# Evaluating First Order Formulas—the foundation for a general Search Engine*

**Fahiem Bacchus**
**Dept. Of Computer Science**
**University Of Waterloo**
**Waterloo, Ontario**
**Canada, N2L 3G1**
**fbacchus@logos.uwaterloo.ca**

**Michael Ady**
**Winter City Software**
**Edmonton, Alberta**
**Canada, T5R 2M2**
**winter.city@v-wave.com**

## Abstract

Search and declarative representations are two of the most important themes in AI research. Many problems in AI require search for their solution, and declarative representations of the knowledge required to solve these problems offer many advantages. In this paper we show how these two themes can be combined. In particular, we show how a general search engine can be designed that is based on the most familiar declarative representation: first order logic. The system can be used to solve both search problems and optimization problems, e.g., job-shop scheduling, as well as problems traditionally viewed as being planning problems, e.g., simple logistics planning. The system is based entirely on the idea of evaluating first-order formulas against finite (or computable) models. We show that formula evaluation can be used to perform arbitrary computations, to expand an implicitly defined search space, to compute heuristics, and even provide sophisticated control over the search engine. The end result is a very cleanly designed system that is well suited for experimental exploration of a wide range of search problems.

## 1 Introduction

Many problems in AI can be viewed as search problems. That is, we can solve the problem by searching some state space. Generally, this space is very large and is represented only implicitly. As we search, the implicit specification of the state space is used to generate an expanded, or explicit, representation of various regions of the space. The success of search often revolves around methods that can solve the problem while expanding as small a part of the state space as possible.

A search problem is specified by two components: a representation of the states, and a specification of the legal transitions between the states (the set of legal *actions*). Typically, only the initial state is given explicitly. The entire space consists of all states reachable from the initial state via some sequence of actions. During search the actions are utilized to generate explicitly represented regions of the space.

*This research was supported by the Canadian Government through their IRIS project and NSERC programs.

Many mechanisms can be used to represent the explicitly generated states. These representations can be problem specific data structures. For example, in the game of checkers we might use a two-dimensional array indexed by board position to specify a state of the game. Once a data structure is chosen the legal state transitions can then be specified by routines that make appropriate modifications to this data structure. For example, the legal moves in a checkers game can be specified by routines that modify the array that represents the current board configuration. In this manner the actions can be used to generate explicit representations of various states reachable from the initial state.

Problem specific data structures have the advantage that they can be more efficient, but the disadvantage that they lack flexibility. We have to design and implement a new data structure for every new search problem. There is, on the other hand, a very general state representation: states can be represented as relational structures, i.e., finite first-order models. The STRIPS databases used in classical planning are an instance of such structures. A key feature of such structures is that they support the efficient evaluation of first order formulas.[1]

In this paper we show how to take advantage of this fact, utilizing the mechanism of evaluating first order formulas to build a general search system. In this system search problems are specified using relation structures to represent the states and an extended form of ADL [Ped89] actions to represent the transitions. We will demonstrate how a formula evaluation mechanism can then be used to expand the implicitly defined search space, to compute heuristics, and to provide sophisticated control over the search engine. The system was originally designed to address classical planning problems, and its application in that area is described in [BK98]. That paper describes how temporal logic formulas can be used to control search in planning problems. In this paper, in contrast, we highlight the role of evaluating first order formulas, show how the resulting system can be applied to search problems not typically addressed in the planning community, and

[1] Although STRIPS databases are the foundation of much of the work in planning, these relational structures are not commonly used when *searching* for a plan. For example, in partial-order planning [Wel94] the state space explored consists of states that are partially ordered plans, not STRIPS databases. State spaces consisting of relational structures are used by forward chaining planners.

concentrate on showing how sophisticated control over the search engine can be achieved using mechanisms unrelated to the temporal logic formulas used in [BK98].

## 2 Evaluating First Order Formulas

In this section we briefly discuss the process of evaluating first order formulas on finite relational structures. The details of the algorithms are contained in [BK98].

**Relational Structures.** Relational structures are finite first order models. One starts with a first order language $\mathcal{L}$ that consists of a finite collection of predicate, function and constant symbols. Taking the set of constant symbols to be the domain of discourse, we can construct a relational structure (a finite model) with a collection of tables, one for every predicate and function symbol. In particular, for every arity $k$ predicate symbol $P$ its table contains a set of length $k$ tuples of constants—this set specifies the complete set of instances of $P$ satisfied by the model. For every arity $k$ function symbol $f$ its table provides a mapping from all length $k$ tuples of constants to the value of $f$ in the model when applied to these arguments.

**Evaluating Formulas.** Relational structures allow one to efficiently determine for every atomic formula, containing only constants as terms, whether or not it is satisfied in that structure. It is easy to extend this so that we can evaluate whether or not an arbitrary first order formula is satisfied by that structure.

Evaluation is performed recursively, while maintaining a list of the current bindings of the variables contained in the formula. At the bottom of the recursion lies the evaluation of terms. Constants evaluate to themselves, while variables evaluate to their current bindings. Terms involving the application of a function are evaluated by first evaluating each of the function's arguments, then performing a lookup of the function's value on those arguments in the current state. Each atomic predicate is evaluated by first evaluating the terms it contains and then looking up that tuple in the predicate's table in the current state. If that tuple is present the atomic formula evaluates to true, otherwise it evaluates to false.[2]

Other formulas are evaluated recursively by using their semantics to decompose them into evaluating appropriate sets of atomic formulas. To deal with quantification we utilize bounded quantification. That is, for every quantified variable we specify a range of constants over which it can take its values. To evaluate a quantified formula we successively bind the quantified variable to the constants in its range, then we recursively evaluate the body of the quantified formula.

There are two key features in the design of the evaluator that allow us to utilize it as a rich programming language, rather than solely as a formula evaluator. First the evaluator employs short-circuiting. For example, if we have a universal quantifier and the body evaluates to false on one of the bindings we stop and return false as the evaluation of the entire formula. Short-circuiting allows one to write and evaluate recursive formulas and also to implement a full range of program control flow. These features are illustrated in the examples that follow. The other feature is the provision of a range of system defined pseudo-predicates. These predicates can be evaluated as ordinary atomic formulas, but in addition to returning a truth value they also generate various side-effects when evaluated. By evaluating such predicates inside of formulas we can provide sophisticated control over the system by setting flags, invoking computations, performing I/O, etc., as side-effects.

**Numbers and Definitions.** Although a wide range of computations can be expressed as queries that can be answered using the state's relational structure, a general search engine also needs access to numeric computation.

We add to the evaluator the ability to compute a range of numeric functions, e.g., addition, multiplication, modulus, etc., and numeric predicates, e.g., greater than, less than, equality, etc. One can view each relational structure as being extended with tables for these numeric functions and predicates. Of course, the actual implementation need not store such tables, it can utilize the underlying hardware to compute the denotation of these symbols on the fly. In addition, we supply the evaluator with mechanisms to specify various numeric ranges over which quantified variables can range.

We also permit definitions; i.e., predicate and function symbols defined by first order formulas. Significant computational and expressive power is gained by allowing these definitions to be recursive.

**Examples.**

• Suppose the relational table for the predicate symbol $P$ is $\{(a,a),(b,b)\}$, and for the function symbol $f$ is $\{a \mapsto a, b \mapsto a\}$. Then `(P a (f b))` will evaluate to true, as `(f b)` has value $a$, and $(a,a)$ is in the denotation for the symbol $P$.

• We can define `(above ?x ?y)` to be the formula

```
(or (on ?x ?y)
    (exists (?y1) (on ?y1 ?y) (above ?x ?y1)))
```

`Above` is the transitive closure of `on`. All quantified formulas have three components, a list of quantified variables, a quantifier bound, and the body of the formula. The existential above allows `?y1` to range over all constants that have the property `(on ?y1 ?y)`, where `?y` has already been bound.

If we try evaluating a formula like `(above c d)` the evaluator first checks if `(on c d)` is true. Say that it is not and that instead we have `(on h d)`. Then in evaluating the second disjunct the evaluator will find the binding `h` for `?y1` and will recursively evaluate `(above c h)`. It will terminate when the first conjunct is true or when there is no possible binding for `?y1`. The short-circuiting of the evaluation process is essential in ensuring that such recursive definitions have their intended semantics.

• `(prime ?x)` can be defined to be the formula

```
(forall (?i) (isbetween ?i 2 (floor (sqrt ?x)))
    (not (= 0 (mod ?x ?i))))
```

This is true of an integer iff that integer is a prime. It utilizes the numeric quantifier range `(isbetween ?i ?n ?m)`. This provides a range for the variable `?i` that runs from `?n`

[2] Thus we assume that the tables are complete; i.e., they contain all positive instances of each predicate. This is of course required if these tables are to be regarded as specifying a first order model. In some domains the tables will always be complete, but in some domains we may have to assume that the tables are complete even if they are not necessarily so. This is the closed world assumption.

to ?m inclusive. In this case, from 2 to the floor of the square root of ?x. The formula says that no integer in this range divides ?x (has a modulus of zero with ?x). Thus (prime 33) evaluates to false while (prime 3) evaluates to true.

```
(forall (?i) (posint ?i)
    (implies (prime ?i) (print ?i)))
```

will print all primes. The numeric quantifier range posint ranges over all positive integers in ascending order, thus only short-circuiting will terminate the computation. In this case, the computation is infinite. If the current binding of ?i is a prime (the antecedent of the implication is true) (print ?i) will be evaluated (the consequent of the implication). This predicate always evaluates to true, and has the side effect of printing out its evaluated arguments.

• Functions like (gcd ?a ?b) can be defined

```
(or (and (= ?b 0) (:= gcd ?a))
    (:= gcd (gcd ?b (mod ?a ?b))))
```

This is Euclid's algorithm. The assignment predicate, :=, always evaluates to true and has the side effect of assigning a value to its first argument, a "variable". We use the convention that an assignment to a function name in its definition determines the value returned by the function. The function gcd can be used inside of formulas just like any other function. To make specifying functions easier we also allow local variables inside of function definitions. These local variables can be assigned to, and such assignments can make evaluating the function more efficient.

## 3 Specifying State Transitions

To specify a search space we must also provide a means of specifying transitions. Since the states in the search space are being represented as relational structures we need a general mechanism for specifying updates to a relational structure. Fortunately, this work has already been done by Pednault in his ADL language [Ped89]. ADL was specifically designed for updating relational structures. What is interesting is that with side-effects such updates can be easily implemented using the evaluator.

In particular, we add two new pseudo-predicates add and del. def-adl-operator is used to declare a transition. It takes two arguments, a precondition formula and a body that consists of a sequence of formulas that may contain add and del predicates. For example,

```
(def-adl-operator (act ?x ?y)
  (pre (?x ?y) (Q ?x ?y)
     (and (R ?x) (P ?y)))
  (forall (?z) (Q ?x ?z)
     (implies (R ?z) (add (P ?z)))))
```

defines the parameterized action act. The operator pre can be read exactly like the universal quantifier forall. For every distinct binding of ?x and ?y satisfying the precondition formula (i.e., ?x and ?y ranging over positive instances of Q and satisfying (and (R ?x) (P ?y))) we have a particular instance of the act action. Each instance of act generates a successor state by first making a copy of the current state and then evaluating every formula in the body.[3] In this case there is only one formula in the body to be evaluated. This formula is evaluated in the standard manner except that any time an add or del predicate is evaluated that predicate always evaluates to true and has the side effect of adding or deleting the ground atomic formula that is its argument (in the newly created copy of the current state). In this case ?z will range over all constants that satisfy (Q ?x ?z) (?x has already been bound in the precondition). For each of these constants if it also has the property R, the action will make it also satisfy P by adding that atomic fact to the new state. add can also be used to update function values by adding equality predicates. For example, (add (= (f a) 10)) will update the function f so that its value on a is 10.

[3] These formulas can include recursively defined predicates, thus defining recursive ADL actions. This goes beyond Pednault's original proposal.

## 4 Search

So far we have a general representation for states (the relational structure), a flexible mechanism for performing computations and querying those states (the evaluator), and a powerful mechanism for specifying transitions (that utilizes the evaluator). To complete the search engine we need a mechanism for specifying the particular search problem, and for performing the search.

For simplicity we restrict ourselves to search problems that involve finding a path from an initial state to some satisfactory goal state. The initial state can be specified as a relational structure, and the system provides various mechanisms to facilitate this. The goal can be specified as a list of ground atomic facts and function values. Any state that satisfies all of these facts and has all of these function values is a satisfying goal state.

Since the goal is specified in this simple way we can augment the evaluator so that it can query the goal. In particular, the evaluator allows one to specify quantifier ranges and atomic formulas inside of a "goal" modality. For example, the formula (forall (?x) (goal (P ?x)) (and (Q ?x) (goal (R ?x)))), is true in the current state if all objects specified to require property P in the goal currently have property Q and are specified to require property R in the goal. The examples below will show the utility of being able to query the goal.

The search mechanism is particularly simple. The system provides two basic search procedures, depth-first and breadth-first with or without heuristic control. When heuristics are used the depth-first search orders the children of the current state by a computed heuristic value. Thus it explores the most promising child first. Similarly, heuristic breadth-first search will continually explore the node on the search frontier with lowest heuristic value.

In addition there are a number of built in controls implemented as pseudo-predicates. For example, one can specify a depth-bound: the search engine will not expand any states that are further away from the initial state than the depth bound. One can also specify other bounds on the search, and gather various statistics during search.

Surprisingly, with the aid of the evaluator a range of more sophisticated search algorithms can be implemented using these basic search procedures and the built in controls.

## 5 Examples

**Lloyd's Puzzle using Heuristic Control.** Lloyd's sliding tile puzzle is a standard problem that can be solved by search. To represent this puzzle using a first-order language we can give a name to each of the positions (`p1`, ..., `p9`, with `p1` being the top left corner and `p9` being the right bottom corner), and tiles (`t1`, ..., `t8`), including the "empty-space" or blank tile (`B`). Then we can use two binary predicates `at` and `nextto` to specify the configuration of a particular state. `(at ?t ?p)` means that tile `?t` is at position `?p`, and `(nextto ?p1 ?p2)` means that position `?p1` is next to position `?p2` in the sense that the blank can be moved from position `?p1` to position `?p2` in a single action.[4] Finally, for convenience we also include two "type" predicates `tile` and `pos`: `tile` is true of all the tiles and `pos` is true of all positions. Thus an initial state

| | | |
|---|---|---|
| t4 | t1 | t6 |
| t2 | | t8 |
| t7 | t5 | t3 |

is represented by the set of positive `at` instances { `(t4,p1)`, `(t1,p2)`, `(t6,p3)`, `(t2,p4)`, `(B,p5)`, `(t8,p6)`, `(t7,p7)`, `(t5,p3)` }, along with the tables for the other (state-invariant) predicates.

A goal state can be similarly represented by specifying the denotation of the `at` predicate.

The set of transitions (moves into the blank position) can be specified with the single action

```
(def-adl-operator (slide ?tile ?from ?to)
  (pre (?to)    (at B ?to)
       (?from)  (nextto ?to ?from)
       (?tile)  (at ?tile ?from))
  (add (at B ?from)  (at ?tile ?to))
  (del (at B ?to)    (at ?tile ?from)))
```

This operator simply locates the blank tile and for every neighboring position it creates a new state where the tile in that position has exchanged places with the blank tile. If, e.g., the blank is in position `p5` then there will be four different successor states (four distinct bindings of the variables appearing in the precondition).[5]

For heuristic control we can define a function that when evaluated on a state computes the Manhattan distance heuristic. Since the positions are represented by symbolic constants it is simplest to define an `mh-dist ?p1 ?p2` function that returns the Manhattan distance between any two positions. The values of this function can be set in the initial state. For example, we set `(mh-dist p1 p9)` to be 4. Now the following function can be defined that computes a value in every state that is equal to the sum of the Manhattan distance of the tiles from their final (goal) location.

```
(def-defined-function (total-mh-distance)
  (local-vars ?v1)
  (and
    (:= ?v1 0)
    (forall (?t ?p) (at ?t ?p)
      (implies (not (= ?t B))
        (forall (?pg1) (goal (at ?t ?pg1))
          (:= ?v1 (+ ?v1 (mh-dist ?p ?pg1))))))
    (:= total-mh-distance ?v1)))
```

This function uses a local (lambda-like) variable specified by the `(local-vars ?v1)` declaration. `?v1` is a variable that scopes the entire formula defining the function. To evaluate the function on a particular state we first initialize `?v1` to be 0, then iterating over all tiles and positions, if the tile is not the blank tile, we find the intended goal position for that tile. This is accomplished by using the goal modality to query the goal. Then the local variable `?v1` is incremented by the Manhattan distance between the tile's current location (the current binding of `?p`) and its intended goal position (the current binding of `?pg1`). Once the universal has been evaluated `?v1` contains the sum of the Manhattan distances, and that value can be returned as the function value.

We can then solve problems using this heuristic by specifying `(set-heuristic-fn (total-mh-distance))`. This instructs the system to evaluate the given term in each state using the value of that term as the state's heuristic figure of merit. Heuristic depth-first or best-first search can then be applied using this heuristic. Furthermore, if we specify `(set-heuristic-fn (+ (plan-cost) (total-mh-distance)))` and use heuristic breadth-first search the system will perform an A* search as now the term specifying the heuristic ranking computes an admissible heuristic that takes into account the cost of reaching the current state (returned by the built in function `(plan-cost)`.

**Job Shop Scheduling.** Job shop scheduling is a standard operations research scheduling problem. Such problems involve a collection of $k$ jobs $J_1, \ldots, J_k$ and $m$ machines. Each job, $J_i$ contains an ordered sequence of activities $a_i^1, \ldots, a_i^\ell$ (where $\ell$ may depend on $i$). Each activity $a_i^j$ must be executed on a specific machine and requires some quantity of time on that machine. The machines can only execute one activity at a time. The problem is to schedule the activities on the machines so as to complete all activities in as short a time as possible. The only constraint on the schedule is that the activities within each job must be executed in order. That is, activity $a_i^j$ must be executed prior to activity $a_i^\ell$ when $j < \ell$.

This problem is generally solved by search, and it can be configured as a search problem in a number of different ways. One simple configuration is for each state to represent a prefix of a completed schedule—i.e., a schedule that determines what to do for the first $n$ time steps only.

The initial state contains a collection of facts specifying the problem: the number of machines, jobs, activities in each job, the duration and machine of each activity, the earliest start time of each job, and earliest free time of each machine. In

[4] Every different state will have a different denotation for the predicate `at`, but `nextto` is invariant across states. Our implementation does extensive structure sharing between states. Thus invariant predicates, like `nextto`, are specified by a single table shared by all the states.

[5] Note that each variable in the precondition is followed by a specification of the set it is to range over. In some cases part of that specification uses a previously bound variable. Thus, once we have bound `?to` to the blank's position, we can only range `?from` over the neighboring positions.

our implementation there are two types of state transitions for building up a schedule. First, for each machine in some fixed order we pick the next activity to be scheduled on it. Once we have chosen the next activity for each machine we then fix the start and end times of those chosen activities that have all of their predecessors scheduled. Every feasible schedule is reachable by these transitions, and various admissible heuristics can be computed in each state.

For example, consider a problem with two machines, $M_1$ and $M_2$, and two jobs, $J_1$ and $J_2$, each containing two activities, ($a_1$, $a_2$) and ($b_1$, $b_2$). Suppose that the durations and machines of the activities are as follows: $a_1$ 10 units on $M_1$, $a_2$ 5 units on $M_2$, $b_1$ 30 units on $M_2$ and $b_2$ 10 units on $M_1$. From the initial state there are two successor states generated by choosing what to run next on $M_1$: $S_1$ where $a_1$ is to be run next on $M_1$ and $S_2$ where $b_2$ is to be run next on $M_1$. The predicate `(before 1 1 2 2)`, which indicates that job 1/activity 1 ($a_1$) comes prior to job 2/activity 2 ($b_2$), is added to $S_1$ and `(before 2 2 1 1)` to $S_2$. The `before` predicates are used to keep track of the ordering constraints imposed by the choices made. The predicate `(is-before ?j1 ?a1 ?j2 ?a2)` defined by the formula

```
(if-then-else (= ?j1 ?j2)
 (< ?a1 ?a2)
 (exists (?a)
         (is-between ?a ?a1 (num-acts ?j1))
  (exists (?j' ?a') (before ?j1 ?a ?j' ?a')
   (is-before ?j' ?a' ?j2 ?a2))))
```

can then be used to determine if two activities are ordered. `(if-then-else f1 f2 f3)` is simply an abbreviation for `(and (implies f1 f2) (implies (not f1) f3))`. The "then" clause utilizes the fact that all activities within the same job are ordered. The "else" clause examines all activities, `?a`, that lie before `?a1` within `?a1`'s job[6] and recursively determines if any activity known to come after `?a`, (`?j'`/`?a'`), can be recursively shown to come before (`?j2`/`?a2`).

Once the next activity for $M_1$ has been chosen we choose the next activity for $M_2$. For example, $S_2$ might have two successors $S_3$ where $a_2$ runs next on $M_2$ and $S_4$ where $b_1$ runs next on $M_2$. However, state $S_3$ has a cycle. In $S_2$, and thus in $S_3$, $a_1$ must come after $b_2$. Hence, $S_3$ has the ordering $b_1 < b_2 < a_1 < a_2$. Since it also places $a_2$ before $b_1$ we obtain a cycle. `(not (is-before ...))` preconditions are used in the actions to avoid generating cyclic states like $S_3$. So in fact $S_2$ will have a single successor state $S_4$.

After the next activity has been chosen for both machines, the next applicable transition involves scheduling all those activities whose predecessors have been scheduled. In $S_4$, $b_2$ has been chosen next for $M_1$ and $b_1$ next for $M_2$. However, only $b_1$ can be scheduled, as $b_2$ must wait for $b_1$. We can start $b_1$ at time 0 and it will end at time 30. Hence, $M_2$ will next be free at time 30. After this scheduling operation we can choose a next job for $M_2$. There is only one remaining choice, $a_2$. Then we can schedule $b_2$ which can run at the maximum of the next free time of $M_1$, 0, and the finish time of $b_1$, 30. Thus down this path $b_2$ starts at time 30 and ends at time 40, $M_1$ is idle for 30 units, and is next free at time 40. $a_2$ is chosen next for $M_1$, and two final scheduling actions are performed, first $a_1$ (from time 40 to 50) then $b_2$ (from time 50 to 55). The final schedule runs for 55 units.

As the choices are made for what activity to run next, we can compute a low estimate of the time to complete the schedule. For example, in state $S_2$ the ordering commitments have generated the chain $b_1 < b_2 < a_1 < a_2$, thus the schedule in any state that is a successor of $S_2$ must take at least $30 + 10 + 10 + 5 = 55$ units. A recursive function can be defined that checks all ordering chains computing the maximum earliest completion time of the schedule. This value can then be used as a heuristic during search (we first explore states with lower earliest completion time). In particular, this heuristic allows us to avoid expanding $S_2$ as we already know it will yield a poor schedule.

## 6 Advanced Search Algorithms

The evaluator allows us to specify domains and domain heuristics using a clean declarative representation. However, in many cases a search problem can only be solved by modifying the search algorithm. In this section we will show how the evaluator can also be used to build up sophisticated search algorithms from the basic algorithms and the built in search controls outlined in Section 4. The simplest example is the following formula, which defines iterative deepening search.

```
(def-defined-predicate (id)
  (local-vars ?dpt)
  (and
   (set-search-strategy "depth-first")
   (:= ?dpt 1)
   (exists (?i) (pos-int ?i)
    (or (and (set-search-depth-limit ?dpt)
             (plan))
        (implies (> (search-max-depth) ?dpt)
                 (and (:= ?dpt (+ 1 ?dpt))
                      (false)))))))
```

All system commands are encoded as pseudo-predicates so that they can be evaluated by the formula evaluator. The evaluation causes some side effect (some computation is performed, some flag is set, etc.), and true or false is returned (or a value when the command is a pseudo-function) dependent on the outcome of the command. In the `id` search formula the `and` first sets the base search strategy to be depth-first, sets the depth-bound variable `?dpt` to be 1, and then evaluates the existential.

The existential tests its body with `?i` bound to every positive integer, until the body evaluates to true. This corresponds to a while loop (in which the test is negated). The body of the existential is a disjunct. The first disjunct is evaluated. This results in setting the search depth limit to the current value of `?dpt`, then since this pseudo-predicate always evaluates to true, the search engine is invoked by evaluating the predicate `(plan)`.[7] `(plan)` returns true only if it succeeds in solving the problem. If it does return true, the disjunction succeeds, the existential is short-circuited and the evaluation of `(id)` terminates with the value true. Otherwise, (if, e.g., no plan is found at this depth) the second disjunct is evaluated. If a node at a depth greater than the current depth bound

[6] `num-acts` is a function returning the total number of activities in a job.

[7] The name `plan` is used as the system was originally designed to be a planning system.

was encountered during search[8] the depth bound will be incremented and then the formula will fail (`(false)` always evaluates to false). The body of the existential will then be evaluated again with `?i`, and more importantly `?dpt`, incremented. If the current depth bound encompasses the entire search space the second disjunct will succeed and again the evaluation of `(id)` will be terminated.

In a similar manner IDA* [Kor85] search can be defined by using the command `(set-search-heuristic-limit)` (which stops the search engine from expanding any state with heuristic value greater than this limit) and incrementing the heuristic limit to expand increasing parts of the search space.

To illustrate the effectiveness of this method of implementing IDA* search we solved the 2 sliding tile problems that have the longest optimal solutions. These problems were taken from [Rei93], and they both require 31 moves to be solved optimally. With breadth first search, using a heuristic that in each state evaluates to the Manhattan distance heuristic plus the cost of getting to that state (thus we are doing A* search), each problem requires 94 seconds to solve.[9] When using IDA* (with the same heuristic) they took only 32 seconds each.

In the job shop domain we experimented with a hard $10 \times 10$ benchmark problem (given in [BL94]). Heuristic Breadth-first and IDA* search are ineffective as the problem is too hard for the simple heuristic outlined in Section 5. However, the job shop domain has the property that every path in the search tree terminates with a feasible schedule. Thus we can employ depth-first branch and bound (DFBB). The following formula implements this strategy.

```
(def-defined-predicate (dfbb)
 (local-vars ?lb)
 (and
  (set-search-strategy "depth-best-first")
  (set-search-heuristic-limit *none*)
  (forall (?i) (pos-int ?i)
   (and
     (plan)
     (select-final-world)
     (current (:= ?lb (heuristic-fn)))
     (set-search-heuristic-limit (- ?lb 1))))))
```

Basically, the strategy solves the problem using heuristically guided depth-first search. On finding a solution it selects the final state (world) as the current state and then evaluates the heuristic function in that state. In a state where the schedule is complete the heuristic evaluates to the cost of the schedule. It then resets the heuristic bound to one less than this cost. This will force the next search to backtrack until it finds a schedule cheaper than the previous one. If no schedule is found the universal is short-circuited, and the evaluation of `(dfbb)` terminates. In trying `dfbb` on the benchmark problem, schedules of length 1297 (in 19.0 sec.), 1272 (19.1 sec.), 1248 (21.6 sec.), 1230 (19.8 sec.), 1223 (19.8 sec.), 1216 (30.7 sec), 1207 (207.2 sec.), and then 1203 (81.3 sec.) were found. The problem is know to have an optimal solution of length 930, so the final schedule found is within 29% of the optimal. Instead of incrementally decreasing the heuristic bound, it is also possible to do a binary search on the optimal heuristic bound. Such a strategy can also be implemented with a formula, and it finds a schedule of length 1178 (in 968.9 sec.). The times and schedules we obtained are competitive with the application of constraint logic programming to solve this problem as reported in [BL94].

[8] `(search-max-depth)` returns the depth of the deepest node encountered during search. Such nodes may be beyond the current depth bound and thus might not be expanded during search.

[9] All timings were performed on a 200MHz Pentium Pro PC running Windows NT.

## 7 Conclusions

We have described a search system that is based on the idea of evaluating first-order formulas over finite models. We have applied the system to a range of search problems, of which we have only mentioned two prototypical problems. Putting a formula evaluator at the core of the system yields considerable flexibility in modeling domains and in customizing the search procedure.

There has not been sufficient space to compare our approach with logic programming, which also attempts to make direct use of logic. Briefly, however, there are two key differences. First, by evaluating formulas over finite models we gain access to the complete first-order language: we are not restricted to Horn clauses. And second, the system we describe is designed much more specifically to be a system for solving problems requiring search. As a result we are able to provide a user with much more control over the search engine. In comparison, in logic programming languages the underlying search procedure (usually SLD resolution) is fixed and cannot be configured by the user. Instead the user has to program their own search procedure.

## References


[BK98] Fahiem Bacchus and Froduald Kabanza. Using temporal logics to express search control knowledge for planning. Under review, currently available at http://www.lpaig.uwaterloo.ca/~fbacchus/online.html, 1998.

[BL94] Silvia Breitinger and Hendirck C. R. Lock. Using constraint logic programming for industrial scheduling problems. In C. Beierle and L. Plümer, editors, *Logic Programming: Formal Methods and Practical Applications*. Elsevier, 1994.

[Kor85] R. E. Korf. Depth-first iterative-deeping: An optimal admissible tree search. *Artificial Intelligence*, 27:97–109, 1985.

[Ped89] E. Pednault. ADL: Exploring the middle ground between STRIPS and the situation calculus. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning*, pages 324–332, 1989.

[Rei93] A. Reinefeld. Complete solution of the eight-puzzle and the benefit of node ordering in ida*. In *IJCAI*, pages 248–253, 1993.

[Wel94] Daniel S. Weld. An introduction to least commitment planning. *AI Magazine*, 15(4):27–61, 1994.