop. Given that these minor improvements are in code that may be run hundreds of billions of times, and that for loops are used frequently in I2NX, the little improvements can add up to a noticeable amount of time saved.

A few other simple techniques can also save a lot of work such as: Exiting loops as early as possible, passing by reference, reusing existing arrays rather than creating them anew, avoiding incremental size increases, declaring all variables rather than allowing them to be created dynamically, minimizing function calls, holding references to objects that will be used again rather than allowing them to be garbage collected, avoiding push or pop operations, ordering comparisons to minimize later compares, and reusing the results of earlier calculations wherever possible rather than recalculating them.

3.6.1 Moving Work out of Main Loop

As much as it may be possible to reduce the number of iterations of the main search loop by application of the various techniques discussed, it is also important to keep as much work as possible out of the main loop. Minimizing loops is good, but so is minimizing the work done the potentially trillions or quadrillions of times around the search loop because these things add up.

While there could be some additional adaptability gained by using dynamic ordering of rules or variables, doing so requires some bookkeeping that occurs on every loop iteration. I2NX avoids this by moving as much of the analysis and optimization process to the initialization stage, setting up the rules, orderings, indexes, and tables prior to the first iteration of the search loop.

3.6.2 Precalculating Tables, Indexes, and Rule Sets

Some of the tables I2NX calculates at initialization are: The positions of each color indexed by color, the cubes having a last, next-to-last, or next-to-next-to-last occurrence of a color, the colors found on the opposite side of a cube from a given color, and the location of any color occurring more than once on a cube. Tables of indexes to other pertinent information are created as well. Rules are established for each node based on relevancy. The tables used by those rules may be truncated to only relevant data.
3.6.3 Finite State Machine

Rather than recalculate the next available path segment on each iteration of the main loop, the limited choices can be embedded into a lookup table serving as a finite state machine (FSM). By passing in the current path segment along with the tabu state for a given cube, the FSM returns the next path segment. Because all calculations needed to create the FSM are done only once during initialization, repeated work is avoided.

3.7 Low Hanging Fruit

Before initiating a potentially protracted search, it is advantageous to consider certain characteristics of a problem which would immediately make the search simpler or even unnecessary. One obvious consideration is whether the problem even conforms to the rule that each color appear on each of four sides of the stack. If some color occurs less than four times within the problem, no amount of searching will find a solution. It takes very little work to identify this situation and many others like it. Such conditions that afford much with little effort are like low hanging fruit, easy to pick.

3.7.1 Minimum Requirements

Up front analysis of a problem can easily and quickly detect the conditions such as the number of cubes not matching the number of colors, some color having less than four occurrences, labeling which is non sequential or not beginning at zero, non conforming JSON, or malformed cubes not consisting of three pairs of colors.

3.7.2 Connectedness

Another condition that can dramatically affect a search is one where a problem is given of a certain size n, but analysis shows that it is actually two or more smaller problems merely presented as one. By performing a simple color graph analysis, it can be quickly determined whether or not the problem represents a connected or disconnected graph. Suppose a problem is given where a subset of the cubes has only colors 0 through k, and the remaining cubes have only colors k+1 through n-1. Solving the two problems is nearly always much faster than would be expected for a single problem of length n.

3.7.3 Exploitable Patterns

Six occurrences of each color are found on each of the problems benchmarked in I2NX for a couple of reasons. First, because this condition tends to create problems that are harder to solve than when less than six of some color(s) and more than six of some other color(s) are allowed. Second, because it removes some variation that allows for a more informative comparison between versions of I2NX. For problems allowing other than six of each color, certain additional exploits are available, such as weighting more heavily orderings where the cubes with less than six colors are moved toward the beginning of the search sequence thus leading to a narrowing at the top of the search tree.

In an earlier chapter, a rationale was given for only needing to search paths using two of the three pairs on the first cube. There is an easily identifiable pattern that allows this to be
halved. If some cube has two pairs of opposite sides duplicated, then using this cube as the first cube leads to a condition where only one of the duplicated pairs must be searched. To illustrate, if the side pairs of each cube are chosen uniformly and independently then there is a one in \((n^2+n)/2\) chance of two given side pairs matching. Assuming that \(n\) is large, the likelihood of having a cube with a repeated side pair is approximately 1/6. In testing, this pattern occurred frequently enough to justify the little work required to identify it.

There are a few other patterns that occur less frequently, but can be easily identified such as where two or three cubes have all occurrences of some color and no orientation of those cubes allows for a solution. More complex patterns may take more work and care must be given that the average cost does not exceed the average benefit.

3.8 Parallelism

Most of the optimization discussed in this paper revolves around algorithmic efficiency. But given that recent hardware provides multiple-core processors, it is practical to maximize use of the available processing capacity. Dividing the problem into pieces that can be searched simultaneously can reduce the elapsed time from start to completion. I2NX uses web workers not only to keep long running searches from crashing the browser and to allow for a fluid user interface, but also to leverage otherwise idle CPU capacity.

3.8.1 Partitioned Search

One way to partition the search space of I2N is by dividing it among a number of sub-trees. A path through the search space is specified by a sequence of branching decisions made at each node. For instance, a sub-path prefix of \([1,0,2,1]\) would indicate that the second pair of opposite sides was chosen at cube0, the first pair at cube1, followed by the third pair on cube2 and the second pair on cube3. A permutation of all such prefixes of length 4 would partition the entire search space into \(3^4\), or 81, sub-trees. Each of these sub-trees, referred to in I2NX as fragments or shards, could be searched independently of the others, even in parallel, and the results combined in some subsequent step.

3.8.2 Replicated Workers

I2NX distributes such partitions among concurrent web workers. This form of parallel computing is referred by some as Replicated Workers as both the code and the data is copied to each worker process[4]. What differs is the partition specification provided via the path prefix in I2NX. Each worker operates independently of the others. Whenever a worker completes searching its assigned partition it reports its results back to the master process and receives another path prefix if any remain unassigned. Once all partitions have been searched and reported, the master process aggregates the results and displays the total elapsed time.

3.8.3 Communication

Combinatorial search is typically computationally intense relative to I/O. Because there is so little communication overhead in I2NX, there is a high potential for parallel processing
to reduce elapsed time. Communication can be minimized even more through the use of 
data compression and by batching partition assignments and results.

3.8.4 Termination

The master process keeps track of the partitions assigned and reported. Once all as-
signed partitions have been matched with results, the master process displays any solutions 
found with the total elapsed time, and terminates the web workers.

3.9 Disappointment

Some heuristics initially look promising but ultimately prove insufficient at improving 
I2NX performance. This could indicate that further analysis is needed. In other cases, the 
heuristic is simply ill advised.

While ordering of the variables (cubes) most often results in a dramatic affect on search 
time, ordering the domain, known as value ordering, produces little to no improvement. In 
I2NX, the values of each cube are ordered within opposite side pairs from low to high, and 
pairs’ lowest value also from low to high. Variations on this ordering almost never improve 
search time, and occasionally degrade search time. No other value ordering heuristic is as 
of yet found to improve search time for I2N.

Two thirds of the pairs on the first cube are sufficient to find all possible solutions. 
However, the selection of which two of the three possible pairs to use can affect the search 
time. No heuristic is yet found that consistently determines the best two of three pairs to 
use for optimal average search time.

The genetic algorithm for finding a good ordering prior to search runs for a limited 
time by considering the point of diminishing returns. The relatively short ordering time 
reliably yields large reductions in search time on average for cube sizes of 64 or greater 
for randomly generated problems. The factors of reductions are often in excess of 20, even 
50 or more. However, forcing the algorithm to run for an extended period of time, ever 
attempting a better ordering, rarely offers more than an additional factor of two or three 
in the reduction of subsequent search time. Furthermore, this extended ordering time is 
so far neither reliable in the gains it offers, nor predictable in the length of additional time 
required.
Chapter 4

Experimentation

While there are many heuristics and techniques available from which to choose, not all provide the same benefit. Some work better than others, and some better in concert with select others. Having created the I2NX framework in which to test and compare various heuristics for a range of I2N problems, empirical data is collected and analyzed.

4.1 Scope

Problems used for benchmarking I2NX performance range in size from 60 cubes to 80 cubes. All problems are generated using a pseudo random number generator. All have 6 occurrences of each color. They are all generated as solvable and delivered scrambled. Every problem is searched exhaustively to find all possible solutions. Brute force is applied only to the smallest of problems for comparison to more informed search. Backtracking is applied to all problems.

In addition, one problem each of 40-cubes, 56-cubes, 92-cubes, and 100-cubes are used to compare various configurations of heuristics and parallelism.

4.2 Hypothesis

Although 80-cube problems may have an astronomically large number of arrangements, application of combinatorial search heuristics can bring the average time to solve an 80-cube problem, from a sample of randomly generated problems, to only a few hours on a typical desktop computer commonly available as of 2015.

4.3 Design

To allow for ease of distribution, the design is based on a rich client web application written in JavaScript using web workers for concurrency. Compared to many other languages, JavaScript performs relatively well, it is widely available, easy to deploy, and easy to modify. Running in a modern Chrome browser using web workers, it requires very little modification to leverage parallel processing on multiple processing cores as well as distribution across multiple computers. Standard HTML user interface components suffice.

I2NX is divided into two main web apps. One uses a genetic algorithm to optimize the variable ordering as much as possible within a limited time. The other performs the combinatorial search using the various heuristics provided. Both use web workers for concurrency. On the primary machine used throughout the experiments, four workers consume all of the processor’s available resources provided by an Intel Core2 Quad CPU Q9300 @ 2.50GHz. Its 4 GByte RAM provides ample memory.
4.3.1 Rich Client Web App

A textarea component of HTML5 is used to input a given problem in JSON notation allowing a simple cut and paste operation. A selection box lets the user set the number of web workers created and thus the number of concurrent processes. Another selection box provides choices for controlling the degree of partitioning of the problem into fragments. This allows testing various combinations of fragmentation and parallelism. For instance, a 72-cube problem running on a 4-core processor typically completes in less elapsed time, and with less idle time, when using four web workers served by a pool of 81 fragments. A check-box allows the user to control whether or not the program completes after examining only 2 out of 3 pairs of opposite sides on the first cube. A selection box allows the user to adjust for the number of cubes in the problem. A start button allows the user to control when searching begins.

The backtracking algorithm is coded separately as a JavaScript module which works with various user constraint algorithms. Isolation of the constraint logic into a separate module allows for ease of replacement as only the JavaScript filename needs to be changed in I2NX in order to switch to an entirely different constraint logic module. Variable ordering is coded as a separate web app. Output from the ordering app can be cut from its textarea and pasted into the textarea of the search web app. This allows each app to be
modified and tested without any modification to the other app.

4.3.2 Multiple Heuristics

With the meta-heuristic of backtracking search coded as a separate module, the remaining modifications can be isolated in a module dedicated to various constraint logic algorithms. This allows for testing each variant with minimal change to I2NX itself. As far as backtracking module is concerned, the constraint logic is a black-box user module.

First, a simple constraint preventing any color count from exceeding two is tested. Because this heuristic alone provides such poor performance, only a single 40-cube problem is run to completion. This takes nearly 18 minutes, even while using four web workers to leverage 100% of available CPU capacity. However a 56-cube problem advances so slowly that estimated completion time is 200 hours or more.

The color counting is maintained in an array of the counts indexed by color and that array is maintained on a stack indexed by the current node, so that on backtracking, previous counts are maintained. Only the colors on the pair of opposite sides chosen on the current node’s cube are incremented on the array of counts at the position of the stack pointer and carried forward.

Second, by adding the preprocessing step of variable ordering, the same 40-cube prob-
lem can be run to completion in less than 2 minutes. By comparison, the 56-cube problem advances faster than before, but is still estimated to take 20 hours or more. This ordering of variables can have an even greater effect when tuned to account for the heuristics added next.

Next, with the addition of a constraint such that when the last occurrence of a color is encountered the color count must be two, performance is improved significantly. However, checking whether some color on the current node is a last occurrence is expensive as this involves searching a list of blocks indexed by color. Rather than repeat this search over and over again, this information can be built into a lookup table which memoizes the results of the original last-color search. With this improvement, the 40-cube problem can run to completion in less than 16 seconds. The 56-cube problem is completed in under 2.4 hours.

This forward checking can also be applied at the next-to-next-to-last occurrence. Although somewhat more overhead is introduced, the net gain can be kept positive by opportunistic ordering of the constraining rules. To keep track of all the forward checking a tabu table is introduced which can be included on the stack with the color counts. To keep data movement to a minimum, the stack is compressed such that both the tabu table and the color counts share an 8-bit array element. This works, even though the tabu table is indexed by node and the color counts by color, because the number of different colors is always the same as the number of cubes. The tabu table is kept in the high bits, while the color count is kept in the low bits.

For nodes that pass all of the above constraining rules, a further check on arc consistency is made for each distinct color on the current cube not already considered in the previous rules. By checking each subsequent node for a constraint violation deriving from the analysis of the ever changing tabu table and color counts, it may be discovered that the current path leads to unsatisfiability, thus justifying a backtracking move earlier than otherwise.

With these additional improvements, the 40-cube problem is completed in less than half a second, and the 56-cube problem is completed in less than 5.2 seconds.

4.3.3 Static Variable Ordering

This preprocessing step is coded as a separate web app as previously discussed. Because the number of possible orderings is n-factorial, the app uses a genetic algorithm to quickly search for a globally optimal ordering based on a cost estimation function which scores orderings regarding the span of colors among cubes and the proximity of the last occurrence of each color to the earliest cube in the sequence. The effect of variable ordering on I2N processing time can be significant. Speedups of 100 times or more have been seen.

Except for small values of n, n! is much greater than \(3^n\). Because a solution is orthogonal with regard to the ordering of the cubes, any ordering that contributes to a quicker solution gives a performance advantage without sacrificing correctness of the solution. This gives much latitude in searching for an optimal ordering as all orderings are admissible. The trade off with ordering is the time spent ordering compared to the decrease in time
spent searching for solutions as a result of a better ordering.

While it would be gratifying to know whether an optimal ordering is truly the global optimum, finding and comparing all locally optimal orderings would have a worse asymptotic complexity than that of the original problem. Therefore, we are left with a time-constrained search for a really good ordering, and not concerned with guaranteeing that it be the true global optimum. For problems of this type, simulated annealing or a genetic algorithm suffices.

For the I2N variable ordering app, a variation on genetic programming is used. Rather than start with a randomly generated ordering, a greedy algorithm which generates several locally optimal orderings creates the proto-generation. Each progenitor is assigned a distinct initial cube index and subsequent cubes are ordered such that they add the least number of new colors. Ties are broken by scoring the most color reuse as best. Remaining ties are broken randomly. This proto-generation thereby gives N orderings that are distinct and spread out while being locally optimal.

To speed the process, elitism is used mixed with limited random selection from the non-elite subset. The candidate orderings are sorted according to a cost estimation function which values more highly those orderings with minimal sum of color spans along with minimal sum of color last positions. Optimal weighting of the components for spans and lasts is discovered by analysis of the empirical data. (Such analysis might be adapted for machine learning, a possible future enhancement). The best one-third are selected along with a few randomly chosen non-elites for an iterative ordering-improvement regime.

Using the same cost estimation function, chosen orderings are subjected to a hill-climbing element-swapping algorithm. If the result of the attempted improvements scores better than before, the altered ordering is added to the pool of candidates. Once this is completed for a generation, all candidates are sorted again and the best third along with a few non-elites are chosen for the next round where the process is repeated. This is done until no improvements can be generated or the time-limit expires, whichever happens first. The best of the lot is presented for use by the search app.

4.3.4 Distributed Parallel Search

Although at best only a linear improvement in speed is expected, better utilization of existing hardware and the option of adding more hardware to computing a solution may be desired. By extending the number of concurrent web workers to harness all of the host’s processing capacity, the elapsed processing time can be decreased. For the author, waiting fewer hours for a result proved practical.

Because a search problem can run several minutes or hours on a single host, consideration was given to distributing the problem to several computers. This proved to be an especially easy modification due to the web app design. By partitioning the list of fragments into as many subsets as available hosts, and randomizing the selection of fragments into each subset to minimize differences in the amount of work each host would have to perform, each host can work independently of the others. The partitioning of fragments
into subsets and the collection of results is done on a single server. The heavy lifting of searching the fragmented problem space is done by the many client hosts. On one sample test run, a problem was run in parallel on 18 client hosts and completed 16.4 times faster than when run on a single host.

4.4 Results

This section details the results of introducing various heuristics to the I2N program. In all cases the algorithms ran in 4 concurrent workers.

4.4.1 Small Beginnings

Backtracking occurs on discovery that a color count would exceed two. Because performance is so poor, only a small 40-cube problem is completed.

40-cube problem, arbitrary ordering:

(3/3) 1361.724 seconds, (2/3) 1066.423 seconds

56-cube problem, arbitrary ordering:

(3/3) 1,080,000 seconds estimated (300 hours), (2/3) 720,000 seconds estimated (200 hours)

4.4.2 Variable Ordering

Cubes are favorably ordered prior to searching. Backtracking still occurs only on discovery that a color count would exceed two. Performance is about 10 times better, but not enough to see a 56-cube problem completed. Ordering time for 56-cubes is 7.5 seconds, for 40-cubes it is 1.7 seconds. This is in addition to the search time.

40-cube problem, improved ordering:

(3/3) 172.785 seconds, (2/3) 105.008 seconds

56-cube problem, improved ordering:

(3/3) 108,000 seconds estimated (30 hours), (2/3) 72,000 seconds estimated (20 hours)

4.4.3 Forward Checking

With the addition of a check on the possibility of lacking a two-count at some node having a last occurrence of some color, another 6- to 7-fold improvement in performance is achieved. A 56-cube problem can be completed in a few hours.

40-cube problem, improved ordering, forward checking lasts:

(3/3) 23.730 seconds, (2/3) 15.786 seconds

56-cube problem, improved ordering, forward checking lasts:
4.4.4 Constraint Propagation

With the addition of tabu table to improve arc consistency, a near 50-fold improvement is seen for the 40-cube problem, but a whopping 598-fold improvement for the 56-cube problem.

40-cube problem, ordered, lasts, arc consistency:
(3/3) 0.495 seconds, (2/3) 0.321 seconds

56-cube problem, ordered, lasts, arc consistency:
(3/3) 24.057 seconds, (2/3) 13.534 seconds

4.4.5 Aggressive Forward Checking with Local Consistency

With considerably more complex code, a nearly 3-fold improvement is obtained for the 56-cube problem, while the 40-cube problem suffers slightly from the additional complexity.

40-cube problem, ordered, aggressive:
(3/3) 0.591 seconds, (2/3) 0.415 seconds

56-cube problem, ordered, aggressive:
(3/3) 8.527 seconds, (2/3) 5.145 seconds

Without the benefit of improved ordering, these times suffer considerably as the number of cubes increases. For 40-cubes, the cost of ordering (1.7 seconds) exceeds the improvement of less than 0.8 seconds. Thus for problems of 40-cubes and smaller, ordering may well be skipped.

40-cube problem, unordered, aggressive:
(3/3) 1.6571 seconds, (2/3) 1.175 seconds

56-cube problem, unordered, aggressive:
(3/3) 647.038 seconds, (2/3) 393.525 seconds

Table 4.1 shows times for problems in the range of 60-cubes to 80-cubes, where this same ordered and aggressive search is used. While twelve problems were generated randomly for each size, it is their average search times which are listed.

4.4.6 Parallel Computing

By distributing larger problems to 27 computers similar in performance to the one used for the above benchmarks, the following times are expected for problems of 92-cubes and
100-cubes. Both estimated total run time and optimal elapsed time are shown.

92-cube problem, ordered, aggressive:

(3/3) 101,270 seconds (28.13 or 27 x 1.04 hours)
(2/3) 67,513 seconds (18.75 or 27 x 0.69 hours)

100-cube problem, ordered, aggressive:

(3/3) 776,606 seconds (215.72 or 27 x 7.99 hours)
(2/3) 517,737 seconds (143.82 or 27 x 5.33 hours)

To put these estimates to the test, both a 92-cube and a 100-cube problem were run to completion. The 92-cube problem took a total of 36.06 hours (27 * 1.34), nearly twice that estimated. However, the 100-cube problem completed after a total of only 139.84 hours (27 * 5.18), slightly less than estimated. Both times were within the expected variance. It is expected that a sample size of 12 for both the 92-cubes and 100-cubes problems would yield an average more in line with the estimates.

Estimation formula used: $4.52e-6 \times 1.29^n$

<table>
<thead>
<tr>
<th>Cubes</th>
<th>Ordering</th>
<th>3/3 average</th>
<th>2/3 average</th>
<th>estimated</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>15 seconds</td>
<td>30 seconds</td>
<td>19 seconds</td>
<td>20 seconds</td>
</tr>
<tr>
<td>64</td>
<td>34 seconds</td>
<td>81 seconds</td>
<td>50 seconds</td>
<td>54 seconds</td>
</tr>
<tr>
<td>68</td>
<td>69 seconds</td>
<td>209 seconds</td>
<td>137 seconds</td>
<td>150 seconds</td>
</tr>
<tr>
<td>72</td>
<td>114 seconds</td>
<td>597 seconds</td>
<td>406 seconds</td>
<td>415 seconds</td>
</tr>
<tr>
<td>76</td>
<td>168 seconds</td>
<td>1893 seconds</td>
<td>1255 seconds</td>
<td>1148 seconds</td>
</tr>
<tr>
<td>80</td>
<td>229 seconds</td>
<td>4984 seconds</td>
<td>3176 seconds</td>
<td>3179 seconds</td>
</tr>
</tbody>
</table>

Table 4.1: Average Times for Aggressive Algorithms.
Chapter 5

Future Work

While testing the effectiveness of the various heuristics described so far, many additional possibilities were considered but left untested due to time constraints. It seems there remains much potential for improvement in solving I2N. Here a few of those considerations are mentioned.

5.1 Cluster Analysis

Initial evaluation of clustering patterns within I2N problems show some promise of improving the proto-generation for subsequent improvement. Extending this to the scoring and improvement of spans and lasts may yield better ordering optimization.

5.2 Meta-Heuristics for Automated Tuning

Parameterization of the cost function weightings for scoring orderings could allow for an automated tuning algorithm[5].

5.3 Gamification

By turning I2N into a game to stimulate competition for better performing heuristics, it may be possible to crowd-source the optimization effort.

Initial efforts at creating an Instant Insanity simulation using HTML5 and CSS3 transforms seems promising. The user can play a 4-Cube game in the browser. Clicking on various quadrants of each cube’s face triggers an animation of the cube rotating to a new orientation. When the cubes are aligned properly the user is rewarded with a congratulatory message. Various aspects of the game are configurable including choice of color palette, randomization of the color assignments, randomization of the cube order, randomization of the cube orientation, background image, and sound.

Another effort using the JavaScript library Three.js produced a 20-cube game that allows the user to manipulate the game in a simulated 3D world. Two camera views are available. Colors can be replaced with images. The user can even drag images from the desktop to replace a set of colors or existing image. Keyboard arrow keys control the direction of rotation of a selected cube. The ’s’ and ’a’ keys are used to move the cube selection indicator forward or backward. Although it is conceivable that the 20-cube problem could be solved it is unlikely without the help of an algorithmic assist. One future direction for this game prototype is to provide a Code-Pen like interface encouraging the player to write a working solver.

5.4 Non-von Neumann Architecture

As computer hardware design advances, potential for combinatorial search using alternatives to the traditional CPU become attractive. While backtracking search has yet
to benefit from existing GPU architectures, near-future designs for general purpose GPUs may provide adequate memory-CPU-GPU bandwidth to support backtracking search al-

31
gorithms in a way that increases performance. Massive finite state machines built on top of special purpose memory chips such as Micron’s Automata Processor might be used to accelerate search[9]. Special circuitry modified in real time using field programmable gate arrays, FPGAs, might one day become part of commodity hardware and made available as a type of co-processor[10]. Recent release of various software development kits to promote the potential of such emerging architectures seem worth exploring in the context of combinatorial search optimization.
Chapter 6
Conclusion

6.1 Moving Data Costs

Use of Typed Arrays, making efficient use of the stack, passing by reference, and the use of indexes as proxies for more complex values, had a larger affect on performance than expected.

6.1.1 Minimizing Data Helps

Using integers instead of long color name strings to represent color assignments not only keeps the overall representation compact but simplifies many calculations and indexing. Reusing the stack for maintaining both color counts and tabu tables had an outsized impact on performance, possibly by minimizing cache churn.

6.1.2 Minimizing Movement Helps

Passing arrays by reference and maintaining those references by using the splice operator rather than new assignments minimized copying of large data structures.

6.2 Overhead Costs

Many heuristic algorithms were tried that failed to yield a net gain in performance because due to lack of success in mitigating the overhead costs involved. Although these algorithms were in nearly all cases able to reduce the branching depth or branching factor, inability to amortize overhead by either sharing structures or logic with other algorithms resulted in net losses.

6.2.1 Avoiding Work requires Work

Reducing the amount of work a program does each time though a loop generally reduces the total time to complete the loop. However, reducing the number of loops required to reach a solution may have an even greater affect. Paradoxical as it may seem, adding algorithms to I2NX resulted in faster solutions. Many of the heuristic algorithms used to reduce the overall time to solve I2N increased the number of constraining rules while paying for themselves by reducing the branching depth or branching factor. But what works especially well for one problem may slow the time to solution for another. When an improvement in the average time is sought some additional work may end up being more than amortized over all.

6.2.2 Synergistic Heuristics

Having analyzed the efficiency of several heuristic techniques in isolation, it was somewhat surprising to see the degree of improvement when used in concert with other techniques.
6.2.3 Non-overlapping Coverage

One area that required a great deal of analysis was to eliminate duplicate work done by overlapping coverage. By looking at the hit rate of each rule and noticing that for all test cases the rule never resulted in a constraint violation some rules were removed. Further analysis revealed in some cases that the cause was a duplication of effort. Some preceding rule always identified the constraint violation and the subsequent rule was entirely unnecessary. In other cases, the condition that the rule tested was so obscure and rare that the rule failed to meet the benefit to cost ratio minimum threshold.

6.2.4 Order of Execution Matters

On more than one occasion experimental reordering of rules improved average performance even though on certain problems another ordering gave better performance. This leads to the possibility that such cases might be detected by some characteristic of the input and used to optimize I2N.

6.3 Exponential Complexity Limits Scalability

All told, the combination of techniques used in I2NX brought down the total time more than anticipated for solving the average of 12 randomly generated 80-cube problems on a aging desktop computer. Yet comparing the increase of average times for problems of size from 60 to 80 reveals a persistent exponential growth rate. As expected, this effort does not threaten the status of the $P \neq NP$ conjecture.
Appendix A

Data: Inputs, Solutions, Times

A.1 Completion times for 60, 64, 68, 72, 76, and 80-cube problems

//Completion times for 12 randomly generated 60-cube problems in seconds
[54, 32, 19, 44, 28, 17, 42, 49, 19, 25, 9, 19]
// 3/3 sum: 357, avg: 30
[37, 21, 15, 25, 17, 10, 26, 29, 11, 17, 5, 13]
// 2/3 sum: 226, avg: 19

//Completion times for 12 randomly generated 64-cube problems in seconds
[76, 62, 56, 47, 33, 57, 111, 165, 148, 123, 47, 49]
// 3/3 sum: 974, avg: 81
[48, 43, 32, 30, 20, 37, 62, 121, 94, 76, 32, 34]
// 2/3 sum: 599, avg: 50

//Completion times for 12 randomly generated 68-cube problems in seconds
[[324, 161, 269, 138, 183, 130, 199, 173, 264, 255, 238, 176]
// 3/3 sum: 2510, avg: 209
[217, 81, 177, 87, 122, 95, 132, 99, 172, 179, 173, 112]
// 2/3 sum: 1646, avg: 137

//Completion times for 12 randomly generated 72-cube problems in seconds
[700, 685, 498, 348, 508, 881, 385, 216, 860, 344, 775, 958]
// 3/3 sum: 7158, avg: 597
[487, 458, 293, 247, 323, 596, 200, 173, 643, 263, 475, 710]
// 2/3 sum: 4868, avg: 406

//Completion times for 12 randomly generated 76-cube problems in seconds
[2282, 2347, 2692, 2556, 957, 1092, 2076, 659, 1256, 1402, 4130, 1266]
// 3/3 sum: 22715, avg: 1893
[1385, 1505, 1985, 1748, 634, 741, 1261, 471, 804, 798, 2976, 756]
// 2/3 sum: 15064, avg: 1255

//Completion times for 12 randomly generated 80-cube problems in seconds
A.2 Data and time for one 80-cube problem

// First 80-cube problem, its reordering, and search completion time

[[[40, 17], [24, 58], [73, 53]], [[13, 75], [77, 62], [16, 4]],
[[20, 63], [8, 51], [53, 47]], [[9, 2], [4, 33], [52, 72]],
[[79, 29], [8, 32], [28, 75]], [[23, 12], [57, 29], [18, 48]],
[[0, 71], [23, 42], [36, 40]], [[2, 36], [36, 28], [70, 2]],
[[7, 24], [55, 16], [34, 44]], [[1, 43], [32, 58], [70, 12]],
[[51, 63], [43, 43], [2, 47]], [[52, 67], [27, 65], [76, 10]],
[[79, 25], [6, 62], [37, 30]], [[48, 11], [69, 65], [36, 78]],
[[9, 59], [39, 76], [3, 26]], [[55, 77], [48, 73], [62, 51]],
[[35, 74], [64, 16], [20, 67]], [[79, 37], [78, 37], [75, 41]],
[[54, 54], [31, 1], [64, 0]], [[57, 72], [61, 49], [59, 71]],
[[68, 54], [36, 18], [38, 17]], [[39, 23], [31, 46], [42, 76]],
[[4, 44], [40, 21], [47, 29]], [[16, 13], [46, 13], [78, 56]],
[[64, 24], [28, 6], [61, 67]], [[75, 52], [77, 39], [56, 36]],
[[49, 48], [1, 76], [61, 56]], [[10, 4], [22, 50], [26, 45]],
[[61, 13], [49, 20], [8, 38]], [[33, 28], [66, 15], [27, 3]],
[[7, 13], [73, 44], [31, 56]], [[12, 66], [48, 57], [79, 20]],
[[65, 28], [38, 34], [74, 23]], [[19, 55], [74, 48], [32, 58]],
[[33, 26], [45, 59], [30, 18]], [[53, 17], [8, 21], [24, 66]],
[[66, 7], [14, 51], [44, 33]], [[14, 75], [37, 22], [45, 9]],
[[22, 15], [60, 5], [38, 6]], [[39, 8], [54, 68], [17, 8]],
[[1, 30], [10, 72], [69, 64]], [[41, 29], [46, 53], [27, 39]],
[[63, 24], [71, 42], [4, 0]], [[11, 7], [27, 43], [62, 65]],
[[38, 42], [75, 22], [1, 3]], [[53, 5], [52, 21], [77, 28]],
[[32, 16], [5, 50], [47, 10]], [[62, 35], [55, 0], [58, 33]],
[[12, 77], [0, 29], [3, 59]], [[35, 59], [57, 65], [71, 41]],
[[19, 67], [19, 46], [59, 54]], [[30, 4], [11, 15], [15, 18]],
[[76, 61], [10, 11], [66, 15]], [[23, 45], [67, 57], [25, 31]],
[[27, 69], [56, 63], [26, 17]], [[3, 45], [78, 25], [24, 79]],
[[65, 2], [69, 60], [60, 14]], [[70, 71], [74, 45], [66, 42]],
[[58, 77], [30, 46], [72, 25]], [[26, 69], [72, 70], [34, 49]],
[[32, 43], [5, 74], [25, 39]], [[55, 54], [21, 58], [21, 63]],
[[43, 25], [19, 40], [19, 50]], [[47, 53], [14, 26], [20, 0]],
[[41, 23], [5, 31], [60, 60]], [[61, 50], [44, 60], [31, 11]],
[[68, 32], [34, 50], [16, 64]], [[140, 30], [22, 73], [2, 63]],
[[27, 51], [34, 19], [14, 65]], [[18, 49], [70, 3], [37, 69]],
[[9, 22], [78, 51], [34, 14]], [[13, 52], [6, 78], [42, 11]],
[[68, 10], [76, 12], [37, 57]], [[18, 21], [33, 29], [73, 64]],
[[38, 40], [5, 49], [6, 17]], [[12, 71], [7, 35], [35, 35]],
[[62, 70], [7, 46], [47, 50]], [[73, 52], [41, 15], [44, 74]],
[[41, 1], [72, 79], [9, 55]], [[20, 6], [67, 9], [56, 68]]

// ordering resulting from genetic algorithm:

[[0.6737437033364994, 0.4727027027027027, 0.5314864864864864, 1.0, 0.31961527216576846, 0.20573670444638184, 0.3891473684210527, 0.48118205217391307, [80, 0.0004054054054054055, [54, 54], 57], [47], [20, 4], 1.3945945945945948, 0.0002027027027027027, 0.3870967741935484]]

36
A.3 Data, solutions, and times for one 100-cube problem


37
```python
[[52, 93], [63, 5], [5, 1]], [[94, 45], [82, 78], [43, 32]],
[[69, 48], [11, 97], [12, 29]], [[45, 72], [2, 21], [29, 57]],
[[44, 21], [53, 14], [71, 8]], [[35, 98], [26, 59], [33, 2]],
[[24, 34], [96, 74], [4, 24]], [[70, 67], [85, 32], [40, 12]],
[[36, 84], [13, 81], [81, 16]], [[45, 72], [2, 21], [29, 57]],
[[53, 82], [41, 36], [86, 9]], [[11, 79], [41, 41], [20, 13]],
[[60, 44], [50, 33], [38, 65]], [[44, 5], [89, 64], [39, 32]],
[[15, 80], [58, 65], [68, 72]], [[3, 53], [65, 86], [4, 91]],
[[58, 41], [7, 99], [80, 1]], [[96, 95], [94, 62], [55, 25]],
[[30, 0], [56, 75], [77, 48]], [[36, 93], [6, 26], [49, 69]],
[[19, 66], [76, 19], [47, 29]], [[12, 55], [92, 97], [25, 18]],
[[78, 37], [5, 4], [33, 42]], [[42, 46], [20, 66], [92, 9]],
[[68, 39], [13, 71], [10, 89]], [[49, 9], [98, 61], [30, 42]],
[[58, 92], [81, 64], [27, 59]], [[56, 60], [57, 64], [14, 70]],
[[54, 80], [95, 45], [55, 78]], [[18, 85], [58, 75], [38, 88]],
[[21, 51], [86, 27], [22, 29]], [[17, 10], [35, 40], [26, 34]],
[[52, 10], [23, 30], [97, 91]], [[90, 72], [43, 84], [44, 38]],
[[51, 4], [98, 76], [0, 28]], [[54, 88], [20, 24], [21, 3]],
[[96, 86], [4, 15], [31, 2]], [[42, 0], [39, 47], [57, 17]],
[[99, 61], [71, 88], [39, 61]], [[28, 56], [81, 16], [51, 5]],
[[12, 34], [99, 81], [63, 37]], [[1, 79], [49, 49], [47, 80]],
[[81, 52], [7, 56], [41, 11]], [[69, 38], [57, 22], [54, 33]],
[[59, 62], [78, 20], [90, 94]], [[93, 78], [62, 63], [0, 57]],
[[51, 49], [73, 40], [8, 80]], [[65, 83], [77, 17], [24, 9]],
[[68, 78], [73, 25], [35, 77]], [[37, 2], [38, 12], [67, 71]],
[[60, 72], [75, 91], [48, 91]], [[90, 30], [40, 19], [77, 37]],
[[99, 74], [8, 63], [8, 71]], [[50, 15], [83, 22], [31, 31]],
[[6, 65], [47, 32], [69, 25]], [[46, 31], [75, 73], [22, 52]],
[[83, 39], [70, 27], [16, 83]], [[64, 24], [42, 59], [29, 19]],
[[82, 25], [84, 27], [46, 66]], [[23, 50], [34, 99], [85, 82]],
[[9, 40], [9, 10], [79, 15]], [[3, 76], [74, 54], [33, 1]],
[[44, 55], [45, 84], [3, 14]], [[79, 59], [60, 12], [22, 87]],
[[19, 14], [28, 1], [48, 60]], [[79, 38], [7, 89], [60, 71]],
[[63, 64], [50, 89], [16, 26]], [[56, 92], [32, 51], [92, 14]],
[[6, 53], [74, 77], [70, 73]], [[30, 34], [75, 7], [91, 87]],
[[46, 96], [37, 10], [58, 71]], [[76, 93], [57, 16], [7, 88]],
[[11, 74], [55, 95], [11, 36]], [[37, 90], [20, 10], [2, 85]],
[[56, 94], [61, 45], [67, 96]], [[47, 69], [31, 97], [52, 73]],
[[20, 48], [66, 90], [50, 13]], [[15, 94], [8, 87], [2, 14]],
[[33, 62], [3, 5], [87, 84]], [[67, 43], [93, 70], [11, 93]],
[[66, 95], [27, 77], [3, 6]], [[39, 54], [92, 89], [96, 36]],
[[17, 80], [35, 23], [17, 67]], [[22, 41], [95, 72], [79, 99]],
[[85, 28], [54, 51], [73, 43]], [[23, 69], [62, 1], [65, 66]],
[[30, 43], [74, 86], [28, 49]], [[68, 61], [90, 34], [36, 0]],
[[68, 35], [74, 86], [6, 17]], [[91, 88], [8, 18], [98, 62]],
[[59, 18], [82, 89], [18, 13]], [[98, 75], [50, 58], [29, 42]]

// used this ordering:
//[(0.6716816298087729, 0.47310638297872365, 0.5167659574468086, 0.3201613975508711, 0.2000384351407001, 0.38165263157894735, 0.48251437477406676), 3, 3, 265452]
```
null
null
null
null
null
null
null
0,2,0,0,1,2,0,1,0,0,2,1,2,2,1,1,1]]},null]

//first solution found in 58 minutes, 34 seconds
//remainder found in 4 hours, 38 minutes, 23 seconds

Total: 139.84 hours
Host average: 5.17926 hours
References


