ABSTRACT
In this paper, we introduce a new algorithmic paradigm called dissection, which allows us to solve a special type of combinatorial search problems with greatly reduced combinations of time and space complexities. To demonstrate the wide applicability of our algorithms, we show how to untangle Rubik’s cube and how to solve the NP-complete partition problem with improved efficiency. As most combinatorial search problems, these problems can be described by a collection of possible states, a list of possible actions which map each current state into some next state, and a pair of initial and final states. The problem is to find a sequence of actions which map the given initial state into the desired final state. One can always solve such problems by trying out all the possible actions, but in many cases (such as Rubik’s cube and the partition problem), it is possible to exploit special properties of the states and actions in order to lower this exponential complexity. We call such problems, whose solution can be partitioned both along the action and the state dimensions, bicomposite search problems, and show that any such problem can be efficiently solved with our new dissection algorithms by partially guessing several states which are generated after applying various numbers of actions.

1. INTRODUCTION
A central problem in the design of efficient algorithms is how to solve search problems, in which we are given a pair of states and a collection of possible actions, and we are asked to find how to get from the first state to the second state by performing some sequence of actions. In some cases, we only want to decide whether such a sequence exists at all, while in other cases it is clear that such sequences exist but we are asked to find the shortest possible sequence. Many search problems of this type have associated decision problems which are NP-complete, and thus we do not expect to find any polynomial time algorithms which can solve all their instances. However, what we hope to find are new exponential time algorithms whose exponents are smaller than in the best previously known algorithms. For example, the problem of breaking a cryptographic scheme whose key has $n = 100$ unknown bits cannot be solved in a practical amount of time via an exhaustive key search algorithm, since its time complexity of $2^n$ would be beyond reach even for the largest currently available data center. However, if we manage to find a better cryptanalytic attack whose running time is $2^{n/2}$, we can break the scheme with a modest effort in spite of the exponential nature of this complexity function. One trick which is often helpful in such situations is to find a tradeoff between the time and space complexities of the attack: Exhaustive search requires a lot of time but a negligible amount of memory, and thus a tradeoff which uses more memory (in the form of large tables of precomputed values) in order to reduce the time (by skipping many computational steps) will be very beneficial. For reasons which are explained in the extended version of this paper (available at [2]), we usually consider the product of the amount of time and the amount of space required by the algorithm as the appropriate complexity measure that we try to minimize. In the example above, breaking the cryptosystem with $T = 2^n$ time and a $S = 1$ space is infeasible, breaking it with $T = 2^{n/3}$ time and $S = 2^{n/3}$ space (whose product $TS = 2^n$ is the same as before) is better but still barely feasible, and breaking it in $T=2^{n/2}$ time and $S = 2^{n/4}$ space (whose product $TS = 2^{3n/4}$ has a smaller exponent) is completely feasible.

A typical search problem is defined by a condition $F$ (e.g., in the form of a CNF Boolean formula which is a conjunction of clauses) and a candidate solution $X$ (e.g., in the form of a 0/1 assignment to all the variables in $F$), and the goal is to find among all the possible $X$ at least one that satisfies the condition that $F(X)$ is true. Such a representation has no internal structure in the sense that it uses the full description of $F$ and the full value of $X$ in order to decide whether $F(X)$ is satisfied. However, we can usually replace the all-or-nothing choice of $X$ by a sequence of smaller decisions. For example, we can start with the assignment of 0 to
all the variables, and at each stage we can decide to flip the current value of one of the Boolean variables. At any intermediate point in this process \( F \) is in a state in which some of its clauses are satisfied and some are not, and our goal is to reach a state in which all the clauses are simultaneously satisfied. More generally, we say that a search problem is \textit{composite} if solving it can be described by a sequence of atomic actions, which change the system from some initial state through a series of intermediate states until it reaches some final desired state. Most search problems have such a composite structure, which can be represented by the \textit{execution matrix} described in Figure 1. The rows of this matrix represent states \( S_0, S_1, \ldots \), and the solution \( X \) is represented by the sequence of actions \( a_1, a_2, \ldots \) on the left side of the figure in which action \( a_i \) changes state \( S_{i-1} \) to state \( S_i \). In many cases, the atomic actions are invertible operations over the states, which make it possible to map \( S_{i-1} \) to \( S_i \) by using the action \( a_i \), and to map \( S_i \) back to \( S_{i-1} \) by using the inverse action \( a_i^{-1} \). For example, in the case of Boolean formulas we are allowed to flip the current value of any one of the variables in \( X \), but we can cancel its effect by flipping the same variable a second time, so in this case the action and its inverse happen to be the same. In this paper we consider only such invertible cases, which make it possible to split the search problem by applying some forward actions to the initial state, applying some inverse actions to the final state, and searching for a meet-in-the-middle (MITM) state which combines these parts. Our new dissection technique can be applied even when some of the actions are not invertible, but this makes its description more complicated and its complexity slightly higher, as discussed in [2].

\[ S_0 \quad S_1 \quad S_2 \quad \cdots \]

\[ a_1 \quad a_2 \quad a_{i-1} \quad a_i \quad a_{i+1} \quad a_{i+2} \quad a_1 \quad a_2 \]

**Figure 1: An Execution Matrix of a Composite Search Problem**

So far we have partitioned the execution matrix into multiple rows by dividing the solution process into a series of atomic actions. In order to apply our new dissection technique, we also have to partition the execution matrix into multiple columns by dividing each state \( S_i \) into smaller chunks \( S_{i,j} \) which we call substates, as described in Figure 2. However, only partitions in which the substates can be manipulated independently of each other will be useful to us. We say that a search problem has a \textit{bicomposite structure} if it can be described by an execution matrix in which the knowledge of the action \( a_i \) makes it possible to uniquely determine substate \( S_{i,j} \) from \( S_{i-1,j} \) and \( S_{i-1,j} \) from \( S_{i,j} \), even when we know nothing about the other substates in the matrix. This immediately implies that if we choose any rectangle of any dimensions within the execution matrix, knowledge of the substates \( S_{i-1,j}, S_{i-1,j+1}, \ldots, S_{i,k} \) along its top edge and knowledge of the actions \( a_i, a_{i+1}, \ldots, a_r \) to its left suffices in order to compute the substates \( S_{t,j}, S_{t,j+1}, \ldots, S_{t,k} \) along its bottom edge, and vice versa.

**Figure 2: An Execution Matrix of a Bicomposite Search Problem**

Not every search problem has such a bicomposite representation, but as we demonstrate in this paper there are many well known problems which can be represented in such a way. When a problem is bicomposite, we can solve it with improved efficiency by using our generic new technique. Our main observation is that in such cases, we can improve the standard meet-in-the-middle algorithms (which try to match the forward and backward directions at a full intermediate state) by considering algorithms which only partially match the two directions at a partially specified intermediate state. In addition, we can reverse the logic of MITM algorithms, and instead of trying to converge from both ends of the execution towards some intermediate state which happens to be the same, we can start by partially guessing this intermediate state in all possible ways, and for each guessed value we can break the problem into two independent subproblems by proceeding from this intermediate state towards the two ends (exploiting the fact that in bicomposite problems we can determine the effect of long sequences of actions on partially specified states). We can then solve each one of these subproblems recursively by partially guessing additional substates in the execution matrix. We call this approach a \textit{dissection} algorithm since it resembles the process in which a surgeon makes a sequence of short cuts at various strategically chosen locations in the patient’s body in order to carry out the operation. For example, in Section 4 we show how to solve the well known combinatorial partition problem by first guessing only two seventh of the bits of the state which occurs after performing three seventh of the actions, and then solving one of the resultant subproblems by guessing in addition one seventh of the bits of the state which occurs after performing five seventh of the actions.

Our main purpose in this paper is to introduce the new ideas by applying them in the simplest possible way to several well known search problems. We intentionally overlook several nuisance issues whose proper handling is not easy to explain, and ignore several possible optimizations which can further reduce the complexity of our algorithms. When we analyze the running time of our algorithms, we often assume for the sake of clarity that the instance we try to solve...
is randomly chosen, and that the intermediate states we try
to guess are uniformly distributed.

The paper is organized as follows. In Section 2 we describe
the problem of solving Rubik’s cube with the smallest pos-
sible number of steps as a bicomposite search problem, and
in Section 3 we show how to solve it with time complexity
which is approximately the square root of the size of the
search space, and a space complexity which is approxi-
imately the fourth root of the size of the search space by
using the simplest version of the new dissection algorithm.
In Section 4 we describe several improvements of the basic
dissection technique, and show how to use them in order
to solve a different search problem called the combinatori-

2. REPRESENTING RUBIK’S CUBE AS A
BICOMPOSITE SEARCH PROBLEM

In this section we show how to construct a bicomposite
representation of the well known problem of solving a stan-
dard $3 \times 3 \times 3$ Rubik’s cube [6]. We can assume that we
always hold the cube in a fixed orientation, in which the
white center color is at the top, the yellow center color is
on the left, etc. One of the 27 subcubes is at the center of
the cube, and we can ignore it since it is completely invis-
able. The six subcubes at the center of each face are not
moved when we rotate that (or any other) face, and thus
we can ignore them as well in our state representation. The
actions we can take are to rotate each one of the six faces
of the cube by 90 degrees, 180 degrees, or 270 degrees (we
are not allowed to rotate a center slice since this will change
the standard orientation of the cube defined above). Con-
sequently, we have a repertoire of 18 atomic actions we can
apply to each state of the cube. Note that all these actions
are invertible mappings on the state of Rubik’s cube in the
sense that the inverse of a 90 degree rotation is a 270 de-
gree rotation applied to the same face, and both of them are
available as atomic actions.

Among the 27 – 6 – 1 = 20 subcubes which we can move,
12 have two visible colors and are called edge subcubes, and
8 have three visible colors and are called corner subcubes.
Each such subcube can be uniquely described by the com-
bination of colors on it, such as a blue-white (BW) edge
subcube, or a green-orange-red (GOR) corner subcube. In
addition, each location on the cube can be described by its
relevant sides (i.e., a combination of top/bottom, left/right,
front/back). We can thus describe any state of the cube by
a vector of length 20, whose $i$-th entry describes the current
location of the $i$-th subcube (e.g., the first entry in the vector
will always refer to the blue-green edge subcube, and specif-
ify that it is currently located at the top-front position). To
complete the specification, we also have to choose some stan-
dard orientation of the colors, and note that edge subcubes
can be either in the standard (e.g., BW) or in an inverted
(e.g., WB) state, and each corner subcube can be in one of
two possible orientations (e.g., GOR, ORG, or GOR).

Note that any possible action can only move edge subcubes
to edge subcubes and corner subcubes to corner subcubes.
If we use the first 12 positions in the state vector to describe
the current locations of the 12 edge subcubes (in some fixed
order), then each entry in these positions can be described
by a number between 1 and 24 (specifying in which one of
the 12 possible positions it is currently located and in which
one of its 2 possible orientations). Similarly, when we use
the last 8 positions in the state vector to describe the cur-
rent locations of the 8 corner subcubes (in some fixed order),
then each one of these entries can again contain a number
between 1 and 24, this time specifying in which one of the 8
possible positions it is located and in which of its 3 possible
orientations. We can thus describe any state of Rubik’s cube
by a vector of length 20 whose entries are numbers between
1 and 24. Any one of the 18 atomic actions will change 8 of
the entries in this vector, by moving edge subcubes and 4
corner subcubes to new positions and orientations, leaving
the remaining 12 entries unchanged.

![Figure 3: Rubik’s Cube](image-url)
orientation of that particular subcube (namely, the value of \( S_{i,j} \)) even when we know nothing about the current location and orientation of any other subcube (namely, all the other \( S_{i,k} \) values in the execution matrix). Notice that many other natural representations of the states of Rubik’s cube do not have such a bicomposite structure. For example, if we associate the first entry in the state vector with a particular cube position (such as top-front) and use it to denote which edge subcube (such as BW) is currently located in it and in which orientation, then knowledge of just this entry in the first state does not tell us anything about which edge subcube (such as GR) replaces it at the top-front position if we rotate the top face by 90 degrees. Such a representation requires knowledge of other columns in the execution matrix, depending on which action was applied to the state, and thus we cannot use it in our new dissection technique.

As shown in Section 3, we can use the bicomposite representation in order to find for any given initial state of Rubik’s cube a sequence of up to 20 face rotations using a completely feasible combination of a time complexity which is the square root of the size of the search space (namely, in about \( 2^{39} \) steps, or a few minutes on a standard PC) and a space complexity which is about the fourth root of this number (namely, about \( 2^{19.5} \) memory locations, or a few megabytes). The resultant algorithm is completely generic, makes no use of the details of the problem besides its bicompositeness, and matches the complexities of the best previous algorithm for solving Rubik’s cube (designed about 25 years ago, see [3]) which was highly specific and depended on the group-theoretic properties of the set of permutations defined by Rubik’s cube.

3. THE BASIC DISSECTION TECHNIQUES

We now assume that we are given an initial state vector \( S_0 \) and a final state vector \( S_f \) of Rubik’s cube, and our goal is to find a series of atomic actions \( a_1, a_2, \ldots, a_\ell \) that transforms the initial state into the final state. As described in the previous section, we know that \( \ell = 20 \) suffices to find a solution, and hence our goal is to find \( a_1, a_2, \ldots, a_\ell \).

Our dissection algorithms are extensions of the classical meet-in-the-middle (MITM) algorithm, which was first presented in 1974 by Horowitz and Sahni (see [5]) in order to solve the Knapsack problem. We can apply a MITM algorithm to almost any composite problem with invertible actions. When the size of the search space is \( 2^n \), the MITM algorithm requires about \( 2^{n/2} \) time and \( 2^{n/2} \) space. For the sake of completeness, we describe below how to apply this algorithm in the context of Rubik’s cube, whose search space has about \( 2^{78} \) states. In this case, a time complexity of \( 2^{78/2} = 2^{39} \) is feasible, but a space complexity of \( 2^{78/2} = 2^{39} \) random access memory locations is too expensive.

The complete details of the algorithm are given in Figure 4. The first step of the algorithm is to iterate over all possible action vectors \( a_1, \ldots, a_{\ell} \) and obtain a possible value for \( S_{10} \). We store the \( 2^{39} \) values of \( a_1, \ldots, a_{10} \) in a list next to the corresponding values of \( S_{10} \), and sort \(^2\) them according to \( S_{10} \). Next, we iterate over all possible action vectors \( a_{11}, \ldots, a_{20} \). Again,

\[ \text{there are about } 2^{39} \text{ such action vectors, and for each one, we apply its inverted actions to } S_{20} \text{ and obtain a possible value for } S_{10}. \]

The MITM algorithm requires about \( 2^{39} \) memory cells in order to store the sorted list, and its time complexity is about \( 2^{39} \), which is the time required in order to iterate over each one of the action vectors \( a_1, \ldots, a_{10} \) and \( a_{11}, \ldots, a_{20} \).

Algorithm MITM-Rubik

**Input:** Initial state \( S_0 \) and final state \( S_{20} \)

**for all** \( a_1, a_2, \ldots, a_{10} \) **do**

Compute \( S_{10} = a_{10}(\ldots(a_2(a_1(S_0))\ldots)) \)

Store \( (S_{10}, a_1, a_2, \ldots, a_{10}) \) in a list \( L \)

Sort \( L \) according to the value of \( S_{10} \) in each entry (under some lexicographical order)

**for all** \( a_1, a_2, \ldots, a_{20} \) **do**

Compute \( S_{10} = a_{20}(\ldots(a_{19}(a_{18}(\ldots(a_{10}(S_{20})\ldots))) \ldots) \)

Search for \( S_{10} \) in \( L \)

\[ \text{if } S_{10} \text{ is found then} \]

\[ \text{return the associated } (a_1, a_2, \ldots, a_{10}) \text{ and } a_{11}, a_{12}, \ldots, a_{20} \text{ as a solution} \]

**Figure 4:** Meet-in-the-Middle Procedure to Solve the Rubik’s Cube

3.1 Improving the MITM Algorithm Using Dissection

In this section, we show how to improve the classical MITM algorithm on Rubik’s cube by using a basic version of our new dissection technique. The main idea here is to “dissect” the execution matrix in the middle by iterating over all the possible values of some part of the middle state \( S_{10} \). The size of the partial \( S_{10} \) that we iterate on is chosen such that it contains about \( 2^{14} = 2^{19.5} \) partial states. Since \( S_{10} \) is represented as a 20-entry vector, where each entry can attain 24 values, we choose to iterate on its first 4 entries, which can assume about \( 2^{14} \approx 2^{18.5} \) values. For each such partial value of \( S_{10} \), we use the bicomposite structure of the problem in order to independently work on the two partial execution matrices shown in Figure 5 as red and blue rectangles, and finally join the partial solutions in order to obtain the full action vector.

The complete details of the algorithm are given in Figure 6. We have an outer loop which iterates over all the possible values of \( S_{10,1}, S_{10,2}, S_{10,3}, S_{10,4} \). Assuming that this value is correctly guessed, we first concentrate on the upper part of the execution matrix and find all the partial action vectors \( a_1, \ldots, a_{10} \) which transform \( S_{10,1}, S_{10,2}, S_{10,3}, S_{10,4} \) into \( S_{10,1}, S_{10,2}, S_{10,3}, S_{10,4} \). This is done using a simple MITM algorithm on this smaller execution matrix. For each solution \( a_1, \ldots, a_{10} \) that we obtain using the MITM algorithm, we apply its actions to the full state \( S_0 \), obtain candidate values of the full \( S_{10} \) state, and store it next to \( a_1, \ldots, a_{10} \) in a list. After the MITM algorithm finishes populating the list, we sort it (e.g., in lexicographic order) according to the value of \( S_{10} \).
We now focus on the bottom execution matrix and find all the partial action vectors \(a_1, \ldots, a_0\) which transform \(S_{10,1}, S_{10,2}, S_{10,3}, S_{10,4}\) into \(S_{20,1}, S_{20,2}, S_{20,3}, S_{20,4}\). We use the same idea that we used for the upper part, i.e., we execute a MITM algorithm on the bottom execution matrix. For each solution \(a_1, \ldots, a_0\) that we obtain, we apply its inverse actions to \(S_0\) and obtain a value for \(S_0\). Then, we check for matches on \(S_{10}\) in the sorted list, and for each match, we output a full solution \(a_1, a_2, \ldots, a_0\).

In order to analyze the algorithm, we fix a value of \(S_{10,1}, S_{10,2}, S_{10,3}, S_{10,4}\) and estimate the average number of solutions that we expect for the upper (smaller) execution matrix. Namely, we calculate the expected number of action vectors \(a_1, \ldots, a_0\) which transform \(S_{10,1}, S_{10,2}, S_{10,3}, S_{10,4}\) into \(S_{10,1}, S_{10,2}, S_{10,3}, S_{10,4}\). First, we notice that the number of possible action vectors \(a_1, \ldots, a_0\) is about \(2^{39}\). Each such action vector transforms \(S_{10,1}, S_{10,2}, S_{10,3}, S_{10,4}\) into an arbitrary partial state which matches \(S_{10,1}, S_{10,2}, S_{10,3}, S_{10,4}\) with probability of about \(1/(2n) \approx 2^{-18.5}\) (which is inverse-proportional to the number of possible values of \(S_{10,1}, S_{10,2}, S_{10,3}, S_{10,4}\)). Thus, the expected number of solution (that we store in our sorted list) is \(2^{20} \cdot 2^{-18.5} = 2^{20.5}\).

In general, the time complexity of the MITM algorithm is about square root of the search space, and thus its time complexity on the upper execution matrix is about \(2^{20.5^2}/2 = 2^{19.5}\). However, since in this case we could not split the problem into two parts of exactly equal sizes, we expect \(2^{20.5}\) solutions (which we enumerate and store), and thus its time complexity is slightly increased to \(2^{21.5}\). This is also the expected time complexity of the MITM algorithm on the bottom part (although here we do not store the solutions, but immediately check each one of them). Since we have an outer loop which we execute \(2^{21.5} \approx 2^{18.5}\) times, the expected time complexity of the full algorithm is about \(2^{18.5+20.5} = 2^{39}\). The expected memory complexity is \(2^{20.5}\), required in order to store the solutions for the MITM on the upper part (note that we reuse this memory for each guess of \(S_{10,1}, S_{10,2}, S_{10,3}, S_{10,4}\)).

4. IMPROVED DISSECTION TECHNIQUES

A closer look at the algorithm presented in Section 3 reveals that the algorithm treats the top and bottom parts of the execution matrix differently. Indeed, while the suggestions from the top part are stored in a table (\(L_{19}\) in the example of Figure 6), the suggestions from the bottom part are checked on-the-fly against the table values. As a result, while the number of suggestions in the top part is bounded from above by the size of the memory allowed for the algorithm, the number of suggestions from the bottom part can be arbitrarily large and generated on-the-fly in an arbitrary order.

This suggests that an asymmetric division of the execution matrix, in which the bottom part is significantly bigger than the top part, may lead to better performance of the algorithm.

In this section we show that this is indeed the case. As the algorithm for untangling the Rubik’s cube presented in Section 3 is already practical on a PC so that there is no significant value in further improving it, we choose another classical search problem, known as the combinatorial partition problem to be our running example in this section.

The problem is defined as follows. We are given a set of \(n\) integers, \(U = \{x_1, x_2, \ldots, x_n\}\). Our goal is to partition \(U\) into two complementary subsets \(U_1, U_2\) whose elements sum up to the same number, i.e.,

\[
\sum_{x_i \in U_1} x_i = \sum_{x_j \in U_2} x_j.
\]  

(1)

The combinatorial partition problem is known to be NP-complete [4], and hence, one cannot expect a sub-exponential solution in general. Nevertheless, there are various techniques which allow to find a solution efficiently in various cases, especially when there exist many partitions \((U_1, U_2)\) which satisfy Equation (1). We thus consider the “hardest” instance of the problem, in which each of the \(x_i\),’s is of \(n\) digits in binary representation (i.e., \(x_i \approx 2^n\)). In this case, at most a few solutions \((U_1, U_2)\) are expected to exist, and no sub-exponential algorithms for the problem are known. For the sake of simplicity, we focus on the modular variant of the problem, in which Equation (1) is slightly altered to

\[
\sum_{x_i \in U_1} x_i \equiv \sum_{x_j \in U_2} x_j \quad (\text{mod} 2^n).
\]  

(2)

As a specific numeric example, consider the case \(n = 112\). In this case, checking all \(2^{112}\) possible partitions is, of course, completely infeasible. The standard meet-in-the-middle algorithm allows to reduce the time complexity to \(2^{56}\), but it increases the space complexity to \(2^{56}\) which is currently infeasible. As we show below, the problem can be represented as a bicomposite problem, and hence, the technique of Section 3 can be applied to obtain the better tradeoff of \(T = 2^{28}\) and \(S = 2^{28}\). While these numbers are almost practical, the
relatively large amount of required memory disallows the use of FPGA’s in the computation, which makes it barely feasible. We show below that by an asymmetric variant of the dissection algorithm, we are able to obtain the complexities $T = 2^n$ (which is only 2^n times larger) and $S = 2^n$ (which is 2^n time smaller), which allow for a significantly faster computation using memory-constrained FPGA’s. Note that the asymmetric dissection algorithm outperforms the symmetric one by a factor of sixteen according to the complexity measure $S \cdot T$ (the complexities are $2^{2n}$ vs. $2^{4n}$). Such a factor, while not extremely big, can make a difference in practical scenarios.

### 4.1 Representing Combinatorial Partition as a Bicomposite Search Problem

In order to apply dissection algorithms to the combinatorial partition problem, we have to find a way to represent it as a bicomposite search problem.

First, we represent it as a composite problem. We treat the problem of choosing the partition $(U_1, U_2)$ as a sequence of $n$ atomic decisions, where the $i$th decision is whether to assign $x_i \in U_1$ or $x_i \in U_2$. We introduce a counter $C$ which is initially set to zero, and then at the $i$th step, if the choice is $x_i \in U_1$ then $C$ is replaced by $C + x_i (\mod 2^n)$, and if the choice is $x_i \in U_2$, $C$ is replaced by $C - x_i (\mod 2^n)$. Note that the value of $C$ after the $n$th step is $\sum_{x_i \in U_1} x_i - \sum_{x_i \in U_2} x_i (\mod 2^n)$, and hence, the sequence of choices leads to the desired solution if and only if the final value of $C$ is zero.

In this representation, the partition problem has all the elements of a composite problem: an initial state ($C_{initial} = 0$), a final state ($C_{final} = 0$), and a sequence of $n$ steps, such that in each step, we have to choose one of two possible atomic actions. Our goal is to find a sequence of choices which leads from the initial state to the final state. In terms of the execution matrix, we define $S_i$ to be the value of $C$ after the $i$th step (which is an $n$-bit binary number), and $a_i$ to be the action transforming $S_{i-1}$ to $S_i$, whose possible values are either $C \leftarrow C + x_i (\mod 2^n)$ or $C \leftarrow C - x_i (\mod 2^n)$.

The second step is to represent the problem as a bicomposite problem. The main observation we use here is the fact that for any two integers $a, b$, the $m$th least significant bit of $a + b (\mod 2^n)$ depends only on the $m$ least significant bits of $a$ and $b$ (and not on their other digits). Hence, if we know the $m$ LSBs of $S_{i-1}$ and the action $a_i$, we can compute the $m$ LSBs of $S_i$.

Using this observation, we define $S_{i,j}$ to be the $j$th least significant bit of $\sum_{x_i \in U_1} x_i - \sum_{x_i \in U_2} x_i (\mod 2^n)$, and if the choice is $x_i \in U_2$, $C$ is replaced by $C - x_i (\mod 2^n)$.
significant bit of $S_i$. This leads to an $n$-by-$n$ execution matrix $S_{i,j}$ for $i,j \in 1,2,\ldots,n$ with the property that if we choose any rectangle within the execution matrix which includes the rightmost column of the matrix, knowledge of the substates $S_i^{j-1}, S_i^{j+1}, \ldots, S_i^{n-1}$ along its top edge and knowledge of the actions $a_1, a_1+1, \ldots, a_{2^j}$ to its right suffices in order to compute the substates $S_i^1, S_i^2, \ldots, S_i^{2^j}$ along its bottom edge.

Note that the condition satisfied by our execution matrix is weaker than the condition given in the definition of a bicomposite problem, since in our case, the “rectangle” property holds only for rectangles of a certain kind and not for all rectangles. However, as we show in the next subsection, even this weaker property is sufficient for applying all dissection algorithms we present.²

4.2 Dissection Algorithm for the Combinatorial Partition Problem

The basic idea in the algorithm is to divide the state matrix into seven(!) parts of $n/7$ steps each, where three parts belong to the top part $S^i$ and four parts belong to the bottom part $S^j$. The partition is obtained by enumerating the $2n/7$ least significant bits of the state $S_{n/7}$.

For each value $v$ of these bits, we perform a simple meet-in-the-middle algorithm in the top part, which yields about $2^{3n/7}$ possible combinations of actions $a_1, a_2, \ldots, a_{2n/7}$ which lead to a state $S_{n/7}$ whose $2n/7$ LSBS equal to the vector $v$. For each of these combinations, we compute the full value of the state $S_{n/7}$. The resulting values of $S_{n/7}$ are stored in a table, along with the corresponding combinations of $a_1, a_2, \ldots, a_{2n/7}$.

Then we consider the bottom part, and apply to it the dissection algorithm described in Section 3 (thus, dividing it into four chunks of $n/7$ steps each). This results in $2^{2n/7}$ possible combinations of actions $a_{3n/7}, a_{3n/7}+1, a_{3n/7}+2, \ldots, a_4$, which lead (in the inverse direction) to a state $S_{3n/7}$ whose $2n/7$ LSBS equal to the vector $v$. For each of these combinations, we compute the full value $S_{3n/7}$ and compare it to the values in the table. If a match is found, this means that the corresponding sequences $\{a_1, a_2, \ldots, a_{2n/7}\}$ and $a_{3n/7}, a_{3n/7}+1, a_{3n/7}+2, \ldots, a_4$ match to yield a solution of the problem. Note that if a solution exists, then our method must find it, since it actually goes over all possible combinations of actions (though, in a sophisticated way). The pseudo-code of the algorithm is given in Figure 7.

The memory complexity of the algorithm is $O(2^{n/7})$, as both the standard meet-in-the-middle algorithm for the top part and the dissection algorithm for the bottom part have this complexity: (For the bottom algorithm, the complexity is $(2^{4n/7})^{1/4} = 2^{n/7}$.)

The time complexity is $2^{4n/7}$. Indeed, the enumeration in the state $S_{3n/7}$ is performed over $2^{2n/7}$ values, both the standard meet-in-the-middle algorithm for the top part and the dissection algorithm for the bottom part require $2^{2n/7}$ steps, and the remaining $2^{2n/7}$ possible combinations of $a_{3n/7}, a_{3n/7}+1, a_{3n/7}+2, \ldots, a_4$ are checked instantly. This leads to time complexity of $2^{2n/7} \cdot 2^{2n/7} = 2^{4n/7}$.

²For the sake of simplicity, we disregard the issue of carries. We note that in order to know all the carries required to execute our dissection algorithms, we guess the values of $S_{i,j}$ from the least significant bit to the most significant bit. For more details, refer to the extended version of this paper [2].

In the special case $n = 112$, each of the seven chunks consists of 16 steps, the enumeration is performed on the 28 least significant bits of the state $S_{12}$, the memory complexity is $2^{4n/7} = 2^{64}$, and the time complexity is $2^{n/7} = 2^{32}$.

4.3 Advanced Dissection Algorithms

The algorithms presented in Section 3 and in this section are the two simplest dissection algorithms, which demonstrate the general idea behind the technique. In the extended version of the paper [2] we present more advanced dissection algorithms, which include division of the matrix to “exotic” numbers of parts, such as 11 and 29, and show the optimality of such choices within our general framework.

So far we only considered search algorithms which are not allowed to fail (i.e., if there are any solutions to the problem then our algorithm will always find all of them, but its running time may be longer than expected if the instances are not randomly chosen or if the number of solutions is too large). In [2], we also consider algorithms which may fail to find a solution with a small probability, and show how to improve the efficiency of our algorithms in this case by combining them with a classical technique called parallel collision search, devised by Wiener and van Oorschot in 1996 [7].

5. CONCLUSIONS

In this paper we introduced the notion of bicomposite search problems, and developed new types of algorithmic techniques called dissection algorithms in order to solve them with improved time and space complexities. We demonstrated how to use these techniques by applying them to two standard types of problems (Rubik’s cube and combinatorial partitions). However, some of the most exciting applications of these techniques are in cryptanalysis, which is beyond the scope of this paper. For example, many banks are still using a legacy cryptographic technique called triple-DES, which encrypts sensitive financial data by encrypting it three times with three independent keys. A natural question is whether using quadruple-DES (which encrypts the data four times with four independent keys) would offer significantly more security due to its longer key and more complicated encryption process. By using our new dissection techniques, we can show the surprising result that finding the full key of quadruple-DES could be achieved with the same time and space complexities as finding the full key of the simpler triple-DES encryption scheme, and thus there is no security advantage in upgrading triple-DES to quadruple-DES.

Acknowledgements: The second author was supported in part by the Israel Science Foundation through grant No. 827/12, and in part by the German-Israeli Foundation for Scientific Research and Development through grant No. 2282-2222.6/2011. The third author was supported by the Alon Fellowship.

6. REFERENCES

[2] I. Dinur, O. Dunkelman, N. Keller, and A. Shamir. Efficient Dissection of Composite Problems, with Applications to Cryptanalysis, Knapsacks, and
Algorithm Dissect7-Partition

Input: $U = \{x_1, x_2, \ldots, x_n\}$
for all $S_{5n',1}, S_{5n',2}, \ldots, S_{5n',2n'}$ do
  for all obtained $a_1, a_2, \ldots, a_{5n'}$ do
    Compute the $5n'$ MSBs of $S_{5n'} = a_{5n'}(\cdots(a_2(a_1(S_0)))\cdots)$
    Store $(S_{5n',1}, a_1, a_2, \ldots, a_{5n'})$ in $L_{5n'}$
  Sort $L_{5n'}$ according to the values of $S_{5n'}$
for all $S_{5n',1}, S_{5n',2}, \ldots, S_{5n',n'}$ do
  call PartialMITM($S_{5n',1}, S_{5n',2}, \ldots, S_{5n',n'}, S_{5n',1}, S_{5n',2}, \ldots, S_{5n',n'}, 2n'$)
  for all obtained $a_1, a_2, \ldots, a_{5n'}$ do
    Compute the $n'$ bits $S_{5n',n'+1}, S_{5n',n'+2}, \ldots, S_{5n',2n'} = a_{5n'}(\cdots(a_{n'+2}(a_{n'+1}(S_{5n',n'+1}, S_{5n',n'+2}, \ldots, S_{5n',2n'})))\cdots)$
    Store $(S_{5n',1}, a_1, a_2, \ldots, a_{5n'})$ in $L_{5n'}$
  Sort $L_{5n'}$ according to the values of $S_{5n'}$
for all obtained $a_{n'+1}, a_{5n'+2}, \ldots, a_{7n'}$ do
  Compute $S_{5n',n'+1}, S_{5n',n'+2}, \ldots, S_{5n',2n'} = a_{5n'+1}(\cdots(a_{n'+2}(a_{n'+1}(S_{n'})))\cdots)$
  Search for $S_{5n'}$ in $L_{5n'}$
  if $S_{5n'}$ value is found then
    obtain $a_{n'+1}, a_{5n'+2}, \ldots, a_{7n'}$ from $L_{5n'}$
    for all obtained $a_{7n'+1}, a_{7n'+2}, \ldots, a_{10n'}$ do
      Compute $S_{5n',2n'+1}, S_{5n',2n'+2}, \ldots, S_{5n',7n'} = a_{7n'+1}(\cdots(a_{n'+2}(a_{n'+1}(S_{n'})))\cdots)$
      Search for $S_{5n'}$ in $L_{5n'}$
      if $S_{5n'}$ value is found then
        obtain $a_1, a_2, \ldots, a_{5n'}$ from $L_{5n'}$
        return $a_1, a_2, \ldots, a_{10n'}$ as a solution

Procedure PartialMITM

Input: A partial state $S_0, 2n'$, a partial state $S_{(t+1)n'}, S_{(t+1)n'}, 2n'$, and “distance” $(t+1)n'$ for all $a_{1}, a_{2}, \ldots, a_{n'}$
  do
    Compute $S_{n',1}, S_{n',2}, \ldots, S_{n',t'n'} = a_{n'}(\cdots(a_2(a_1(S_0, S_{2n'})))\cdots)$
    Store $(S_{n',1}, S_{n',2}, \ldots, S_{n',t'n'}, a_1, a_2, \ldots, a_{n'})$ in a list $L_{n'}$
    Sort $L_{n'}$ according to the values of $S_{n',1}, S_{n',2}, \ldots, S_{n',t'n'}$
  for all $a_{n'+1}, a_{n'+2}, \ldots, a_{(t+1)n'}$ do
    Compute $S_{n',1}, S_{n',2}, \ldots, S_{n',t'n'} = a_{n'+1}(\cdots(a_{(t+1)n'}(a_{n'+2}(a_{n'+1}(S_{(t+1)n'}, S_{(t+1)n'}, 2n'))))\cdots)$
    Search for $S_{n',1}, S_{n',2}, \ldots, S_{n',t'n'}$ in $L_{n'}$
    if $S_{n',1}, S_{n',2}, \ldots, S_{n',t'n'}$ are found then
      Obtain the associated $(a_1, a_2, \ldots, a_{n'})$ from $L_{n'}$
      return $a_1, a_2, \ldots, a_{(t+1)n'}$ as a candidate solution

Figure 7: Solving the Partitioning Problem using Dissection into 7


