

The *SetPlayer* System for Symbolic Computation on Power Sets

DAVID BERQUE, RONALD CECCHINI, MARK GOLDBERG, REID RIVENBURGH

Department of Mathematics and Computer Science, DePauw University
Department of Computer Science, Rensselaer Polytechnic Institute

(Received 10 June 2000)

SetPlayer, a software system for symbolic set computation, is described. Its objective is to provide a tool for experimental research in Combinatorics and Graph Theory. The standard means of presenting sets by explicitly listing their elements are extended in *SetPlayer* by allowing expressions describing power sets and their derivatives. The user of *SetPlayer* is permitted to write expressions representing power sets; the system then stores and manipulates these expressions directly without enumerating the corresponding sets. The data structures and algorithms used by *SetPlayer* are described, and the system is also presented from the users' point of view.

1. INTRODUCTION.

The design of symbolic computation software extends back to the nineteen-fifties (Boyle & Caviness (1990); Pavelle & Wang (1988); Wolfram (1988)). Surprisingly, none of the present systems has the ability to *efficiently manipulate* power sets, described *symbolically* with the use of standard mathematical expressions such as 2^A and $\binom{A}{k}$, where A is a set and $k \geq 0$. Many problems in Discrete Mathematics and Graph Theory are formulated in terms of collections of subsets of a given finite set (see Dinitz and Stinson (1992)). A famous example is *Turán's Problem* (see Kim and Roush (1983)) which is often cited as one of the most difficult open problems in Combinatorics. The problem is to evaluate the function $T(n, k, t)$ ($n \geq k \geq t \geq 2$) defined as the minimum number of t -element subsets (t -subsets for short) of an n -set such that every k -subset of the n -set contains at least one of the t -subsets. Turán (Turán (1954); also see Mills (1990) for a survey) established the exact value of $T(n, k, 2)$ for all n and k , but except for some special cases, values of $T(n, k, t)$ ($t > 2$) are not known. It is obvious, that if a programming tool for doing experiments related to this problem stores and manipulates sets explicitly, then the tool would not be practical even for reasonably small input values, say $n = 30$, $k = 10$, $t \geq 3$. On the other hand, a software tool that is capable of *symbolic* set manipulation can be a valuable aid to the researcher who wants to perform experiments in hopes of building the insight that might be needed to solve this type of problem.

Some computer languages allow the user to *write* expressions which describe power sets. In *SETL*, for example, (see Schwartz *et al.* (1986)), 2^A can be written as *POWA*, or by defining a mapping from A to the set $(0, 1)$. However, *SETL's* use of hash tables

to store every set, regardless of how the set was described, makes the system impractical for dealing with $POWA$ even when $|A|$ is small.

Obviously, it is not difficult to design a system which uses expressions to represent sets *internally* as well as at the user interface level. The system can simply store each formula using the same string of characters that the user typed when describing the set. However, this alone does not significantly enhance a set manipulation system. Usually, all set manipulation systems are required to support the *core set operations*, which we define to be the *union*, *intersection*, and *difference* operations. Therefore, if a system represents some sets with expressions, that is *symbolically*, then it should support algorithms which perform the core set operations directly on these expressions. To a large degree, the efficiency of the whole system will depend on the efficiency of these algorithms. It will also depend on the ability of the system to efficiently solve the *core set-theoretic tasks* for all its sets, regardless of the way they are represented. We define these tasks to be computing the *cardinality function*, and deciding the *membership*, *subset*, and *equality predicates*. Many other important tasks can then be performed using the subroutines for the core operations and core set-theoretic tasks.

To summarize, our requirements for symbolic set manipulation system involve the following three areas:

- the use of expressions to *describe* sets;
- the use of expressions to *store* sets internally; and
- the use of algorithms which can *efficiently manipulate* expressions directly (without converting them to an explicit representation).

When designing algorithms for the core operations, one can consider a simple concatenation algorithm, which produces the union, intersection, or difference of two expressions by simply joining the two input expressions together using the appropriate operator.[†] This approach essentially builds a tree in which the leaves are labeled by expressions and the internal vertices are labeled by the set operators. While this algorithm for performing the core operations is quite efficient itself, it can lead to overall system inefficiency. This inefficiency stems from the fact that many different expressions can be used to describe the same set. The structure of such an expression tends to provide relatively little information about the set it represents, but rather gives information about how the set was constructed. Furthermore, when the expressions are developed this way, computing the core set-theoretic functions does not seem to be possible without a complete, or at least partial, enumeration of the symbolically represented sets. Clearly, this largely eliminates the advantages that symbolic set manipulation systems are intended to provide.

In this paper we describe an interactive software system named *SetPlayer*[‡] which attempts to meet the three requirements for symbolic set manipulation described above. The standard means of describing sets by explicitly listing their elements are extended in *SetPlayer* by introducing the expressions $POW(A)$ and $POW.k(A)$. Here $POW(A)$ denotes the set of all subsets of the set A while $POW.k(A)$ denotes the set of all subsets of size k of A . We introduce a data structure, called a *difference tree*, which allows

[†] This approach is used in the latest version of *Cayley* (Butler (1989)).

[‡] The first version of the system, also called *SetPlayer*, (see Goldberg *et al.* (1988)) had no capability to manipulate expressions.

SetPlayer to manipulate symbolically defined sets without explicitly enumerating their contents.

When symbolic means for describing sets (in our case these are the *POW* and *POW.k* operators) are supported by a software system, it is natural to classify the types of expressions that the system can manipulate. We define expressions of rank 0 to be those expressions that are explicit; that is they represent sets without using the *POW* and *POW.k* operators. For $k \geq 1$, an expression of rank k is an expression for which the argument to every occurrence of the operators *POW* and *POW.k* are themselves expressions of rank $< k$. Thus, all expressions of rank $k - 1$ are also of rank k .

Using the conventional explicit representation of sets, it is easy to develop a scheme for canonical set representation. Having a unique, “normal”, representation of a set can be useful in a complicated software system. Obviously, every set represented by an expression of rank ≥ 1 can also be represented explicitly, which is a fundamentally different representation. Thus, when expressions of rank > 0 are used to represent sets, the ability to easily develop a canonical[†] representation for sets is inevitably lost. While the difference trees that *SetPlayer* uses to represent sets are not canonical, their characteristics are considerably more regular than those of the rather arbitrary trees that the concatenation algorithm produces. This compromise enables *SetPlayer* to efficiently compute the core operations and the core set-theoretic tasks regardless of whether the sets are represented explicitly or symbolically.

The *SetPlayer* system deals with expressions of rank 0 and 1 only. Further, sets of rank 0 are restricted to finite sets of integers. Although, the latter restriction made the user interface of the system less flexible, it reduced the amount of programming needed, and allowed us to concentrate our efforts on algorithm development and implementation. The reason for bounding the rank of expressions by 1 was due to theoretical difficulties related to the development of efficient algorithms for manipulating expressions of higher ranks. Obviously, there are many expressions other than *POW* that can be used to represent sets; the development of a comprehensive theory of symbolic set manipulation is a task for the future. Nevertheless, even with all these restrictions, the domain of *SetPlayer* covers many important areas of Mathematics including traditional finite graph theory, and combinatorial design theory.

The data structures and basic algorithms used by *SetPlayer* are outlined in Section 2. Section 3 presents the system as it appears to the user. The fourth section addresses possible avenues for future work.

2. DATA STRUCTURES AND ALGORITHMS

2.1. BASICS

Throughout the paper we use standard set-theoretic terminology. In particular, we use $A - B$ and $A \setminus B$ to represent the difference of sets A and B . It is always true that $A \setminus B = A - B$, but we write $A \setminus B$ only if $B \subseteq A$. If $S = (s_1, s_2, \dots, s_n)$ and $T = (t_1, t_2, \dots, t_m)$ are two sequences of integers, then we say that S is *lexicographically smaller* than T if either $n < m$, or $n = m$ and for the smallest i for which $s_i \neq t_i$, $s_i < t_i$.

Let \mathcal{A} be a collection of subsets $\{A_1, \dots, A_m\}$ of a set U . We call \mathcal{A} *connected*, if no partition $U_1 \cup U_2 = U$ ($U_1 \cap U_2 = \emptyset$) exists such that $U_1 \neq \emptyset; U_2 \neq \emptyset$, and every A_i is a

[†] At least for the straightforward interpretation of “canonical”.

subset of either U_1 or U_2 ($i = 1, \dots, m$). A collection \mathcal{B} is a *subcollection* of \mathcal{A} , if every member of \mathcal{B} is also a member of \mathcal{A} . A *maximal* connected subcollection \mathcal{B} of \mathcal{A} is called a connected component of \mathcal{A} . Obviously, every collection \mathcal{A} is the union of its connected components.

By a rooted (nonempty) binary tree we mean, as usual (see Aho *et al.* (1982)) a tree in which every vertex has either no children, or a left child, or a right child, or both a left and a right child. The subtree rooted at a vertex z is the subtree comprised z together with all its descendants. A branch of a vertex is the subtree rooted at one of the children of the vertex.

The basic data type that *SetPlayer* supports is a set. In *SetPlayer* terminology, a *set* refers to a finite set over the integers. Sets can be defined by the user by explicitly listing integers (or ranges of integers) between a pair of parentheses, or by applying a set operation to previously defined sets (a list of supported set operations follows). *SetPlayer* stores every set internally by listing its members in increasing order. This list is allocated dynamically and is anchored by a header which stores the set's cardinality. Whenever a set or other object is destroyed, the memory it uses is freed.

2.2. DIFFERENCE TREES

In *SetPlayer* terminology, a *family* is a collection of sets. Families can be defined by explicitly listing the member sets between a pair of braces, or by implicitly describing the family with the use of the operators *POW*, *POW.k*, *UPOW*, and *UPOW.k* defined below. Since *SetPlayer* families are simply a special type of mathematical set, families may also be defined by applying standard set-theoretic operations to previously defined families. Internally, *SetPlayer* stores an explicit family as a lexicographically ordered list of sets. The list of sets is allocated dynamically, and is anchored by a header which stores the family's cardinality.

To manage symbolically represented families, we use $UPOW(\{A_1 \dots A_m\})$ (resp. $UPOW.k(\{A_1 \dots A_m\})$) as a shorthand for $\bigcup_{i=1}^m POW(A_i)$ (resp. $\bigcup_{i=1}^m POW.k(A_i)$). To represent these families internally, an additional marker indicating the presence of the operator *UPOW* or *UPOW.k* is attached to the list which represents the explicit family[†] $\mathcal{A} = \{A_1, \dots, A_p\}$. To be able to perform the core set operations and the core set-theoretic tasks on symbolically represented sets without resorting to explicit enumeration, we introduce a data structure called a *difference tree* which is a special type of rooted labeled binary tree. Each leaf of a difference tree is labeled with an expression of the form

$$\mathcal{A} \cup UPOW(\mathcal{B}) \cup UPOW.k(\mathcal{C}) \cup UPOW.l(\mathcal{D}) \dots,$$

where $\mathcal{A}, \mathcal{B}, \mathcal{C}, \mathcal{D} \dots$ are explicit families. \mathcal{A} , $UPOW(\mathcal{B})$, $UPOW.k(\mathcal{C})$, and $UPOW.l(\mathcal{D})$ are called the *constituents* of the label; \mathcal{A} is called the *explicit constituent*, all other constituents are called *symbolic constituents*. Each label is stored internally in a dynamically allocated list. The explicit constituent (if it is present) appears first, followed by the constituent expressed with the *UPOW* operator (if it is present) followed by any constituents represented by the *UPOW.k* operator, in order of increasing value of k .

For each vertex z of a difference tree, the subtree rooted at z represents a family

[†] In the current implementation, the nested use of *POW* is not allowed. Therefore *SetPlayer* can only deal with sets of integers, and sets of sets of integers.

which we denote by \mathcal{F}_z . If z is a leaf, then \mathcal{F}_z is the family described by its label. If z is an internal vertex and z_l and z_r are its left and right children, respectively, then $\mathcal{F}_z = \mathcal{F}_{z_l} \setminus \mathcal{F}_{z_r}$. Thus, by definition, $\mathcal{F}_{z_r} \subseteq \mathcal{F}_{z_l}$. Each vertex z of a difference tree may have an additional label which stores the cardinality of \mathcal{F}_z . This label is stored at vertex z if the cardinality of \mathcal{F}_z has been previously computed. If the system determines that $|\mathcal{F}_z| = 0$, then the subtree rooted at z is deleted from the tree and the remaining tree is rearranged to preserve the properties of difference trees. If \mathcal{S} is a difference tree of height > 1 , then \mathcal{S}_l and \mathcal{S}_r denote its left and right branches respectively; sometimes we write \mathcal{S} as $\mathcal{S}_l \setminus \mathcal{S}_r$.

The following statements are immediate corollaries of the definitions given above.

Statement 1.

$$\begin{aligned} S \in UPOW(\mathcal{F}) &\Leftrightarrow \exists i F_i \in \mathcal{F} \wedge S \subseteq F_i; \\ S \in UPOW.k(\mathcal{F}) &\Leftrightarrow \exists i F_i \in \mathcal{F} \wedge S \subseteq F_i \wedge |S| = k. \end{aligned}$$

Statement 2.

$$\begin{aligned} \text{If } \mathcal{F}' \text{ is an explicitly represented family, then} \\ UPOW(\mathcal{F}) \cap \mathcal{F}' &= \{F \mid F \in \mathcal{F}' \wedge F \in UPOW(\mathcal{F})\}; \\ UPOW.k(\mathcal{F}) \cap \mathcal{F}' &= \{F \mid F \in \mathcal{F}' \wedge F \in UPOW.k(\mathcal{F})\}. \end{aligned}$$

Statement 3.

$$\begin{aligned} UPOW(\mathcal{A}) \cap UPOW(\mathcal{B}) &= UPOW(\mathcal{C}) \text{ where} \\ \mathcal{C} &= \{C : \exists i, j A_i \in \mathcal{A}, B_j \in \mathcal{B}, C = A_i \cap B_j\}; \\ UPOW(\mathcal{A}) \cap UPOW.k(\mathcal{B}) &= UPOW.k(\mathcal{C}) \text{ where} \\ \mathcal{C} &= \{C : \exists i, j A_i \in \mathcal{A}, B_j \in \mathcal{B}, C = A_i \cap B_j, |C| \geq k\}; \\ UPOW.k(\mathcal{A}) \cap UPOW.k(\mathcal{B}) &= UPOW.k(\mathcal{C}) \text{ where} \\ \mathcal{C} &= \{C : \exists i, j A_i \in \mathcal{A}, B_j \in \mathcal{B}, C = A_i \cap B_j, |C| \geq k\}; \\ \text{If } k \neq l, UPOW.k(\mathcal{A}) \cap UPOW.l(\mathcal{B}) &= \emptyset. \end{aligned}$$

Statement 4.

$$\begin{aligned} UPOW(\mathcal{A}) \cup UPOW(\mathcal{B}) &= UPOW(\mathcal{C}) \text{ where } \mathcal{C} = \mathcal{A} \cup \mathcal{B}; \\ UPOW.k(\mathcal{A}) \cup UPOW(\mathcal{B}) &= UPOW.k(\mathcal{C}) \text{ where } \mathcal{C} = \mathcal{A} \cup \mathcal{B}. \end{aligned}$$

Statement 5.

$$\begin{aligned} \text{Let } \mathcal{F} \text{ and } \mathcal{G} \text{ be two families and let } \mathcal{F} &= \mathcal{F}_l \setminus \mathcal{F}_r \text{ and } \mathcal{G} = \mathcal{G}_l \setminus \mathcal{G}_r. \text{ Then} \\ \mathcal{F} \cap \mathcal{G} &= [(\mathcal{F}_l \cap \mathcal{G}_l) \setminus (\mathcal{F}_r \cap \mathcal{G}_l)] \setminus [(\mathcal{F}_l \cap \mathcal{G}_r) \setminus (\mathcal{F}_r \cap \mathcal{G}_r)]. \\ \mathcal{F} - \mathcal{G} &= \mathcal{F} \setminus (\mathcal{F} \cap \mathcal{G}). \\ \mathcal{F} \cup \mathcal{G} &= [(\mathcal{F}_l \cup \mathcal{G}_l) \setminus (\mathcal{G}_r \setminus (\mathcal{G}_r \cap \mathcal{F}_l))] \setminus (\mathcal{F}_r \setminus (\mathcal{F}_r \cap (\mathcal{G}_l \setminus \mathcal{G}_r))) \end{aligned}$$

Whenever the system creates, or modifies a leaf of a difference tree, it automatically applies the following *simplification rules* to the leaf's label:

If \mathcal{F} is the argument to the operator $UPOW$ or $UPOW.k$, $A, B \in \mathcal{F}$, and $A \subseteq B$, then A is removed from \mathcal{F} .

If \mathcal{F} is the argument to the operator $UPOW.k$, $A \in \mathcal{F}$ and $|A| < k$, then A is removed from \mathcal{F} .

If a set belongs to the explicit constituent and also belongs to one of the symbolic constituents of the label, then this set is removed from the explicit constituent.

If \mathcal{F} is the argument to the operator $UPOW$, \mathcal{F}' is the argument to the operator $UPOW.k$, $A \in \mathcal{F}$, $B \in \mathcal{F}'$, and $B \subseteq A$, then B is removed from \mathcal{F}' .

All empty families are removed.

Any occurrence of the expression $UPOW.0(\mathcal{F})$ or $UPOW.1(\mathcal{F})$ is replaced by an equivalent explicit representation.

All like constituents[†] of a leaf are merged into a single constituent so that the sets within the newly created constituent are ordered lexicographically.

A difference tree is simplified using a procedure named *Simplify* which applies the above rules to each leaf of the difference tree until none of the rules are applicable. *Simplify* is called from many of the algorithms for performing the core operations which are presented below. Since the simplification routine is separate from these algorithms, new simplification rules may be easily incorporated into the system as they are developed.

2.3. ALGORITHMS FOR THE CORE OPERATIONS

Below, we describe procedures for performing the core operations directly on difference trees. Formally this means that if \mathcal{A} and \mathcal{B} are two families represented by trees \mathcal{S} and \mathcal{T} , respectively, and \mathcal{C} is the result of applying a particular operation to \mathcal{A} and \mathcal{B} , then the corresponding procedure applied to \mathcal{S} and \mathcal{T} produces a tree \mathcal{R} which represents family \mathcal{C} . The proofs of correctness of these procedures can be easily deduced from statements 1 through 5.

Procedure *Intersection*;

/ Input: \mathcal{S}, \mathcal{T} ; Output: $\mathcal{R} = \mathcal{S} \cap \mathcal{T} \equiv (\mathcal{S}_l - \mathcal{S}_r) \cap \mathcal{T}$ */*

begin

if the heights of \mathcal{S} and \mathcal{T} are both 1 **then**

Form a new label L whose constituents are the pairwise intersections of the constituents in the label of \mathcal{S} with those of \mathcal{T} ;

Let \mathcal{R} be a new tree whose root is a leaf labeled by L ;

return *Simplify*(\mathcal{R});

else

if \mathcal{S} has height > 1 **then**

Construct $\mathcal{R}' = \text{Intersection}(\mathcal{S}_l, \mathcal{T})$;

Construct $\mathcal{R}'' = \text{Intersection}(\mathcal{S}_r, \mathcal{T})$;

return $\mathcal{R} = \mathcal{R}' \setminus \mathcal{R}''$;

else */* \mathcal{T} must have height > 1 . */*

Construct $\mathcal{R}' = \text{Intersection}(\mathcal{T}_l, \mathcal{S})$;

Construct $\mathcal{R}'' = \text{Intersection}(\mathcal{T}_r, \mathcal{S})$;

return $\mathcal{R} = \mathcal{R}' \setminus \mathcal{R}''$;

end if

end if

end

Procedure *Difference*;

/ Input: \mathcal{S}, \mathcal{T} ; Output: $\mathcal{R} = \mathcal{S} - \mathcal{T} \equiv \mathcal{S} \setminus (\mathcal{T} \cap \mathcal{S})$ */*

begin

[†] Two constituents are said to be *like* if they are both explicit, or both expressed with the *UPOW* operator, or both expressed with the *UPOW.k* operator for the same value of k .

```

    Let  $\mathcal{R}'$  be a copy of  $\mathcal{S}$ ;
    Construct  $\mathcal{R}'' = \text{Intersection}(\mathcal{S}, \mathcal{T})$ ;
    return  $\mathcal{R} = \mathcal{R}' \setminus \mathcal{R}''$ ;
end

```

```

Procedure Union;
/* Input:  $\mathcal{S}, \mathcal{T}$ ;   Output:  $\mathcal{R} = \mathcal{S} \cup \mathcal{T} \equiv (\mathcal{S}_l \setminus \mathcal{S}_r) \cup \mathcal{T} = (\mathcal{S}_l \cup \mathcal{T}) \setminus (\mathcal{S}_r \setminus (\mathcal{T} \cap \mathcal{S}_r))$ 
*/
begin
  if the heights of  $\mathcal{S}$  and  $\mathcal{T}$  are both 1 then
    Form a new label  $L$  by concatenating
    the label of  $\mathcal{S}$  with that of  $\mathcal{T}$ ;
    Let  $\mathcal{R}$  be a new tree whose root is a leaf labeled by  $L$ ;
    return Simplify( $\mathcal{R}$ );
  else
    if  $\mathcal{S}$  has height  $> 1$  then
      Construct  $\mathcal{R}' = \text{Union}(\mathcal{S}_l, \mathcal{T})$ ;
      Construct  $\mathcal{R}'' = \mathcal{S}_r \setminus \text{Intersection}(\mathcal{S}_r, \mathcal{T})$ ;
      return  $\mathcal{R} = \mathcal{R}' \setminus \mathcal{R}''$ ;
    else /*  $\mathcal{T}$  must have height  $> 1$ . */
      Construct  $\mathcal{R}' = \text{Union}(\mathcal{T}_l, \mathcal{S})$ ;
      Construct  $\mathcal{R}'' = \mathcal{T}_r \setminus \text{Intersection}(\mathcal{T}_r, \mathcal{S})$ ;
      return  $\mathcal{R} = \mathcal{R}' \setminus \mathcal{R}''$ ;
    end if
  end if
end

```

2.4. ALGORITHMS FOR THE CORE SET-THEORETIC TASKS

Let \mathcal{T} be a difference tree with height greater than one, let w be the root of \mathcal{T} and let the left and right children of w be w_l and w_r respectively. It follows immediately from the definition of difference trees that $|\mathcal{T}_w| = |\mathcal{T}_{w_l}| - |\mathcal{T}_{w_r}|$. Therefore the problem of computing the cardinality of an arbitrary difference tree can be reduced to the problem of computing the cardinality of the leaves of that tree. Consider an arbitrary leaf and let its label be

$$\mathcal{A} \cup UPOW(\mathcal{B}) \cup \bigcup_{i=1}^m UPOW.i(\mathcal{C}_i).$$

The simplification rules guarantee that if $S \in \mathcal{A}$ then S is not a member of the family represented by any symbolic constituent. It is obvious that when $k \neq l$ the families represented by $UPOW.k(\mathcal{C}_i)$ and $UPOW.l(\mathcal{C}_j)$ are disjoint. Therefore the cardinality of the family represented by the label is equal to

$$|\mathcal{A}| + |UPOW(\mathcal{B})| + \sum_{i=1}^m |UPOW.i(\mathcal{C}_i)| - \sum_{i=1}^m |UPOW.i(\mathcal{B}) \cap UPOW.i(\mathcal{C}_i)|.$$

The preceding expression, together with statement 3, shows that computing the cardinality of the family represented by an arbitrary label can be reduced to the problem of computing the cardinality of families represented by expressions of the form $UPOW(\mathcal{A})$ and $UPOW.k(\mathcal{A})$, where \mathcal{A} is an arbitrary explicit family.

To describe an algorithm for calculating these cardinalities, we first define functions $subs$, $subs.k$, trs [†], and $trs.k$. Below, $\mathcal{A} = \{A_1, A_2, \dots, A_m\}$ is an explicit family, U is a set, and $k \geq 0$ is an integer.

$$subs(\mathcal{A}) = |UPOW(\mathcal{A})|; \quad subs.k(\mathcal{A}) = |UPOW.k(\mathcal{A})|;$$

$$trs(U; \mathcal{A}) = |\{X : X \subseteq U \wedge \forall_{i=1}^m (X \cap A_i \neq \emptyset)\}|$$

$$trs.k(U; \mathcal{A}) = |\{X : X \subseteq U \wedge |X| = k \wedge \forall_{i=1}^m (X \cap A_i \neq \emptyset)\}|.$$

Let $U = \bigcup_{i=1}^m A_i$, $\overline{A}_i = U - A_i$ ($i = 1, \dots, m$), and $\overline{\mathcal{A}} = \{\overline{A}_1, \dots, \overline{A}_m\}$.

Statement 4.1

$$subs(\mathcal{A}) = 2^{|U|} - trs(U; \overline{\mathcal{A}}) \quad subs.k(\mathcal{A}) = \binom{|U|}{k} - trs.k(U; \overline{\mathcal{A}}).$$

Proof. Indeed, if a set $S \subset A_i$, for some $i = 1, \dots, m$ then $S \cap (U - A_i) = \emptyset$, implying that S is not counted by $trs(U; \overline{\mathcal{A}})$ (resp. by $trs.k(U; \overline{\mathcal{A}})$). The reverse is obviously true as well. \blacksquare

Thus, computing $subs$ (resp. $subs.k$) is computationally equivalent to computing trs (resp. $trs.k$). Furthermore, computing $trs(U; \mathcal{A})$ (resp. $trs.k(U; \mathcal{A})$) when each set belonging to \mathcal{A} has cardinality two, is equivalent to solving the *Monotone 2-SAT* problem (resp. to counting the number of vertex covers of size k in a graph). Both problems are known to be #P-complete (see Garey and Johnson (1979) and Valiant (1979), respectively). Therefore, the following statement holds.

Statement 4.2

The problems of computing $subs$ and $subs.k$ are #P-complete. \blacksquare

Thus, unless $P=NP$, algorithms for computing $subs(\mathcal{A})$ and $subs.k(\mathcal{A})$ (where $\mathcal{A} = \{A_1, A_2, \dots, A_m\}$) must be worst case exponential with respect to the length $L = \sum_{i=1}^m |A_i|$ of the symbolic representation of $\bigcup_{i=1}^m POW(A_i)$. On the other hand, without using a symbolic representation for the family, the running time of computing all the core set-theoretic functions, being linear in $\sum_{i=1}^m 2^{|A_i|}$, is exponential for *all* inputs. From our experiments, it appears that an additional advantage of symbolic computation is that the base of the exponential function is reasonably low. The exact evaluation of this constant seems to be a difficult problem; we address it in a different paper (see, Goldberg, Spencer, and Berque (1991)).

The algorithms *SetPlayer* uses to compute trs and $trs.k$ have two phases: the *replacement* phase and the *recursive* phase.

First, the *replacement* phase checks to see if the problem instance can be replaced by a simpler instance, or can be calculated directly. The steps performed by the replacement

[†] The name trs comes from the word *transversal*.

phase are based on the following statements. In what follows, let $\mathcal{F} = \{F_1, F_2, \dots, F_m\}$ for some $m \geq 0$:

Statement 4.3

$$\begin{aligned} |F| = 0 &\Rightarrow trs(U; \mathcal{F}) = 2^{|U|} \text{ and,} \\ |F| = 0 &\Rightarrow trs.k(U; \mathcal{F}) = \binom{|U|}{k}. \end{aligned}$$

Statement 4.4

$$\exists_{i=1}^m (|F_i| = 0) \Rightarrow trs(U; \mathcal{F}) = trs.k(U; \mathcal{F}) = 0.$$

Statement 4.5

Define $\mathcal{F}_{\overline{v}}$ to be the family consisting of those sets in \mathcal{F} that do not contain element v . Then $\exists_{i=1}^m (|F_i| = 1 \wedge v \in F_i) \Rightarrow trs(U; \mathcal{F}) = trs(U - \{v\}; \mathcal{F}_{\overline{v}})$ and $\exists_{i=1}^m (|F_i| = 1 \wedge v \in F_i) \Rightarrow trs.k(U; \mathcal{F}) = trs.k(U - \{v\}; \mathcal{F}_{\overline{v}})$.

Statement 4.6

If every set $S \in \mathcal{F}$ has cardinality two and every $x \in U$ belongs to at most two members of \mathcal{F} , then $trs(U; \mathcal{F}) = DegTwoGraph(U; \mathcal{F})$ and $trs.k(U; \mathcal{F}) = DegTwoGraph.k(U; \mathcal{F})$. The functions $DegTwoGraph$ and $DegTwoGraph.k$ are discussed below.

Statement 4.7

If $k \leq 1$ or $k \geq |U| - 1$, $trs.k(U; \mathcal{F})$ is computed directly.

Observe that the conditions outlined in Statement 4.6 ensure that when the functions $DegTwoGraph(U; \mathcal{F})$ and $DegTwoGraph.k(U; \mathcal{F})$ are computed, the family \mathcal{F} represents a graph of maximum degree two, which means that every connected component of this graph is either a path or a cycle. In this case, the problem is reduced to that on individual paths and cycles. It turns out, that for the latter, there are closed form expressions for computing trs and $trs.k$. For example,

Statement 4.8

Let F be a Fibonacci-type sequence defined by $F(0) = 2$, $F(1) = 3$ and, for all $i \geq 2$, $F(i) = F(i-1) + F(i-2)$. Then, for all $n > 0$ if $P_n = (V, E)$ is a path of length n , then $trs(V; E) = F(n)$.

Proof. If $n = 0$, then $trs(V; E) = 2 = F(0)$. Similarly, when P is a path of length 1, $trs(V; E) = 3 = F(1)$. Now let $n \geq 2$, v be an endpoint of P , and u be adjacent to v . Then $trs(V; E)$ equals the number of transversals of P that contain v plus the number of those that do not contain v , and therefore contain u . This is equivalent to the number of transversals of P_{n-1} plus that of P_{n-2} , which is $F(n-1) + F(n-2)$. \blacksquare

The *recursive phase* of the algorithms starts when no replacement rule is applicable. Two ways to split the problem to smaller subproblems are considered. First, the procedure checks to see whether the family is disconnected. If there are $p \geq 2$ connected components of \mathcal{A} , then the problem $(U; \mathcal{A})$ is reduced to the subproblems $(U_t; \mathcal{A}_t)$ ($t = 1, \dots, p$) and the answer to the original problem is the product of the answers to the subproblems.

Now, let $\mathcal{A}_i = \mathcal{A} - \{A_i\}$ and $\mathcal{A}^{(i)} = \{A_1 - A_i, \dots, A_{i-1} - A_i, A_{i+1} - A_i, \dots, A_m - A_i\}$. Then the second type of reduction is based on the following equalities: for all $i = 1, \dots, m$,

$$trs(U; \mathcal{A}) = trs(U; \mathcal{A}_i) - trs(U - A_i; \mathcal{A}^{(i)})$$

and

$$trs.k(U; \mathcal{A}) = trs.k(U; \mathcal{A}_i) - trs.k(U - A_i; \mathcal{A}^{(i)}).$$

Several strategies can be used in the selection of A_i as the “pivot”. The basic idea of the subroutine *FindPivot* implemented in *SetPlayer* is to select an A_i that intersects as many other members of \mathcal{A} as possible.

```

Procedure trs;
  /* Input:  $U; \mathcal{A} = \{A_1, \dots, A_m\}$ ; Output:  $trs(U; \mathcal{A})$  */

  begin
     $(U; \mathcal{A}) = \text{Replace}(U; \mathcal{A})$ ;
    if  $\mathcal{A}$  is disconnected and
       $A_1 \dots A_p$  are the component of  $\mathcal{A}$  then
         $trs(U; \mathcal{A}) = \prod trs(U_i; A_i)$ ;
      else
         $A_i = \text{FindPivot}(U; \mathcal{A})$ ;
         $trs(U; \mathcal{A}) = trs(U; A_i) - trs(U - A_i; \mathcal{A}^{(i)})$ ;
      end if
    end
  
```

The procedure for *trs.k* is similar to *trs*. This completes the outline of the algorithm used by *SetPlayer* for computing the cardinality of an arbitrary symbolic family. We conclude this section with outlines of procedures for computing the core set-theoretic functions on symbolically represented families. Note that some of these procedures rely on computing the cardinality of the symbolically represented families. This need stems from the fact that the only algorithm we know for determining whether a symbolically represented family is empty involves computing the cardinality of the family.

```

Procedure Equality;
  /*Input:  $S, \mathcal{T}$ ; Output: True if  $S = \mathcal{T}$  else False */

  begin
    if  $|S| = |S \cap \mathcal{T}| = |\mathcal{T}|$  then
      return true;
    else
      return false;
    end if
  end
  
```

```

Procedure Subset;
  /* Input:  $S, \mathcal{T}$ ; Output: True if  $S \subseteq \mathcal{T}$  else False */

  begin
    if  $|S| > |\mathcal{T}|$  then
      return False;
    else if  $|S - S \cap \mathcal{T}| = 0$  then
      return True;
    else
  
```

```

        return False;
    end if
end if
end

```

Procedure *Member*;

/ Input: S, \mathcal{T} ; Output: true if $S \in \mathcal{T}$ else false */*

```

begin
  if  $\mathcal{T}$  has height 1 then
    /* Use statement 1 to check the following: */
    if  $S \in$  at least one constituent of the label of  $\mathcal{T}$  then
      return true;
    else
      return false;
    end if
  else
    return  $Member(S, \mathcal{T}_l) \wedge \neg Member(S, \mathcal{T}_r)$ ;
  end if
end

```

3. THE USERS' VIEW

3.1. PRELIMINARIES AND DATA TYPES

SetPlayer supports more than fifty commands which can be executed from both a *menu-driven* and *command-driven* interface. The combination of these interfaces allows *SetPlayer* to serve the needs of both novice and experienced users.

Sets are described by explicitly listing their members between parentheses, by specifying a range of elements like (8.47), or by combining these techniques as in (6, 8, 9.44, 17, 1.4). Families are described by listing the sets which they contain, or by using the operators $POW(S)$, $POW.k(S)$, $UPOW(\mathcal{A})$ or $UPOW.k(\mathcal{A})$ where S is a set and $\mathcal{A} = \{A_1, A_2, \dots, A_m\}$ is an explicit family. New sets and families can then be constructed by applying the core set operators to existing sets and families.

As an additional convenience, *SetPlayer* allows the user to write expressions of the form $UPOW(\mathcal{A}; \mathcal{B})$ (resp. $UPOW.k(\mathcal{A}; \mathcal{B})$) where $\mathcal{A} = \{A_1, A_2, \dots, A_m\}$ and $\{B = B_1, B_2, \dots, B_n\}$ are explicit families. This represents the set of all X 's (respectively all X 's with $|X| = k$) such that for at least one $m \in [1, i]$, $X \subseteq A_m$ and for each B_n , $X \cap B_n \neq \emptyset$. *SetPlayer* automatically converts this type of expression into a standard difference tree. For example, if $G = (V, E)$ is a graph, then the expression $UPOW.10(V, E)$ represents the collection of all vertex covers of size 10 of G (all 10-subsets of the vertex set that intersect with every edge in the edge set).

The collection of *SetPlayer*'s main commands is shown in the following menu; notice, that some of the commands have their own sub-menus. The complete description of *SetPlayer*'s commands can be found in Berque et al (1991).

```

E N V I R O N M E N T   C O M M A N D S :
MENU                   COMMAND          LIBRARY          EXIT
HELP                   HISTORY           SYSTEM           SHORTEN
CREATE                 COPY             DESTROY          RENAME
NAMES                 PRINT            LIST             TREELIST
READ                  WRITE            SAVE             LOAD
DISPLAY              ATTRIBUTES      DEFAULT          CNTR-C

O P E R A T I O N S :
INSERT                DELETE           UNION            INTERSECT
DIFFERENCE           SYMMETRIC       INDUCE           REDUCE
CONTRACT             DUAL            MOVE             MAGNIFY
FUNION               FINTERSECTION  EXTRACT          INDEX
INCIDENT             ADJACENT        POWER            UPOWER
RELABEL              APPLY           GENERATE
RANGE                DOMAIN          MULTIPLY         APPEND
SET                  TUPLE           REVERSE          SORT
ORBITS               CERTIFICATE     PERMUTE          CHOOSE

P R E D I C A T E S & F U N C T I O N S :
EQUALITY             MEMBERSHIP      SUBSET           PLANAR
ISOMORPHISM
CARD                 PARTITION       TRS              GCD/LCM

```

3.2. OVERVIEW OF SELECTED COMMANDS

In the following descriptions, the term *entity* refers to the *SetPlayer* data types *integer*, *set*, and *family*. Unless otherwise stated, commands that deal with families can be applied to families that are represented either explicitly or symbolically. Finally, note that the names of sets are written using Roman letters while the names of families are written using caligraphic letters. For example, B will be used to name a set while \mathcal{B} will be used to name a family.

ENVIRONMENT COMMANDS.

Menu. When executed from the menu driven interface, this command displays a menu of *SetPlayer* commands. When executed from the command driven interface, this command causes the system to switch to the menu driven interface.

Command. When executed from the menu driven interface, this command causes the system to switch to the command driven interface.

Shorten. Shortens the length of the formula representing a user-specified symbolically represented family (if a formula with a shorter length can be found). This operation can be time consuming so it is only carried-out at the request of the user (although it would be possible to implement the system so that this operation was carried out automatically whenever a sufficiently long formula was detected).

Create. Returns a newly created entity, initialized to a user-specified value.

Copy. Returns a copy of a user-specified entity.

Destroy. Destroys a user-specified entity by freeing the memory associated with it and allowing its name to be re-used.

Rename. Renames a user-specified entity, thereby allowing its old name to be re-used.

Print. Displays the value of a user-specified entity.

TreeList. Displays the structure of the formula used to represent a user-specified symbolically represented family.

Write. Writes a user-specified set or family to a disk file.

Read. Reads a disk file containing a set or a family and creates the corresponding entity with a user-specified name.

OPERATIONS:

Insert. Returns a new entity that results from inserting a user-specified object into an existing entity specified by the user.

Delete. Returns a new entity that results from deleting a user-specified object from an existing entity specified by user.

Union. Returns the union of a collection of user-specified sets or families.

Intersect. Returns the intersection of a collection of user-specified sets or families.

Difference. Returns the difference of two user-specified sets or families.

Symmetric. Returns the symmetric difference of two user-specified sets or families.

Funion. For a given set-family $\mathcal{F} = \{F_1, \dots, F_m\}$, returns the set $U = \bigcup_{i=1}^m F_i$.

Fintersection. For a given set-family $\mathcal{F} = \{F_1, \dots, F_m\}$, returns the set $S = \bigcap_{i=1}^m F_i$.

Pow. The input is a set S and an optional integer k . If k is not supplied, the command returns the power set of S , that is, 2^S . Otherwise, the command returns $\binom{S}{k}$. In either case, the resulting family is represented symbolically.

UPow. The input is a family \mathcal{A} , an optional family \mathcal{B} , and an optional integer k . Both of the input families must be represented explicitly. Depending on the combination of arguments that are supplied, the command returns the family represented by one of the following expressions: $UPOW(\mathcal{A})$, $UPOW.k(\mathcal{A})$, $UPOW(\mathcal{A}, \mathcal{B})$, or $UPOW.k(\mathcal{A}, \mathcal{B})$. The meaning of these expressions is described below:

- $UPOW(\mathcal{A}) = \bigcup_{A_i \in \mathcal{A}} POW(A_i)$. $UPOW.k(\mathcal{A}) = \bigcup_{A_i \in \mathcal{A}} POW.k(A_i)$.
- $UPOW(\mathcal{A}, \mathcal{B}) = \{X : (\exists A_i \in \mathcal{A}, X \subseteq A_i) \wedge (\forall B_j \in \mathcal{B}, X \cap B_j \neq \emptyset)\}$.
- $UPOW.k(\mathcal{A}, \mathcal{B}) = \{X : (\exists A_i \in \mathcal{A}, X \subseteq A_i) \wedge (|X| = k) \wedge (\forall B_j \in \mathcal{B}, X \cap B_j \neq \emptyset)\}$.

Apply. This command allows the user to apply a set operation to each set belonging to a family much the way the Lisp's mapcar function allows an operation to be performed to every member of a list.

Magnify. The input is a set S , an integer q , and an optional integer m . The command returns the set $S' = \{s * q : s \in S\}$. If m is specified then all of the multiplication is done modulo m . For example, if $S = (2, 4, 6, 8)$ then the command `T = magnify(S, 2)` assigns T to be the set $(4, 8, 12, 16)$.

Move. This command is similar to the *magnify* command which is described above; however, instead of computing the product of s and q , the sum is computed.

Incident. The input is a family \mathcal{F} and a set S . The function returns a new family \mathcal{F}' comprised of those sets in \mathcal{F} that have a non-empty intersection with S . The input family \mathcal{F} may be represented either symbolically or explicitly. The output family \mathcal{F}' will be represented the same way as the input family.

Adjacent. The input is a family \mathcal{F} and a set S . The function returns a new family \mathcal{F}' constructed by computing the *incident* function with arguments \mathcal{F} and S and then subtracting S from each set in the resulting family.

Extract. The input is the set $A = (a_0, a_1, \dots, a_{n-1})$ and $I = (i_0, i_1, \dots, i_{k-1})$. The function returns the set B comprised of the elements of A whose indices (the members of all *SetPlayer* sets are arranged in increasing order) belong to I . If none of the integers in I is an index of an element in A , then the empty set is returned.

Index. The input is the set $A = (a_0, a_1, \dots, a_{n-1})$ and $B = (b_0, b_1, \dots, b_{k-1})$. The function returns the set I comprised of the indices (the members of all *SetPlayer* sets are arranged in increasing order) of those members of A that belong to B . If none of the members of A belong to B , then the empty set is returned.

Contract. The input to this operation is an explicit family \mathcal{F} and a set S . The output is a new family \mathcal{F}' which is constructed so that for each set $T \in \mathcal{F}$, \mathcal{F}' contains the set $(T - S) \cup i$ where i is the minimal element of $T \cap S$, if it exists; otherwise T is unchanged. This command allows several elements of a family to be identified with each other.

Dual. The input to this operation is an explicit family F . The function returns a new family F_d with the property that there is one set S in F_d corresponding to each element e belonging to any set in F . The members of set S are the indices of the sets that e belongs to in F (the sets are indexed according to their lexicographical ordering). For example if $F = \{(1, 2), (2, 3), (3, 4)\}$, then $F_d = \{(0), (0, 1), ((1, 2), (2))\}$.

PREDICATES AND FUNCTIONS.

Equal. Determines whether two entities have the same value.

Member. Determines whether a user-specified integer (respectively, set) is a member of a user-specified set (respectively, family).

Subset. Determines whether a user-specified set (respectively, family) is a subset (respectively, subfamily) of another user-specified set (respectively, family).

Card. Computes the cardinality of a user-specified set or a family.

Trs. Defined earlier.

3.3. SAMPLE SESSION

The sample session included in this section demonstrates *SetPlayer* operating in command mode. For clarity, system output appears in a **typewriter style font**, and the users' commands appear in **bold face**. The files have been edited slightly to conserve space and were created on a Sun SPARCstation-1. The session demonstrates *SetPlayer* being used to find all minimum size vertex covers of graphs constructed by Robertson and Foster respectively (see Wong (1982)). These graphs have the minimum number of vertices among all 5-regular graphs that have no cycle with length less than 5 (the so called (5,5)-cages).

The sample session proceeds as follows. In commands (1) and (2), the edges of the graphs are read from the text files named "robertson" and "foster" and stored in the explicitly represented *SetPlayer* families named *rob* and *fos* respectively. Next, command (3) computes the union over all sets belonging to the family *rob*. The resulting set, which is the vertex set of both graphs, is stored in the *SetPlayer* set which is named *vert*.

Command (4) computes the collection of all subsets of *vert* that have a non-empty intersection with every member of the family *rob*[†]. The resulting family, which is the collection of all vertex covers (of all sizes) of the Robertson graph, is stored in the symbolically represented family named *covrob*. Command (5) has a similar effect.

In command (6), the number of sets belonging to the family *covrob* is computed and displayed. Hence we see that the total number of vertex covers (regardless of size) of the Robertson graph is 68,836. We now seek to determine the cardinality of the minimum size vertex covers of this graph. Using commands (7), (8), and (9), we see that every vertex cover of the Robertson graph contains at least 20 vertices. Furthermore, there are precisely five vertex covers with 20 vertices. Using command (10), we print each of these vertex covers. We then determine that the Foster graph has no vertex covers of size 19. Finally, we print the vertex covers with size 20 of the Foster graph using command (13). Observe that the Foster graph has one more minimum size vertex cover than the Robertson graph, but its total number of vertex covers is substantially smaller than that of the Robertson graph.

SetPlayer also supports a graphical interface which runs under the X- window system. When this interface is active, four windows are displayed. The upper left window is the *textual* window which is identical to the textual interface as illustrated in the sample session. When the mouse pointer is in this window, the user can interact with the system using either menu-mode or command-mode. The lower left window is the *database* window. This window shows the names of the currently existing sets and families and allows the user to display their contents. The *display* window is the upper right window and is used to show a graphical representation of a family. Explicit families are displayed by graphically connecting each set to the elements it contains. A symbolic family is displayed by drawing the difference tree which represents it. The user can click on an internal vertex of the tree to see the cardinality of the family represented by the subtree rooted there, or on a leaf to see its label and the cardinality of the family it represents. The lower right window is the *help/history* Window which displays information produced by the Help and History commands.

3.4. IMPLEMENTATION DETAILS

SetPlayer is written in the C programming language and runs under the UNIX[†] operating system. The C source files are currently over 70,000 lines long. The system makes use of a multiple precision arithmetic package available on UNIX[‡] as well as some UNIX system service routines for accessing the system clock. In addition *Lex* and *YACC* are used to build the command mode interface. The system is currently run on the Sun-3, Sun-4, and Sun SPARCstation-1 machines and we expect it can be easily ported to other systems which run the UNIX operating system.

The data structures that *SetPlayer* uses to store sets and families were outlined in section 2.1. The routines which perform the core operations, and solve the core set-theoretic tasks on *SetPlayer*'s data structures have been designed so that they can easily be called from higher level routines. These routines are grouped in files based on their

[†] Since *SetPlayer*'s command-driven interface allows commands to be nested, commands (3) and (4) can be replaced by `covrob = upow({funion(rob)}, rob)`.

[†] UNIX is a registered trademark of AT&T.

[‡] A version of the system that uses integers is also available.

```
Command (1): read(rob, "robertson")
Command (2): read(fos, "foster")
Command (3): vert = funion(rob)
Command (4): covrob = upow({vert}, rob)
Command (5): covfos = upow({vert}, fos)
Command (6): card(covrob)
1.49 CPU seconds were used.
68836
Command (7): card.23(covrob)
3.12 CPU seconds were used.
14355
Command (8): card.20(covrob)
2.98 CPU seconds were used.
5
Command (9): card.19(covrob)
1.39 CPU seconds were used.
0
Command (10): print(upow.20({vert}, rob))
Set #1, Card = 20, (0..4, 6, 8, 10..13, 15, 16, 18, 19, 21, 23, 24, 26, 28, 29)
Set #2, Card = 20, (1, 3, 4, 6, 7, 9, 11, 12, 14, 16, 17, 19..22, 24, 26, 28..30)
Set #3, Card = 20, (1, 3, 5, 6, 8, 10, 11, 13..16, 18, 20, 22..25, 27, 28, 30)
Set #4, Card = 20, (2, 4..7, 9, 10, 12, 13, 15, 17, 18, 20, 22, 23, 25..28, 30)
Set #5, Card = 20, (2, 4, 5, 7..10, 12, 14, 16..19, 21, 22, 24, 25, 27, 29, 30)
Command (11): card(covfos)
1.57 CPU seconds were used.
45375
Command (12): card.19(covfos)
2.10 CPU seconds were used.
0
Command (13): print(upow.20({vert}, fos))
Set #1, Card = 20, (1..4, 6, 8, 9, 11, 13, 14, 16, 17, 19, 21, 22, 24, 26, 28..30)
Set #2, Card = 20, (1, 2, 4, 5, 7, 9, 10, 12, 14, 16..22, 24, 26, 27, 29)
Set #3, Card = 20, (1, 2, 4, 6, 7, 9, 11, 12, 14, 15, 17, 19, 20, 22, 24, 26..30)
Set #4, Card = 20, (1, 2, 4, 6, 8..14, 16, 18, 19, 21, 23, 24, 26, 27, 29)
Set #5, Card = 20, (1, 3, 4, 6, 7, 9, 11, 12, 14, 16, 18..24, 26, 28, 29)
Set #6, Card = 20, (1, 3, 4, 6, 8, 10..16, 18, 20, 21, 23, 25, 26, 28, 29)
```

Oacci

Figure 1. *SetPlayer* Sample Session

functional similarity (for example all operations which perform primitive computational tasks on sets are stored in one file). It is possible for a programmer to learn how these core routines work while ignoring the majority of the code. Therefore, a user (with some programming experience) can use these routines to add new operations to *SetPlayer*. These operations can then be easily added to the *SetPlayer* menu (and if the programmer is proficient in the use of *YACC*) can be added to the command-line language with slightly more work. This makes it possible for each user of *SetPlayer* to have a personalized version of the system which reflect his/her individual research needs.

4. CONCLUSION AND FUTURE WORK

SetPlayer was designed as a tool for experimental research in Combinatorics and Graph Theory. It is intended to give researchers the ability to check, in a few seconds, an example of moderate size which would be practically impossible to deal with manually. Our approach in designing the system was motivated by the understanding that this

type of tool must have (1) a large variety of commands; (2) a short response time; and (3) an interactive interface. We believe *SetPlayer* meets these objectives and has thus demonstrated that symbolic computation on sets is practical.

To make the system a more powerful tool, the data objects which *SetPlayer* supports will have to be expanded to include sets described with the *nested* use of the *UPOW* operation, as well as multi-sets and other objects. The algorithms for computing *trs* and *trs.k* are central to the system, therefore they should be studied in depth and improved if possible. In addition to improving the algorithms which the system already supports, *SetPlayer* can be enhanced by increasing the number of operations it performs on sets and families.

Similar to the interdependency between the work on the design of computer algebra algorithms and the development of computer algebra systems (see Boyle and Caviness (1990); Kaltofen (1987)), progress in the development of combinatorial algorithms will stimulate and, in turn, be stimulated by the development of systems like *SetPlayer*. Odlyzko (1985) points out that one of the major purposes of computational algebra systems is to help the researcher develop the *insight* necessary to solve new problems. Solving problems often raises new questions which then suggest new features that computational algebra systems need. Thus, to a large degree, the evolution of computational algebra systems has depended on the results obtained from using these systems. Similar evolution should be expected from computational combinatorics systems like *SetPlayer*.

5. Acknowledgment

The authors are grateful to Robert W. McCloskey for his contributions to the early stages of the work on *SetPlayer* and the three referees for many useful comments. D. Berque was supported in part by the NSA under grant MDA 904-90-H-4027; R. Cecchini was supported in part by the NSF under grant IRI-8900511; M. Goldberg was supported in part by the NSA under grant MDA 904-90-H-4027 and by the NSF under grant CDA-8805910; R. Rivenburgh was supported in part by the NSF under grant IRI-8900511.

References

- Aho, A.V., Hopcroft, J.E., and Ullman, J.D., *The Design and Analysis of Computer Algorithms*, Addison-Wesley Publishing Co., (1974).
- Berque, D., Cecchini, R., Goldberg, M., Rivenburgh, R., *The SetPlayer System: An Overview and a User Manual*, Technical Report 90-20, (1991).
- Boyle, A., Caviness, B.F. (editors), *Future Directions for Research in Symbolic Computation*, Report of a Workshop on Symbolic and Algebraic Computation, Society of Industrial and Applied Mathematics, August (1990).
- Butler, G., *Private Communication*, June (1989).
- Dinitz, J., Stinson, D., (editors), *Contemporary Design Theory: A Collection of Surveys*, John Wiley & Sons, New York (1992).
- Garey, M. R., Johnson, D. S., *COMPUTERS AND INTRACTABILITY A Guide to the Theory of NP-Completeness*, W.H. Freeman and Company, San Francisco, California, (1979).
- Goldberg, M., Berque, D., McCloskey, R., *SetPlayer: An Interactive Software System for Set Manipulation*, Tech. Report, 88-26, Computer Science Department, RPI, October (1988).
- Goldberg, M., Spencer T., Berque, D., *A Low-Exponential Algorithm for Counting Vertex Covers*, submitted for publication, (1992).
- Kaltofen, E., *Computer Algebra Algorithms*, Ann. Rev. Comp. Sci. (1987), 2, 91-118.
- Kim, K.H., Roush, F.W., *On a Problem of Turán*, In P. Erdős, editor, *Studies in Pure Mathematics, To the Memory of Paul Turán*, Birkhäuser Verlag, Basel, (1983).
- Mills, W.H., *Construction of Covering Designs*, Congressus Numerantium, 73, (1990), 29-35.
- Odlyzko, A.M., *Applications of Symbolic Mathematics to Mathematics*, Applications of Computer Algebra, R. Pavelle, ed., Klumer, (1985), 95-111.

- Pavelle, R., Wang, P.S., *MAXYMA from F to G.*, J. Symb. Comp. (1985).
- Schwartz, J. T., Dewar, R.B.K, Dubinsky, E. and Schonberg, E., *Programming with Sets: An Introduction to SETL*, Springer-Verlag New York, (1986).
- Turán, P., On the theory of graphs, *Colloq. Math.* **3**, (1954), 19 -30.
- Valiant, L.G., *The Complexity of Enumeration and Reliability Problems*, SIAM Journal of Computing, **8**, (1979), 410-421.
- Wolfram, S., *Mathematica: A System for Doing Mathematics by Computer*, Addison-Wesley Publishing Co., (1988).
- Wong, Pak-Ken, *Cages -A Survey*, Journal of Graph Theory, Vol. 6, (1982), pp. 1 - 22.