A Breadth-First Strategy for Mating Search

Matthew Bishop*

Department of Mathematical Sciences, Carnegie Mellon University,
Pittsburgh, PA 15213, U.S.A.
mbishop+@cs.cmu.edu

Abstract. Mating search is a very general method for automating proof search; it specifies that one must find a complete mating, without specifying the way in which this is to be achieved. It is the foundation of TPS, an automated theorem-proving system for simply-typed lambda-calculus, and has proven very effective in discovering proofs of higher-order theorems. However, previous implementations of mating search have all relied on essentially the same mating search method: enumerating the paths through a matrix of literals. This is a depth-first strategy which is both computationally expensive and vulnerable to blind alleys in the search space; in addition, the incremental computation of unifiers which is required is, in the higher-order case, very inefficient. We describe a new breadth-first mating search method, called component search, in which matings are constructed by taking unions from a fixed list of smaller matings, whose unifiers are stored and manipulated as directed graphs. Component search is capable of handling much larger search spaces than were possible with path-enumeration search, and has produced fully automatic proofs of a number of interesting theorems which were previously intractable.

1 Introduction

The mating search (connection) method [And81,Bib82] is the foundation of TPS, an automated theorem-proving system for Church's simply-typed λ-calculus [Chu40]. It has proven very effective as a component of a procedure for discovering proofs of higher-order theorems. The salient feature of this method is that it requires consideration of the vertical paths through a two-dimensional matrix of literals. This matrix, also called a *jform*, represents the negation-normal form of the Skolemized negation of the theorem to be proven. Specifically, a proof is represented by a set of pairs of literals (a *mating*) with a substitution that simultaneously makes each of these pairs complementary, and such that every vertical path through the matrix contains at least one of these pairs (i.e. is *spanned by the mating*; a path which is not spanned is said to be *open*). Such a mating is said to be *complete*.

This characterisation of a proof suggests an algorithm for proof search: given such a matrix, begin with the empty mating and an empty substitution. At each

* Supported by the National Science Foundation under grants CCR-9624683 and CCR-9732312, and by Intel Corporation through an equipment grant.

step, given a mating M, find the leftmost open path in the matrix, and attempt to span it by adding a new pair of literals (*connection*) to M and verifying that the result is unifiable. If there are no open paths, then M is complete and we have succeeded. If there is an open path but it cannot be spanned, then backtrack by removing the most recently added connection in M. If we attempt to backtrack when M is empty, then fail.

The above method was the basis of every search procedure in TPS. (Of course, the method as presented is both naïve and incomplete; a full discussion of possible optimizations, or of the modifications required to allow for quantifier duplication and set-variable instantiation, is beyond the scope of this paper. We refer the reader to [And89,ABI$^+$96,Iss90] for more details.) We shall refer to such a search as a *path-enumeration* search. However, mating search is a very general method for automating proof search; note that it specifies only that one must find a complete mating, without specifying the way in which this is to be achieved. It is therefore possible to define alternative mating search procedures which do not rely on path enumeration; one such procedure is presented in this paper. We first discuss the reasons why one might want to define an alternative search method, and then sketch the details of the new procedure.

Firstly, determining the leftmost open path in a matrix is computationally quite expensive; it would be more efficient to have a search algorithm which could add connections to a mating without searching for open paths, thus separating the construction of the mating from the check for completeness. Furthermore, path-enumeration search becomes increasingly inefficient for large search spaces, since it is essentially a depth-first search of the space of possible matings. The order in which these matings are enumerated depends critically on the order in which vertical paths are enumerated, and hence upon the arrangement of literals in the matrix. In the worst case, if the very first connection which is added is wrong, we must explore all matings which contain this connection before finally backtracking and beginning again essentially from scratch. To compound the problem, one of the best strategies for duplication of quantifiers in a path-enumeration search says (more or less) "don't backtrack until you've tried duplicating all the quantifiers on the current open path"; this makes it even more difficult to backtrack out of blind alleys in the search.

In the higher-order case, unification is also a major problem. Higher-order unification is both undecidable and branching, and in a higher-order mating search the vast majority of the search time is spent on unification rather than mating search. Since our unification algorithm can also introduce negations, unification only gives very weak constraints on the connections which can be added to a mating (essentially, we can unify any connection in which one literal has a higher-order flexible head). This observation, together with the absence of most general unifers, prevents us from introducing many of the control mechanisms and reduction operations which have recently been described in first-order contexts (see, for example, [BBE$^+$95,Brü94]).

The above remarks suggest several problems which our new procedure should address: the problem of devising a breadth-first search strategy (preferably one which is impervious to rearrangements of the matrix), the problem of how to store and reuse as much information as possible (particularly information about higher-order unification), and the problem of avoiding the more expensive parts of the search wherever possible.

In this paper, we describe a search strategy in which small matings containing only a few connections are computed and stored along with their unifiers, and then these units (known as *components*) are combined into larger matings until a complete mating is found. We refer to this strategy as *component* search. We will accept a somewhat inefficient search algorithm in exchange for a much more efficient unification method: instead of incrementally unifying after each connection is added, we unify whole components, store their unifiers as a directed acyclic graph, and then compute the unifiers of larger matings by combining these graphs in an appropriate way. Thus, the main application of component search is to the higher-order case, since in the first-order and propositional cases, this trade-off between unification and search makes little sense: first-order unification can produce a most general unifier, if one exists, in linear time.

Components are similar to the hyper-links of [LP92] (which work only for first-order resolution-based search), while the unifiers of a component can be stored and combined in a manner similar to the ordered binary decision diagrams of [Bry86]. Component search is (almost) breadth-first, avoids path enumeration wherever possible, and can handle many more quantifier duplications than previously possible. Furthermore, the new unification method allows for unification modulo a bounded number of applications of some arbitrary equality rules, a feature not previously available in TPS. In defining component search, we have attempted to separate decisions affecting the search space itself (such as quantifier duplications and set-variable substitutions) from decisions affecting the search for a mating (such as which connection to consider next).

2 Component Search

The following is a well-known result:

Lemma 1 (The Disjunction Heuristic). *If J is a jform containing a disjunction $A \vee B$, and M is a complete and minimal mating for J which uses some literal in A, then M must also use some literal in B.*

Proof. Suppose $c \in M$ is a connection which uses some literal in A. Since the mating is minimal, there must be some vertical path P through J on which c is the only connection (if there were no such path, then $M \setminus \{c\}$ would be a complete mating, contradicting the minimality of M). Let P' be a variant of this path which is identical to P outside of A, but which passes through B instead of A. Since M is complete, P' must also be spanned by some connection $c' \in M$, and clearly $c' \neq c$ since c cannot span P'. If c' does not use any literal from B, then c' spans P, contradicting the assertion that c is the only connection spanning P. Hence c' must use a literal from B. □

This lemma gives us a simple method for extending many matings, and it will form the basis of our search procedure. We now formalise this idea.

Definition 1. *Given a jform J, we define an equivalence relation $\sim_J$ on the literals of J by: $A \sim_J B$ iff A and B lie on all the same horizontal paths (i.e. every horizontal path through* either *A or B passes through* both *A and B).*

It is clear that $\sim_J$ is an equivalence relation, and we will refer to the equivalence classes of $\sim_J$ as the *fragments* of J. For example, if J were in clause normal form, the fragments would correspond exactly to the clauses.

In order to define a search algorithm which is based on fragments, we will require the concepts of *touching* and *blocking*, which we now define.

Definition 2. *Let J be a jform, F a fragment of J, M a mating, and L the set of literals occurring in M. Then*

- *F is* touched by *M iff $F \cap L \neq \emptyset$.*
- *F is* blocked by *M iff $F \subseteq L$.*

We now have the following trivial but essential lemma, which follows immediately from the disjunction heuristic and the definition of $\sim_J$:

Lemma 2. *If M is a minimal complete mating for some jform J, then every fragment touched by M is blocked by M.*

Lemma 2 is the major motivation for the new search procedure, which we shall now outline informally.

Suppose that we are given a jform J, and that there exists some complete mating for J. Suppose further that we have somehow enumerated the acceptable (i.e. unifiable) connections of J, and also that we have enumerated, for each fragment of J, all of the minimal matings which block that fragment.

If we are then given a mating M, we may attempt to extend it to a larger mating by computing the list of fragments $f_1, \ldots, f_n$ which it touches but does not block. By Lemma 2, these fragments must be blocked in any minimal complete mating which extends M. Since we have enumerated all of the minimal matings which block f_1 (suppose they are $C_1, \ldots, C_k$), we may simply generate the new matings $M \cup C_1, \ldots, M \cup C_k$, discarding any new matings which are not unifiable. Each of these new matings is clearly strictly larger than M, and hence if M is in fact a subset of a minimal complete mating, we are guaranteed that one of the new matings is a larger subset of the same minimal complete mating. We may then repeat this process with each of the new matings, until we either reach a complete mating or fail.

Unfortunately, it is possible that there are no fragments which M touches but does not block. This is a necessary (Lemma 2 again) but not sufficient condition for M to be minimal and complete, so at this point we should check for completeness. If the mating is complete, we can stop; if not, we can always find an open path, and we can certainly enumerate all of the connections $c_1, \ldots, c_k$ which block it, and so we can generate $M \cup \{c_1\}, \ldots, M \cup \{c_k\}$ and continue as before. (In fact, we can do a little better than this in some cases, but finding and blocking an open path will remain our last resort.)

If we can somehow get this process started, then we have all the makings of a sound and complete search procedure. To start, we may (for example) pick a fragment using the set of support strategy, and begin with all the minimal sets which block it. If this fragment is touched by any minimal complete mating, we will eventually find a complete mating; if not, then we will eventually fail everywhere, in which case we may choose a new fragment and start again.

Definition 3. *A* component *is a mating M which blocks at least one fragment. A component M is said to be* minimal *for the fragment F iff M blocks F and no proper subset of M blocks F. A* minimal *component is one which is minimal for some fragment.*

Our search algorithm will use the following as a check for completeness of a mating M:

1. If there is a fragment which is touched but not blocked, then the next step of the algorithm should be to block one such fragment.
2. If all the touched fragments are also blocked, it may still be possible to use Lemma 1 to determine some list of fragments, at least one of which must be blocked in any minimal complete mating extending M. (For example, if the jform contains $A \vee (B \wedge C)$, where A, B and C are fragments, and A is blocked, then either B or C must also be blocked.) If we can find such a list, we return it, and at the next step the algorithm should attempt to block all of the given fragments.
3. Otherwise, find an open path and return that. If there is no open path, then M is complete; return `success`.

The first and third steps are essential; the former distinguishes component search from path-enumeration search, and the latter makes sure that component search is complete. We will not discuss the second step further in this paper, for reasons of space. In any case, the second step is non-essential, and its omission will simply mean that one case in the following algorithm never occurs.

The basic search algorithm for a jform J is as follows:

1. Fix all of the search bounds (unification depth if the problem is higher-order, number of quantifier duplications and so on). Amplify J, by applying set-variable substitutions, duplicating outermost quantifiers, and (optionally) hiding any equalities which can be handled by the unification algorithm (see the remarks following Definition 5), to produce a new jform J'.
2. Check each potential connection in J' for unifiability. If it is not unifiable, it will never be considered again; if it is unifiable, then store it as an *acceptable connection.*
3. Divide J' into fragments, and enumerate all of the minimal components for each fragment, discarding any which are not unifiable.

4. Choose one fragment f which has at least one unifiable minimal component. Let *current* be the list of all minimal components for f, and let *stored* be the list of all other components.
5. For each element c of *current*, apply the completeness checker to c:
   (a) If it returns `success`, we have a complete mating; return c.
   (b) If it returns a list of fragments which are touched but not blocked by c, choose one such fragment. Produce a list of new components by taking the union of c with each of the *stored* components which blocks this fragment; discard any elements of this list which are not unifiable.
   (c) If it returns a list of fragments at least one of which should be blocked, then combine c (by taking unions, as above) with each of the stored components which blocks any one of these fragments.
   (d) If it returns an open path, combine c with each of the acceptable connections on this path.
6. If some new components were produced in step 5, set *current* to be the list of components produced, and return to step 5.
7. Otherwise, no minimal complete mating can involve the fragment chosen in step 4. Delete all components touching this fragment, and return to step 4 to choose another starting fragment (unless no components remain, in which case either fail or return to step 1 and expand J' further).

A naïve implementation of this algorithm would be extremely inefficient, and the reader may well wonder why it works at all. The improvements to the unification method will be discussed in Section 3; in the following, we concentrate on the search algorithm itself. Firstly, we attempt to avoid making more than one pass through the outermost loop, by amplifying J many times at step 1; this creates many copies of each quantifier and so we must also introduce methods for controlling the resulting redundancy in the search space, which are discussed in Section 4.

In step 4, we are required to choose a starting fragment; since this fragment will occur in all of the components generated by step 5 (until we either reach step 7 or find a complete mating), we would like to choose a fragment which is very likely to be involved in a minimal complete mating in J'. If we can choose such a fragment, then the loop which begins at step 4 will be completed in one pass. The set of support strategy is one appropriate method for choosing such a fragment. If there are several good candidates, we will choose the one with fewest minimal components, since this will reduce the branching of the search. We also avoid (if possible) choosing any fragment which lies below a set-variable substitution, since this would initially restrict our search to only those matings involving that substitution.

Step 5 of the algorithm above is a strictly breadth-first search; given a mating, it will consider all extensions of that mating during the next step of the procedure. By assigning a weight to each component as it is created (with the intention that a larger weight represents a more promising component, in some sense), and modifying the above algorithm so that only components of maximal weight are extended in step 5, we may force the search procedure to explore the more promising parts of the search space first. This helps to prevent combinatorial explosion of the number of components in the early stages of the search, although the resulting search is of course no longer strictly breadth-first. There are many possible choices for such a weighting, but one which has proven effective in a wide variety of searches is $weight(C)$ = number of fragments blocked by C.

During step 5(b), we must select one fragment which is touched but not blocked. If there is more than one such fragment, then it seems reasonable to select the one which will cause the least branching in the search, which is the one having the fewest minimal components. In practice, we need to take account of the fact that some fragments may be duplicate copies of other fragments, and so we also prefer any original fragment to any duplicate copy, and any nth duplicate copy to any $(n+1)$st duplicate copy.

Finally, we observe that if we are forced to return to step 1 after failing everywhere, it is both possible and desirable to preserve and reuse information from the previous iteration of the search. The information which can usefully be preserved will depend on the exact way in which the search bounds are to be changed in step 1; in particular, if there are changes to the bounds on unification then much less information can be preserved than if the unification bounds remain fixed.

We now turn to the soundness and completeness of this algorithm. For the remainder of this chapter, we assume that unification contains no equality reasoning, and hence that no equalities are hidden by step 1. We also assume, for the sake of clarity in the proof of completeness, that if we reach step 7 and no components remain, we will choose to fail.

Theorem 1 (Soundness). *The above algorithm is sound.*

Proof. If the algorithm terminates without failing (which can only happen at step 5), then the result is a complete and unifiable mating. Hence the procedure is trivially sound, by previous results about the matings method. □

Theorem 2 (Restricted Completeness). *The above algorithm is complete, in the sense that it will find a complete mating in J' if there is one to be found within the given search bounds.*

Proof. (Sketch) At each stage, each component C in *current* is replaced by a set of new components, each of which contains a proper superset of the connections of C. If C was a subset of a minimal mating in J', then at least one of the new components will be a subset of the same minimal mating. Hence it is not possible to repeat step 5 forever; either we will eventually find a complete mating, or we will discard every component that is generated. In the former case, we are done; in the latter case, we will return to step 4. Since, by hypothesis, there is a complete mating M in J', step 4 must eventually choose some fragment f which is blocked by M. □

Note that since there are bounds on the depth of unification, the number of quantifier duplications, and so on, and since we have chosen to fail at step 7 rather than return to step 1, the algorithm is not complete in the more general sense. However, if we suppose that we always opt to return to step 1 (instead of failing), and that all the search bounds are relaxed each time we pass step 1, then we have the following:

Theorem 3 (Completeness). *If there is a complete mating in some expansion of J, then the above algorithm will terminate with a complete mating.*

Proof. (Sketch) Let J^* be the smallest expansion of J in which a complete mating can be found. By soundness, if J' contains no complete mating then the algorithm will eventually return to step 1 and replace J' by some J''. If we have relaxed the search bounds in an appropriate way, then eventually we will reach some jform $J^{(n)}$ which contains a copy of J^* and for which there is a complete mating within the search bounds. Then, by the restricted completeness theorem above, a mating will be found. □

3 Unification

In all of the path-enumeration searches in TPS, unification is performed after each new connection is added. This has the advantage of potentially making unification less expensive (in that the new connection may already be unified, or nearly unified, by the substitutions already generated). However, it also means that the same connection may be unified many times, in many contexts. It also requires unification to backtrack when a connection is discarded. Lastly, very little of the information generated by unification can usefully be stored, since it generally refers only to the immediate situation, which will not recur during the search.

Because of the undecidability of higher-order unification [Hue73,Gol81], we must have bounds on unification during the search. In TPS, these bounds were originally on the depth of the entire unification tree; however, it has been discovered that a uniform bound on the complexity of the substitution for each free variable is a much more efficient approach. It is generally not possible to increase these bounds during the search, since doing so would require reconsidering almost every connection and mating which had previously been discarded on grounds of non-unifiability.

Given this constraint, we can enumerate all of the possible ground substitutions for a given free variable before we even begin the search, using a variant of Huet's algorithm [Hue75,SG89]. This enumeration is possible since we know the maximum depth of any substitution, the types of all the free variables, the constants which occur in the search space, and any new constants that might be required by unification (we will need at most one new constant at each base type, to complete the unification of any remaining flex-flex pairs).

The ground substitutions are used to determine unifiability of a connection in the initial stages of the search. The set of unifiers for a connection or a mating is stored as a directed acyclic graph; when we combine components to produce a larger mating, we can also combine their unification graphs in a natural way to produce a graph containing the unifiers for the larger mating. We now outline the method by which this is achieved.

We assume that a total ordering of the free variables of J is fixed in step 1 of the algorithm. Since we have enumerated the possible substitution terms $t_1, t_2, \ldots, t_n$, we may refer to them by their numbers; we use the number 0 to mean "no substitution". For example, given the variable ordering (x, x', x''), we write $(41, 72, 0)$ for the substitution $\{x \mapsto t_{41}, x' \mapsto t_{72}\}$. There are of course typing restrictions to be satisfied, and the substitutions which occur should be understood to be appropriately typed.

Lemma 3. *Given a variable ordering $(v_1, \ldots, v_k)$, two substitutions $(n_1, \ldots, n_k)$ and $(m_1, \ldots, m_k)$ are compatible iff for all $1 \leq i \leq k$, either $n_i = m_i$ or $n_i m_i = 0$.*

Proof. Since all of the non-zero substitutions are ground, they are pairwise incompatible. Hence two substitutions can only be compatible if they are the same, or one of them is zero. □

Definition 4. *Given a variable ordering $V = (v_1, \ldots, v_n)$ and a list of substitutions S for (some of) the variables in the ordering, the* unification graph $U(V, S)$ *is an arc-labelled directed acyclic graph defined inductively as follows:*

- *If $n = 0$ (V is empty), then $U(V, S)$ is a single node $\perp$ with no descendants.*
- *Otherwise, define the equivalence relation $\sim_S$ on S by $\sigma \sim_S \tau$ iff $\sigma(v_1) = \tau(v_1)$. Let $S_1, \ldots, S_j$ be the equivalence classes of S under $\sim_S$, and for $1 \leq i \leq j$, let m_i be the value of $\sigma(v_1)$ for every σ in the equivalence class S_i. Let $V' = (v_2, \ldots, v_n)$. Now $U(V, S)$ is a node labelled v_1, with j descendant arcs, labelled $m_1, \ldots, m_j$, leading to the nodes $U(V', S_1), \ldots, U(V', S_j)$ respectively.*

We take a fixed variable ordering V of the variables of the jform, and construct a unification graph for each connection C, taking S to be the set of unifying substitutions for C. Each path through a unification graph from top to bottom represents a substitution for V. Our implementation ensures that two unification graphs are equal iff they are the same graph (i.e. `eq` in Lisp), which both saves space and speeds up the search.

For example, suppose $V = (x^1, x^2, x^3)$ and C_1 and C_2 are two connections for which the unifiers are given by $S_{C_1} = \{(1,3,7), (1,4,6), (1,6,8), (2,3,6), (2,4,8)\}$ and $S_{C_2} = \{(1,0,7), (2,3,7), (2,4,0), (3,5,0)\}$ respectively. Then $U(V, S_{C_1})$ and $U(V, S_{C_2})$ are as shown in Figure 1.

We use Lemma 3 to define the *merge* of two unification graphs, which is the intersection of the sets of substitutions they represent. In particular, if C_1 and C_2 are two components having unification graphs U_1 and U_2, then the component $C_1 \cup C_2$ has unification graph *merge*(U_1, U_2). For example, taking C_1 and C_2 as above, the unification graph for the mating $\{C_1, C_2\}$ is the merge of the unification graphs for C_1 and C_2, as in Figure 1.

Unification graphs are very similar to levelized ordered binary decision diagrams, and the algorithm for *merge* is essentially the same as the *apply* algorithm (for the boolean function $\wedge$) given in [Bry86]. The details of the algorithm are omitted for lack of space.

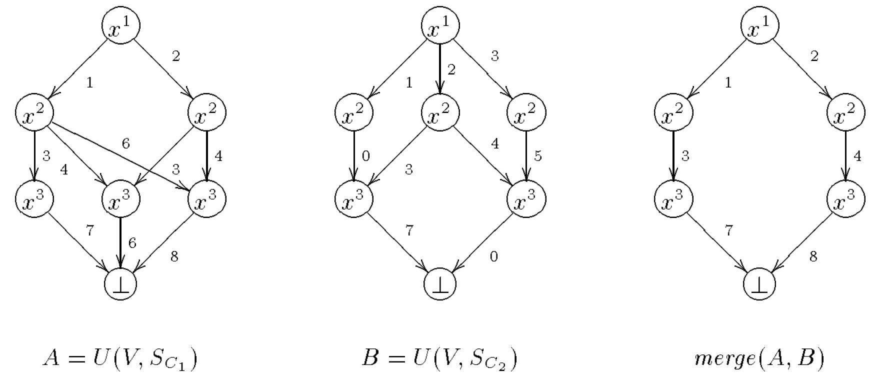

Fig. 1. Some examples of unification graphs.

Since unification is done before the mating search begins, we also have a convenient way to unify modulo (some bounded number of applications of) an arbitrary set of equalities. We now outline a method for unification with equalities; the implementation details, and proofs of their correctness, are beyond the scope of this paper.

Definition 5. *Let J be a jform, and $A = B$ an equality*[1] *which occurs positively in J as $\forall q.[\sim qA] \vee qB$. If every vertical path through J passes through either the literal $\sim qA$ or the literal qB, we say that $A = B$ is* global. *If $A = B$ is not global, but every vertical path through a literal L passes through either the literal $\sim qA$ or the literal qB, we say that the equality is* local to L.

When we are performing a search with unification modulo equalities, we will remove all of the global equalities from the jform and restrict the search to whatever literals remain. This can dramatically reduce the size of the search space, the number of quantifier duplications required for a proof, and the minimum unification depth required. We can do this because of the way in which the unification routine is applied: given a pair of literals L_1 and L_2 representing wffs s and t, apply all of the potential unification substitutions σ to the disagreement pair $\langle s, t\rangle$ in the usual way. Now, instead of rejecting any σ for which $\sigma s \neq \sigma t$, take the two ground terms σs and σt and attempt to find a common rewrite for them using (ground instances of) the global equalities and the equalities which are local to both L_1 and L_2. If a common rewrite can be found within the search bounds that have been given, we will accept (L_1, L_2) as a valid connection. In

[1] Recall that TPS expands equalities using Leibniz' definition: $A = B$ iff $\forall q.qA \supset qB$.

fact, we will implicitly have a chain of connections which begins at L_1 and passes through some number of equalities before eventually reaching L_2; using the definitions of global and local equalities, we can show that this chain of connections will span every path which is spanned by the pair L_1, L_2. When the mating is complete and we attempt to translate it to a natural deduction style proof using the methods of [Pfe87], this chain of connections must be made explicit in order for the translation to work.

4 Duplication and Set-Variable Instantiation

Previous search procedures in TPS have had two possible approaches to duplication: older procedures waited until all matings had been tried and then duplicated all of the outermost quantifiers, while the newer procedures used path-focused duplication [Iss91] in which the quantifiers on a particular vertical path would be duplicated if there was no other connection available on that path. Path-focused duplication has proven more effective than the older method, but neither approach can handle more than a few quantifier duplications because of the depth-first nature of path-enumeration.

In component search, we perform many duplications (specifically, all the duplications that fall within the search bounds set by the user) before we begin to search for a mating. So, for example, we might duplicate all outermost quantifiers k times before beginning the search. It may appear that, as a strategy for duplication, this is no better than the worst of the former stategies, but in the context of a component search we can control the way in which the new duplications are allowed to appear in a mating, and hence recover the benefits of duplication by need without also recovering its disadvantages.

We distinguish the original copy of a fragment from the first duplication, and that from the second duplication, and so on, and require that a component must touch the first k copies before it is allowed to touch the $(k+1)$st copy[2]. In this way, we recover the idea of "duplication by need".

We also introduce the concept of equivalence between matings; essentially, two matings M_1 and M_2 are equivalent if there is a permutation of the instances of each quantifier in the jform such that under the induced permutation σ on literals, we have $\sigma(M_1) = M_2$. We then modify the search algorithm to discard new components which are recognisably equivalent to known components (the condition in the previous paragraph is a specific case of this more general restriction). The proof that mating equivalence is well-defined, and that the resulting restrictions on the search algorithm do not affect completeness, are omitted for reasons of space.

We handle set-variable instantiations [BA98,ABI$^+$96] in much the same way as duplications. All of the set-variable instantiations that are to be tried are

[2] In fact, a more complicated condition — which we do not have the space to justify in this paper — is necessary: a duplicate fragment created by duplicating some variable x may be touched only if, for every earlier duplicate copy of x, some fragment below that earlier duplicated quantifier has been touched.

applied simultaneously at the beginning of the search, and to avoid a combinatorial explosion, the number of such instantiations that can be touched by any particular component is restricted, usually to one or two.

5 Results

Component search was tested on 56 first-order problems which had already been solved by path-enumeration search in TPS. Most of these searches took less than a second for both procedures; path-enumeration was generally slightly faster. This was expected, since component search has to spend time enumerating connections and minimal components before it even begins searching.

Results were also compared for 17 previously-solved higher-order problems; component search performed a little better on these problems than did path-enumeration search. However, component search spent a significantly smaller proportion of the total search time on unification than did path-enumeration search (46% of the total time, as opposed to 82% for path-enumeration).

To summarise the results above: on the theorems which could already be proven by TPS, component search is comparable to path-enumeration search. However, the motivation behind component search was to tackle larger search spaces which are intractable for path-enumeration search. The following theorems[3], all proven completely automatically by component search, are intractable for the path-enumeration searches in TPS. All the timings quoted below represent the time taken to find a complete mating on a Dell System Workstation 400MT, with dual 300 MHz Intel Pentium II processors, running TPS under Allegro Common Lisp 4.3 for Linux.

THM126: $\forall h^1_{\beta\gamma} \forall h^2_{\alpha\beta} \forall A_{o\gamma} \forall f^1_{\gamma\gamma\gamma} \forall B_{o\beta} \forall f^2_{\beta\beta\beta} \forall C_{o\alpha} \forall f^3_{\alpha\alpha\alpha}.\mathrm{HOM2}\ h^1 A f^1 B f^2$ $\wedge\ \mathrm{HOM2}\ h^2 B f^2 C f^3 \supset \mathrm{HOM2}\ [h^2 \circ h^1] A f^1 C f^3$ *(18.27 seconds)*
HOM2 $hAfBg$ is an abbreviation for "the set A is closed under the binary operation f, likewise B is closed under g, h maps A into B, and for all x and y in A we have $h(f(x,y)) = g(h(x), h(y))$" This theorem states that the composition of two homomorphisms is a homomorphism; an interactive proof of a version of it (formulated in the first-order set theory of von Neumann, Gödel and Bernays) was the central theorem of [BLM+86]. A simpler version, THM133, in which the sets A, B and C are replaced by types α, β and γ (making the closure assumptions unnecessary) is easily provable by TPS, but the search space of THM126 is much larger than that of THM133.

THM145: $\forall R_{o\alpha\alpha} \forall U_{\alpha(o\alpha)}.\mathrm{TRANSITIVE}\ R \wedge \forall s_{o\alpha}[\forall z_\alpha[sz \supset Rz.Us] \wedge$ $\forall j_\alpha.\forall k_\alpha[sk \supset Rkj] \supset R[Us]j] \supset \forall f_{\alpha\alpha}.\forall x_\alpha \forall y_\alpha[Rxy \supset R[fx].fy] \supset$ $\exists w_\alpha.Rw[fw] \wedge R[fw]w$ *(59.2 minutes)*
This is the Knaster-Tarski fixed point theorem for lattices [Kna27], in a formulation due to Coquand. It states that in a complete lattice, every monotone

[3] Some remarks about the notation used in TPS: o is the type of truth values, ι is the type of individuals, and $(\alpha\beta)$ is the function type often written $\beta \to \alpha$. A dot corresponds to a left bracket whose mate is as far to the right as is consistent with the brackets already present.

function f has a fixed point. U is is the least upper bound of the set s, and R is the ordering on the lattice. TPS finds the fixed point $U[\lambda x.Rx[fx]]$.

THM146: $\forall r_{o\iota\iota}\forall x_\iota\forall y_\iota.\forall q_{o\iota\iota}[r \subseteq q \wedge \forall u_\iota\forall v_\iota\forall w_\iota[quv \wedge rvw \supset quw] \supset qxy] \equiv \forall p.r \subseteq p \wedge \text{TRANSITIVE } p \supset pxy$ *(22.5 minutes)*
This theorem states the equivalence of two definitions of the transitive closure of a relation r; it was suggested by Dana Scott. It requires a set-variable instantiation for p of $\lambda u\lambda v.\forall w[ruw \supset qxw] \supset qxv$.

THM270: $\forall f_{\beta\alpha}\forall g_{\xi\alpha}\forall h_{\xi\beta}.\forall x_\alpha[h[fx] = gx] \wedge \forall y_\beta\exists x[fx = y] \wedge$ GEN-HOM $f \wedge$ GEN-HOM $g \supset$ GEN-HOM h *(0.53 seconds)*
In this theorem, GEN-HOM is an abbreviation for $\lambda f_{\sigma\tau}\forall x_\tau\forall y_\tau.f[*_{\tau\tau\tau}xy] = *_{\sigma\sigma\sigma}[fx].fy$. Suppose that α, β and ξ are types representing three sets, equipped with binary functions $*_{\alpha\alpha\alpha}, *_{\beta\beta\beta}$ and $*_{\xi\xi\xi}$. If $f_{\beta\alpha}, g_{\xi\alpha}$ and $h_{\xi\beta}$ are functions such that f is surjective, $h \circ f = g$, and both f and g are homomorphisms with respect to the appropriate functions $*$, then h is also a homomorphism. This problem was suggested by Erica Melis.

CR-THEOREM: $\forall R_{o\tau\tau}$.REFLEXIVE $R \wedge$ TRANSITIVE $R \wedge$ DIAM $R \supset \forall M_\tau\forall N_\tau$.EQ* $RMN \supset$ COMMON-REWRITE RMN *(5.08 minutes)*
This is one formulation of the Church-Rosser theorem [CR36]: if R is a reflexive transitive rewrite relation satisfying the diamond property (DIAM: for any A, if A has two rewrites B and C, then B and C have some common rewrite D), and M and N are related by the equivalence closure of R (EQ* R is defined as $\lambda x_\tau\lambda y_\tau\forall P_{o\tau\tau}.R \subseteq P \wedge \text{SYMMETRIC } P \wedge \forall A_\tau\forall B_\tau[\exists w_\tau[PAw \wedge PwB] \supset PAB] \supset Pxy$), then M and N have a common rewrite. This formulation is a variant of Theorem 3.1.12 of [Bar84]. The variable P in the definition of EQ* requires a set-variable substitution of the form $\lambda x^1_\tau\lambda x^2_\tau$COMMON-REWRITE $p_{o\tau\tau}x^1x^2$, which TPS discovers using the set-variable instantiations of [BA98].

GRP-COMM2: $\forall x_\iota[P_{\iota\iota\iota}e_\iota x = x] \wedge \forall y_\iota[Pye = y] \wedge \forall z_\iota[Pzz = e] \wedge \forall x\forall y\forall z [P[Pxy]z = Px.Pyz] \supset \forall a_\iota\forall b_\iota.Pab = Pba$ *(20.97 seconds)*
If every element of a group is self-inverse, then the group is Abelian.

LATTICE THEORY: Using unification modulo equalities, component search has proven a number of examples in lattice theory (axiomatised in first-order logic with equality), taken from [Szá63]. For TPS, these are good examples of very large search spaces, since the definitions of "lattice", "Boolean algebra", "pentagon" etc. contain many axioms.

1. (Schröder; Thm. 28 of [Szá63]) Join distributes over meet iff meet distributes over join (8.2 min).
2. ([Szá63], p. 122) In a Boolean algebra (complemented distributive lattice), the complement is uniquely defined (4.2 min).
3. (Jordan; Thm. 30 of [Szá63]) The equivalence of two definitions of modularity (6.9 min).
4. (Dedekind; part of Thm. 32 of [Szá63]) A lattice is not modular if it contains a pentagon (6.2 min).
5. (Birkhoff; part of Thm. 33 of [Szá63]) A lattice is not distributive if it contains three points which (taken pairwise) have the same meet and join (5.3 min).

6 Conclusions and Further Work

We have described the component search method, a new mating search algorithm which appears in many cases to be a more powerful search method than path-enumeration search. We have also outlined, as a part of the component search procedure, a graph-based method for storing and manipulating sets of higher-order unifying substitutions. Component search is effective for much larger higher-order search spaces than were previously tractable, since it is less vulnerable to blind alleys in the search space and spends significantly less time on higher-order unification. A more detailed discussion of component search can be found in [Bis99].

Component search seems particularly appropriate for problems in which there are significant parts of the search space which are entirely redundant (for example, proofs from a fixed set of axioms in which some of the axioms are unused in the proof). We make use of this property of the search in applying, before the search begins, many quantifier duplications and set-variable instantiations.

For future work on component search, there are several promising lines of investigation. Improved methods for identifying and pruning redundant matings would allow still larger search spaces to be considered, as would a more intelligent method for choosing a fragment (in the places where the algorithm gives one a choice). There are presumably also better possible weighting functions for components, which would make the search more effective. Even the relation $\sim_J$ which defines the fragments can be changed — under the definition in this paper, there exist jforms in which every fragment contains exactly one literal; a different choice of $\sim_J$ would seem to be called for in these cases.

There are a number of other possible optimizations to the unification procedure, some of which have already been implemented. The definition of variable ordering has been generalised to allow one arc of the unification graph to represent many substitutions, but the influence of the choice of variable order on the search has not yet been studied in any detail. If the analogy with ordered binary decision diagrams is valid, then one would expect the choice of variable order to have a considerable effect on the efficiency of the search.

Lastly, the search algorithm looks like a good candidate for OR-parallelism, in which several processors cooperate on a proof using some kind of blackboard architecture to communicate results about unification graphs and partial matings.

References

[ABI+96] Peter B. Andrews, Matthew Bishop, Sunil Issar, Dan Nesmith, Frank Pfenning, and Hongwei Xi. TPS: A theorem proving system for classical type theory. *Journal of Automated Reasoning*, 16:321–353, 1996.

[And81] Peter B. Andrews. Theorem proving via general matings. *Journal of the ACM*, 28:193–214, 1981.

[And89] Peter B. Andrews. On connections and higher-order logic. *Journal of Automated Reasoning*, 5:257–291, 1989.

[BA98] Matthew Bishop and Peter B. Andrews. Selectively instantiating definitions. In Claude Kirchner and Hélène Kirchner, editors, *Proceedings of the 15th International Conference on Automated Deduction*, volume 1421 of *Lecture Notes in Artificial Intelligence*, pages 365–380, Lindau, Germany, 1998. Springer-Verlag.

[Bar84] H. P. Barendregt. *The λ-Calculus*. Studies in logic and the foundations of mathematics, North-Holland, 1984.

[BBE+95] W. Bibel, S. Bruning, U. Egly, D. Korn, and T. Rath. Issues in theorem proving based on the connection method. In Peter Baumgartner, Reiner Hähnle, and Joachim Posegga, editors, *Theorem Proving with Analytic Tableaux and Related Methods. 4th International Workshop. (TABLEAUX '95)*, volume 918 of *Lecture Notes in Artificial Intelligence*, pages 1–16, Schloß Rheinfels, St. Goar, Germany, May 1995. Springer-Verlag.

[Bib82] Wolfgang Bibel. *Automated Theorem Proving*. Vieweg, Braunschweig, 1982.

[Bis99] Matthew Bishop. *Mating Search Without Path Enumeration*. PhD thesis, Department of Mathematical Sciences, Carnegie Mellon University, 1999.

[BLM+86] Robert Boyer, Ewing Lusk, William McCune, Ross Overbeek, Mark Stickel, and Lawrence Wos. Set theory in first-order logic: Clauses for Gödel's axioms. *Journal of Automated Reasoning*, 2:287–327, 1986.

[Brü94] S. Brüning. *Techniques for Avoiding Redundancy in Theorem Proving Based on the Connection Method*. PhD thesis, TH Darmstadt, 1994.

[Bry86] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[CR36] Alonzo Church and J.B. Rosser. Some properties of conversion. *Transactions of the American Mathematical Society*, 3:472–482, 1936.

[Gol81] Warren D. Goldfarb. The undecidability of the second-order unification problem. *Theoretical Computer Science*, 13:225–230, 1981.

[Hue73] Gerard P. Huet. The undecidability of unification in third-order logic. *Information and Control*, 22:257–267, 1973.

[Hue75] Gerard P. Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[Iss90] Sunil Issar. Path-focused duplication: A search procedure for general matings. In *AAAI–90. Proceedings of the Eighth National Conference on Artificial Intelligence*, volume 1, pages 221–226. AAAI Press/The MIT Press, 1990.

[Iss91] Sunil Issar. *Operational Issues in Automated Theorem Proving Using Matings*. PhD thesis, Carnegie Mellon University, 1991. 147 pp.

[Kna27] B. Knaster. Une théorème sur les fonctions d'ensembles. *Annales Soc. Polonaise Math.*, 6:133–134, 1927.

[LP92] Shie-Jue Lee and David A. Plaisted. Eliminating duplication with the hyperlinking strategy. *Journal of Automated Reasoning*, 9:25–42, 1992.

[Pfe87] Frank Pfenning. *Proof Transformations in Higher-Order Logic*. PhD thesis, Carnegie Mellon University, 1987. 156 pp.

[SG89] Wayne Snyder and Jean Gallier. Higher-order unification revisited: Complete sets of transformations. *Journal of Symbolic Computation*, 8:101–140, 1989.

[Szá63] Gabor Szász. *Introduction to Lattice Theory*. Academic Press, New York and London, 1963.