USER MANUAL
for the
WOULDWORK PLANNING SOFTWARE

3rd Edition

David Alan Brown

ISBN: 9798393149611

Table of Contents

USER MANUAL

for the

WOULDWORK PLANNING SOFTWARE (26.1)

Introduction

This is a user manual for the Wouldwork Planner (a.k.a. The I'd Be Pleased If You Would Work Planner, inspired by my grandson Elias). It covers how to download and install the software, how to write a problem specification, how to run the program, and how to interpret the program's output. This manual is available in booklet or Kindle form for nominal cost at Amazon.com under the same name.

Classical Planning

The typical classical planning problem takes place in a fully specified environment, in which an agent is attempting to plan out a sequence of actions to achieve a complex goal. The planning program, acting on behalf of the agent, analyzes numerous possible paths to the goal, and, if successful, presents a complete action plan from start to finish. Given the potentially large number of actions, environmental objects, and situations, classical planners may discover surprising solutions that elude even careful human analysis.

From a somewhat different perspective, a classical planner can also be viewed as a general problem solver. So it is generally applicable to many state-space search problems not normally regarded as planning problems. The main advantage of using a planner for general problem solving is that the user is relieved of the task of developing a specialized state representation for a problem. A problem state is simply a list of propositions that are true in that state, and the planner reasons about states using predicate logic. Therefore, as long as the

user can express potential problem-solving steps in predicate logic, the planner will independently search for a sequence of steps leading to a solution. However, this generality and convenience in specifying a problem comes with a cost. Unlike a specialized, hand-crafted problem solver, a planner cannot take advantage of sophisticated data representations. It can, however, incorporate problem domain specific features to improve efficiency.

The "classic" classical planning problem, called blocks-world, illustrates the basic operation of a planner. There is really no point in using a planner for such a simple problem, but it is suitable for introducing the basic concepts. Three blocks labeled A, B, and C are each resting on a table T. The goal is to stack the blocks so that A is on B is on C is on T. One possible action is 'put x on y', where x can be a block and y can be another block or the table. The shortest successful plan then contains only two steps: 1) put B on C, and 2) put A on B.

Background information about the problem is provided to the planner by the user in a problem specification file. In general, the problem specification will include a list of possible actions (eg, put x on y), a list of environmental objects (eg, A, B, and C are blocks, while T is a table), relevant properties and relations between objects (eg, x is on y), a state description for recording individual states between actions (typically a collection of facts holding at a particular time), a starting state (eg, A is on T, B is on T, and C is on T), and a goal condition (eg, C is on T, B is on C, and A is on B).

The Wouldwork Planner is designed to efficiently find any (or every) possible solution to a goal, within bounds provided by the user. Within those bounds the search is potentially exhaustive, if run to completion. This approach to planning makes it possible to find a needle in a haystack, if it exists, but is not feasible for extremely complex or large problems, which may require an inordinate amount of search time. However, since computer memory use grows only gradually, the main limitation for large problems is simply user patience.

PART 1. THE WOULDWORK PLANNER USER INTERFACE

Planner Features

The Wouldwork Planner is yet one more in a long line of classical planners. A brief listing of some other well-known classical planners would include Fast Downward, LPG, MIPS-XXL, SATPLAN, SGPLAN, Metric-FF Planner, Optop, SHOP3, and PDDL4j. All of these planners are major developments by small research teams to investigate the performance of a wide variety of planning algorithms. But each has its own limitations in its ability to specify and deal with certain kinds of problems.

In contrast, the Wouldwork Planner was developed not to investigate different planning algorithms, but to extend the baseline capabilities for handling a wider variety of classical problems. It focuses on the data structures and programming interface that allow a user to flexibly and conveniently specify a problem of modest size, and perform an efficient search for a solution. The core planning algorithm itself performs a simple depth-first search through state-space, optimized for efficiently examining potentially millions of states, optionally in parallel. The program attempts to combine many of the interface capabilities of the other planners into one package. Some of the basic features of the user interface include:

- General conformance with the expressive capabilities of the PDDL language for problem specification, with extensions
- Arbitrary object type hierarchies
- Mixing of object types to allow efficient selection of objects during search
- Action rules with preconditions and effects, based on predicate logic

- Full nested predicate logic expressiveness in action rules with quantifiers and equality
- Specification of initial conditions
- Goal specification
- Fluents (ie, continuous and discrete variable quantities & qualities)
- Durative actions taking time to complete
- Exogenous events (ie, happenings occurring independently of the planning agent's actions)
- Temporal plan generation (ie, possible action schedules)
- Global constraint specification, independent of action preconditions
- User function specification for on-the-fly, possibly recursive, embedded computations
- Specification of complementary relations to simplify action rules
- Inclusion of arbitrary Common Lisp code in action rules, constraints, and functions
- User control over search depth
- Identification of shortest length, shortest time, or min/max value plans
- Optional parallel processing to speed up search
- Output/debugging diagnostics describing details of the search

Disclaimer

The Wouldwork Planner is open-source software. It can be freely copied, distributed, or modified as needed. But there is no warrant or guarantee attached to its use. The user therefore assumes all risk in its use. Furthermore, the software was developed for experimental purposes, and has not undergone formal unit testing. Run-time error checking is piecemeal, and latent software bugs surely remain. The software runs only in the Steel Bank Common Lisp (SBCL) programming environment, and so is not currently portable to other environments, since it incorporates some SBCL-unique hooks. Users

may send bug reports, suggestions, or other useful comments to Dave Brown at davypough@gmail.com.

Quickstart with SBCL

1) Install the SBCL Common Lisp release for your computer from http://www.sbcl.org/platform-table.html
2) Install Quicklisp from https://www.quicklisp.org/beta/
3) Start SBCL from the command prompt.
4) At the SBCL prompt, enter (ql:quickload :wouldwork)
5) Enter (in-package :ww) to switch the current package from cl-user to wouldwork.
6) Enter (run-all-tests) to verify everything is loaded and running properly.
7) Review the printout from any of the test problems to see the format of solutions.
8) Look at the sample problem specifications (eg, problem-blocks3.lisp) in the src directory to become acquainted with how problems are defined.

Once you have the SBCL Common Lisp environment installed (I like SBCL because it is open-source and compiles into speedy code), you can add your own problem specifications or modify the source code. Note that the environment will need Quicklisp and ASDF pre-installed as a baseline, plus the :alexandria, :iterate, and :lparallel Quicklisp libraries. Check the file wouldwork.asd for detailed information about what Wouldwork loads.

Also note that SBCL loads with 1024MB = 1GB of memory by default. If a planning problem exits with an out-of-memory condition (heap exhausted), you will need to increase the default by loading SBCL at the command prompt with additional memory, for example:

```
> sbcl –dynamic-space-size 2048
```

which doubles the default (or 4096, 8192, etc, depending how much memory you have).

Planner parameters (*depth-cutoff*, *solution-type*, etc) are either set in the problem specification, or simply left at their default values. Finally, executing (run "problemname") runs the planner on the specification file.

Problem Specification

Most classical planners take as input, a text-like specification of a planning problem provided by the user, and this planner is no exception. The standard problem specification language for classical planning is PDDL, which offers the user a straightforward way to define various types of environmental objects, properties and relations, as well as possible actions, constraints, goals, and initial conditions.

The format of a problem specification file for the Wouldwork Planner is a variation on PDDL to allow some further simplifications and elaborations, but at the expense of being able to represent some very complex planning scenarios falling outside the scope of this planner. The following paragraphs outline the essential sections comprising a problem specification file. The blocks-world problem mentioned previously will serve as a running example. Additional specification sections for more complex problems are left for Part 2 and the Appendix. Advanced planning applications may include arbitrary Common Lisp code along with the standard PDDL sections.

Specifying Environmental Objects & Types

The first requirement is to specify the various objects and object types relevant to the problem domain. In the blocks world, there are three blocks (named A, B, C) and a table (named T). The following definition establishes this background information for the planner:

```
(define-types
  block (A B C)
  table (T)
  support (either block table)) ;where a block can be placed
```

To the left are the object types, and to the right, the particular objects of that type appearing in the problem. The last type (namely, support) is a sometimes useful catch-all type signifying a generic entity of some kind, in this case either a block or table, both of which can support blocks. So A, B, C, and T are all supports. Generic types are often useful for simplifying action rules, to be discussed shortly. The 'either' construct simply forms the union of its argument types. In this example the object names correspond to actual objects in the blocks world, but in general objects can also be values like 1, 2, or 3 or any other Common Lisp programmatic object. Value objects provide a convenient way to represent some discrete properties, such as location coordinates or scaled discrete quantities, which then allows straightforward enumeration of the values in action rules.

Specifying Object Relations

The second requirement is to specify the relevant relations (includes properties) which may hold for and between the various object types. Primary relations are used by the planner to generate and analyze various states of the environment during planning. In the blocks world there is only one relation that needs to be considered, namely whether a block is, or is not, on some support, specified as: (on block support). In other words, in this problem it is possible for a block (A, B, or C) to be on some kind of support location (A, B, C, or T), and particular instantiations of this relation will be present in various states during planning. The full relational specification is then:

```
(define-dynamic-relations
  (on block support))
```

The *dynamic* relation specification indicates that the situation of a block being on a support may change from state to state during the

course of planning. (*Static*—ie, unchanging—relations are discussed later in Part 2. Optionally, all relations can be specified as dynamic, but it is computationally more efficient to separate the two.)

Specifying Possible Actions

The third specification is for the individual actions that the planning agent can take. In this case the agent can take a block and put it either on the table or on another block (in a stack). Actions are always composed at least of a *precondition*, specifying the conditions that must be met before the action can be taken, and an *effect*, specifying changes in the state of the environment after the action is taken. Since the action here is one of putting a block (labeled by a variable like ?block) on a support (like ?support), the precondition must specify that there is not another block on top of ?block, and in addition, that there is not another block on top of ?support, unless ?support is the table. It is conventional, and required in Wouldwork, that typed variables have a question mark (?) prefix. The following precondition expresses these conditions in predicate logic, where ?block signifies the block to be put somewhere, ?b signifies some arbitrary block, ?block-support is the support the block happens to be on currently, and ?support signifies the support on which ?block will be put:

```
(and (not (exists (?b block) ;no block is on ?block
            (on ?b ?block)))
     (on ?block ?block-support) ;?block is on some support
     (not (exists (?b block) ;no block is on the support
            (on ?b ?support)))
```

In plain English, the first condition says that there does not exist a block which is on the block to be moved--ie, that it has a clear top. Later it will be explained how to define auxilliary functions, like cleartop?, which can be used to simplify such action conditions. The second condition verifies that ?block is on some kind of support, labeled ?block-support (either the table or another block); and the third condition checks that that the support place, labeled ?support, itself has a clear top.

It is worth noting that the above action can be expressed much more straightforwardly if fluents and functions are used:

```
(and (cleartop? ?block)))
     (bind (on ?block $block-support))
     (cleartop? $support))
```

These kinds of enhancements to PDDL are discussed in Part 2.

Once the action precondition is established, the action effect then must specify what happens if the action is taken by the planning agent. The effect (in this case after an instance of ?block is put on some ?support), is that the block will now be on that support; and also that the block will no longer be on what was previously supporting it. This effect is concisely expressed as:

```
(assert (on ?block ?support)
        (not (on ?block ?block-support)))
```

In other words, ?block now will be on ?support, and ?block is no longer on ?block-support. The *assert* operator groups multiple changes to the current state together, and automatically arranges and executes them in the proper order to maintain database consistency. Like the precondition, the effect necessarily consists of only one (possibly complex) statement.

The *assert* operator establishes a scope for propositional updates to the current state. An *assert* statement, therefore, can include any number of sub-statements (except another *assert* statement). All of the asserted propositions within its scope will contribute to the next state.

Bringing the precondition and the effect together defines a complete action rule, consisting of six parts. First is the name of the rule (put), second is the duration of the action (where 1 means one unit of time), third is the list of free parameters appearing in the precondition (where a variable, or list of variables alternates with its type, as

discussed later), fourth is the precondition, fifth is the free parameters related to the effect, and sixth is the effect itself:

```
(define-action put
   1
  (?block block (?block-support ?support) support)
  (and (not (exists (?b block)
              (on ?b ?block)))
       (on ?block ?block-support)
       (not (exists (?b block)
              (on ?b ?support))))
  (?block ?block-support ?support)
  (assert (on ?block ?support)
          (not (on ?block ?block-support))))
```

On each planning cycle, each trial action is evaluated by the planner in the order presented in the problem specification. For each action during evaluation, all possible nonredundant arrangements of the precondition parameters are considered. For the above rule, the possible instantiations of ?block are A, B, and C, while the possible ?block-support and ?support values are A, B, C, and T. This leads to a set of 18 (?block ?block-support ?support) arrangement triples generated by the precondition parameters: ((A B C) (A B T) (A C B) (A C T) (A T B) (A T C) (B A C) (B A T) (B C A) (B C T) (B T A) (B T C) (C A B) (C A T) (C B A) (C B T) (C T A) (C T B)). After removing duplicate variable assignments, each is tested against the action preconditions in the current state. If one or more instantiations satisfies the precondition, then the action's effect is executed with those instantiations, producing an update to the current state. (See Part 2: Parameter Lists, for additional details.)

Each successful instantiation represents a possible next step in the planner's path to the goal. As discussed in Part 2, the instantiation procedure is the same for local parameters appearing in quantified expressions (ie, expressions headed by *exists* or *forall* in the precondition or effect), except then the local parameter combinations are instantiated in the context of the current action parameter instantiations.

In general, logical expressions in an action can be headed by any of the usual logical labels—*exists, forall, and, or, not, if, <relation name>, <function name>, <lisp function>* (see Part 2 for a full list). An argument in a logical expression can generally be any *<typed variable>, <fluent variable>* (see Part 2), integer (allows iteration over values in the same way as with typed variables), real number (eg, the value of a fluent), or a *<lisp lambda-expression>* (which is first evaluated to produce the argument in the logical expression in which it occurs).

It is not unreasonable to complain that writing action rules is often difficult. We normally do not reason with predicate logic, and keeping all the variables straight requires some amount of bookkeeping (not to mention trial and error). However, predicate logic has been shown to be a highly expressive language for representing all manner of technical problems, one of which is planning. Its precision is often particularly useful in highlighting subtle errors in thinking. To better handle complex logical operations, the logical statements can often be usefully grouped into functions or independent actions. For example, another option for expressing the action *put* above is to break it into two actions, say *put-block-on-table* and *put-block-on-block*. This would simplify the logic for each rule, but at the potential cost of added debugging, maintenance, and planner processing.

When writing action rules, it may be helpful to remember that when an action runs, all of the parameters for the precondition first will be instantiated for all possible combinations of the variables. Then, the body of the precondition is run for each combination. Every time an instantiation meets the precondition, that instantiation becomes the context for instantiating the effect. The body of the effect then runs for each precondition instantiation, along with any further effect instantiations, thereby registering the specific effects of the action in the next state. In this way, executing a single action rule may give rise to multiple effects, each of which generates a successor state to the current state.

Specifying Initial Conditions

The next required specification characterizes the initial state of the environment, from which planning commences. The initial conditions are simply a list of all facts which are true at time = 0. Below are the initial conditions for the blocks world, indicating that all three blocks are on the table:

```
(define-init
  (on A T)
  (on B T)
  (on C T))
```

Note also that the planner bases its analysis of all environmental states, including the initial state, on the usual closed-world assumption. This means every fact is represented as being either true or false, and never unknown. Therefore, if a fact is true, it will appear in a state representation at a particular time—for example, as (on B T) does in the initial state above. If this statement did not appear in the initial state, it would mean that B was not on the table—that is, that (on B T) was false. This way of representing facts (as either present or absent in a state) makes it easy to check for true and false conditions for a state in the action rules—for example, (not (on B T)) expresses the condition in an action rule that B is not on the table. But (not (on B T)) does not appear in the state itself.

Specifying the Goal Condition

The last requirement is for a goal condition, which is also expressed in terms of a single predicate logic statement. When the planner encounters a state in which the goal is true, the planner records that state, and the path to it from the initial state, as a solution. Depending on whether the user has requested the shortest possible path (in number of steps or duration), or merely the first solution path encountered, the program either continues searching, or exits, respectively. In the blocks world, the goal is to appropriately stack three blocks, which is satisfied when the following fact is true:

```
(define-goal
  (and (on C T)
       (on B C)
       (on A B)))
```

Operational Considerations

This completes the basic specification for the blocks world problem, except to note that the user can tell the planner how deeply to search for a solution. Since the shortest solution involves only two steps—namely, (put B C), and (put A B)—the user could set the *depth-cutoff* at 2 steps. Setting the *depth-cutoff* at 2 will end up generating the minimal number of intermediate states, but it is typically not known in advance how many steps will be required to solve complex problems. Choosing a large value will increase the search time due to greater search depth, but the planner can still find the shortest path to the goal among all paths shorter than or equal to that value. But choosing a value too small may lead to finding no solutions. Some experimentation with different depth cutoffs and partial solutions may be needed to solve complex problems. For the blocks-world problem it is not necessary to specify a depth cutoff, since Wouldwork automatically determines when to stop going deeper.

Another viable option for problems where the total number of possible states is not too large (say under 1,000,000) is to leave the depth cutoff at its default value of 10, and stipulate a graph (rather than tree) search. A graph search avoids reanalyzing previously visited states, thereby potentially shortening the search, but incurs additional overhead that a tree search avoids. The depth cutoff can be gradually increased for subsequent runs. For the blocks problem the total number of possible states is relatively small (ie, all combinations of three blocks arranged in different ways), so setting the depth-cutoff to 0 (meaning go as deep as necessary) is fine, potentially examining all combinations in the worst case.

Solution Types

Wouldwork can adjust its search strategy depending on what kind of solution the user wants. The choice is specified with the *solution-type* parameter—eg, (ww-set *solution-type* first). All Wouldwork settings conform to the Common Lisp "earmuff" notation with asterisk pre- & post-fixes. The options are *first* (find the first encountered solution that satisfies the goal), *min-length* (find the solution with the minimum number of steps/actions from start to finish), *min-time* (find the solution taking the least amount of time to complete, given the time to complete each action), *min-value* (find the solution that minimizes an objective value function), *max-value* (find the solution that maximizes an objective value function), and *every* (find and record every solution). The default is *first*.

Note that for *every*, the number of solutions found may differ, depending on whether a tree search or graph search is performed. Graph search will prune states that are worse than previously visited states, and therefore will not find poorer paths to goals derived from those inferior states. Tree search, however, aims to find all paths to legitimate goals without pruning (but without retracing steps), which may take a long long time.

Each of *min-length, min-time, min-value,* and *max-value* also employ pruning to avoid exploring infeasible paths. During the initial phase of debugging a plan specification however, it may be useful to leave the solution type at its default value of *first*, in order to verify quickly that some/any plan is achievable. Then gradually increase *depth-cutoff* to see if there is any solution within a reasonable depth bound, within a reasonable amount of time.

Note that it is possible to find a min-value or max-value state even if there is no goal specified (ie, the goal is set to nil). For example, see the simple knapsack problem in problem-knap4a.lisp. In this case the search will return the min- or max-value of all states examined. If a

goal is specified, then the min- or max-value satisfying the goal will be returned.

Wouldwork stores in the variable *solutions* all solution paths to all goals reached. But note that many paths may lead to the same goal state. The variable *unique-solutions* will contain the paths to each unique goal state reached. After a planning search completes, enter either variable name at the SBCL prompt to see its contents.

Program Control Settings

There are a number of program settings, like *solution-type* mentioned in the prior paragraph, that the user has control over. All of these settings have default values, but the user can change them as needed. Simply add a statement like (ww-set <setting> <value>) to the problem specification file. For example, (ww-set *solution-type* min-length). The setting names and their initial default values (in the package :ww) are as follows:

***problem* = unspecified (any symbol, string, or number)**
Specifies the name of the problem to be solved. The user will normally include a statement like (ww-set *problem* problem-name) in a problem specification file to identify which problem the specification is about. The statement however is optional.

***problem-type* = planning (planning or csp)**
Specifies whether the problem is a planning problem or a constraint satisfaction problem (csp). Wouldwork uses different search strategies for each.

***depth-cutoff* = 0 (n >= 0)**
Specifies the max search depth; 0 means no cutoff. Searching deeper usually requires more time. For a tree search, if no cutoff is specified, the search along each path will continue until a state repeats, indicating a cyclical (infinitely repeating) path, thereby terminating the path. Searching with a fixed *depth-cutoff* > 0 may be somewhat faster, since then there is no need to check for cyclical paths. Cyclical

paths are not an issue for graph search, since any repeated state (on the current path or not) is automatically pruned.

***tree-or-graph* = graph (tree or graph)**
Specifies whether the search space is expected to be a tree (with no repeated states) or a graph (with repeated states, such that the same state can be reached in more than one way). A paradigmatic example of a problem with a graph search space is the 15-puzzle, since there are many ways to arrive at the same tile configuration by different paths. The 8-queens puzzle, however, exemplifies a problem with a tree search space, if the queens are placed on successive rows one after the other. A previous board configuration of non-attacking queens will never be repeated, because there will always be more queens on the board with each new placement. But neither search strategy will get caught in an infinite loop. Tree search may find more solutions than graph search, since different paths of different lengths to the same goal state are allowed in tree search. Graph search is the default, but if there are few repeated states, tree search will likely be faster.

***solution-type* = first (first, min-length, min-time, min-value, max-value, every)**
Specifies what kind of solution is required from the planner: *first* means stop after the first solution is found; *every* means find all solutions by examining the entire search space, where the search space is determined by the *tree-or-graph* setting; *min-length* means find the path containing the least number of steps (ie, actions); *min-time* means find the solution taking the least amount of time. And *min-value* or *max-value* finds the solution with the minimum or maximum value according to a user-specified optimization value.

***progress-reporting-interval* = 200000 (n > 0)**
Specifies how often to report progress to the terminal during search; reports after each multiple n of states examined; useful for long searches

***debug* 0 (0 <= n <= 5)**
This global variable determines how much debugging information is printed to the terminal during or following a search. The default, 0, means no debugging. A value of 1 prints out the search tree after the search is complete, while higher values print out progressively more information. Setting the value to 5 will interrupt the search for inspection after each processing cycle. This type of debugging is only available in serial (not parallel) search, so first verify that **threads** = 0. See the file `ww-settings.lisp` for additional details. Also see the discussion in section entitled *Plan Monitoring & Debugging* in Part 2.

***probe* nil (any action step)**
Setting this global variable will temporarily interrupt processing whenever Wouldwork takes a particular action. It is often useful for debugging, when you know that a particular step is wrong in the current state. It allows inspection of the current processing cycle and optional actions to help diagnose what is going wrong at a particular point in time. The argument to *probe* is a list consisting of an action name, the instantiation for the action, and the depth. For example, (move (area1 area2) 5) will interrupt processing after Wouldwork considers moving from area1 to area5 at a depth of 5. The resulting state can then be analyzed for errors. Note an exception, however, that setting *probe* must occur at the terminal prompt (REPL), not in the problem.lisp file.

In addition to the program control settings above, there is a global variable that controls the degree of parallelism. The value of this variable can be changed, and is located at the top of the `ww-settings.lisp` file. While the previous program control variables (eg, *depth-cutoff*) can be changed after the Wouldwork files are loaded, the parallel global variable must be set before loading SBCL, since it affects the Lisp image. To change to or from serial computation (non-parallelism), edit `ww-settings.lisp`, exit SBCL altogether, and reload.

***threads* 0 (0 <= n <= N-1)**
This global variable specifies how many parallel threads to use in the search for a solution. The default, 0, means no parallel threads (ie, standard serial search). A value of 1 means use one parallel thread (which effectively amounts to a serial search, but uses the parallel processing mechanisms of Wouldwork—potentially useful for parallel debugging). Assuming the platform CPU can handle up to N threads, it can be useful to choose any number up to N-1, since managing parallelism will consume one thread. Choosing a number greater than N threads is feasible, but probably less efficient.

***branch* nil (0 <= n <= N-1)**
When running parallel threads, *branch* refers to any one of the possible actions taken from the start state. Specifying one of these branches tells Wouldwork to concentrate all parallel processing on this one branch from the start state, and ignore the others. This might be useful for very large problems, since each branch can be searched for a solution individually in separate runs.

***randomize-search* nil (nil or t)**
This global variable specifies whether or not the search through the problem state space will be randomized. (Specifically, whether the order of the child states of a given state is randomized before being placed on the stack of states to be expanded next.) Multiple randomized searches may luckily hit on a solution when a standard (repeatable) search does not. Randomized search is incompatible with a heuristic search, since a heuristic will sort child states in a best-first order.

CTRL-C
To interrupt the search at any time, enter CTRL-C from the keyboard, which throws SBCL into the debugger. Then, entering 0 will continue the search, while entering 1 will stop the search, returning user control to the SBCL prompt. However, the most convenient way to stop a parallel search is to exit SBCL entirely, and then reload.

Program Output

```
* (solve)

working...

New path to goal found at depth = 3

New path to goal found at depth = 2


In problem BLOCKS3, performed TREE search for EVERY solution.

Search process completed normally.

Examined every state up to the depth cutoff.

Depth cutoff = 0

Maximum depth explored = 4

Total states processed = 29

Program cycles (state expansions) = 13

Average branching factor = 9.1


Start state:
((ON A T) (ON B T) (ON C T))

Goal:
(AND (ON C T) (ON B C) (ON A B))


Total solution paths recorded = 2, of which 1 is/are unique
solutions
(Check *solutions* and *unique-solutions* for list of all
solution records.)

Number of steps in a minimum path length solution = 2

A minimum length solution path from start state to goal state:
(1.0 (PUT B C))
(2.0 (PUT A B))

Final state:
((ON A B) (ON B C) (ON C T))


A shortest path solution is also a minimum duration solution.
```

```
Evaluation took:
  0.001 seconds of real time
  0.000000 seconds of total run time (0.000000 user, 0.000000
system)
  0.00% CPU
  5,993,999 processor cycles
  97,888 bytes consed
```

The program first outputs progressive improvements to solutions as planning proceeds (if the user has requested every or an optimal solution). In this case there are two solutions. Next is a statement about the kind of search conducted, whether the program finished normally, and the depth cutoff (ie, consideration of all possible plans less than or equal to the cutoff in path length), where *depth-cutoff* = 0 means no cutoff. The total number of states encountered during planning is also listed. The number of program cycles reports how many sweeps through all possible actions there were. The average branching factor indicates how many new states were generated (on average) from any given state during the search. The number of solution plans found is then reported, with reference to where all the successful plans are stored, if needed.

The sequence of actions in the plan is displayed next as the final planning result. The first number in each plan step indicates the time at which that action occurred, if the actions are specified to take time to complete. Each step contains the action taken along with its arguments. The order of the arguments displayed is the same as the order of parameter variables in the action effect specification, so a parameter specification like (?block block (?block-support ?support) support) would then display as (put A T B) meaning “put block A, which is on block-support T, onto support B”. Lastly, the final state and a summary of computational resources expended wraps up the report.

PART 2: EXPLANATION OF OPTIONAL FEATURES

This section discusses some optional features of the Wouldwork Planner that extend its capability for dealing with certain kinds of planning problems more advanced than the blocks world. Most of these features involve adding supplementary information to the problem specification. In general, later parts of a specification often depend on earlier specifications, so it is best to keep to the following order in the problem specification file. Also, comments can be included in a specification following a semi-colon (;). Any text appearing after a semi-colon on a line of the specification is not processed.

Object Types

Every object constant (eg, A or block1) must have a user-specified type (eg, block), in the *define-types* specification. The type is listed first, followed by a list of object constants of that type. The object constants must all be Common Lisp objects, usually symbols. In general, objects can have more than one type, and types can have subtypes, as a convenience for specifying action rules. For example in the blocks world, A is both a block and a support since it can support other blocks; and support includes both table and block as subtypes. By default, block A is also an instance of the supertype called *something*, since every object is a something, and every type is a subtype of *something*. Thus, a generic statement like (something ?obj) is valid for determining if some object is known to Wouldwork.

Specifications of subtypes are distinguished from object constants by the keyword *either*. For example, support has subtypes (either block table). But block has objects (A B C) or, perhaps in a more perspicuous specification, (block1 block2 block3), where the type is included in the name of the object. To include an object type with no object instances, specify the instances as () or NIL. Instead of listing a large

number of objects for a type, you can also use the operator *compute*, followed by Common Lisp code, to automatically compute the object list (see Problem 8: Triangle Peg Puzzle in the Appendix for an example).

It is also possible to specify dynamic types—ie, types whose instances can vary from state to state. For example in the Triangle Peg Puzzle, the number of pegs left on the board in any state after a jump is one less than in the prior state. If each peg on the board is being checked to see if it can jump any other peg, it makes sense to only check those pegs which are currently on the board in any state, instead of all pegs (including those which have already been removed from the board). To specify such a dynamic type, create a query function to compute the current board pegs (see the following section entitled Query, Update, & Lisp Functions for how to write a function). For the Triangle Peg Puzzle, the dynamic type could look like:

```
(define-types
  peg (peg1 peg2 peg3 …)            ;static type (all pegs)
  current-peg (get-current-pegs?)   ;dynamic type
  …)
```

where the function get-current-pegs? retrieves the current pegs from the current state. The type *current-peg* can then be used in action rules to generate pegs in the same way as the type *peg* will generate all pegs. Alternately, the precondition of the action rule could simply use the dynamic type directly in the parameter list as in (?peg (get-current-pegs?)). The savings in processing time may be significant, since the set of current pegs will always be less than or equal to the set of all pegs.

Object Relations

A fundamental description of any object necessarily includes the properties it has and the relations it has to other objects. Accordingly, each relevant relation (or property) of every object must be included either in a *define-dynamic-relations* or a *define-static-relations*

specification. To illustrate the dynamic/static difference, consider a dynamic relation like (on block support), which could be instantiated by a proposition like (on A T). During the course of planning, the planner maintains a local database of propositions that are true for each state. When the status of A subsequently changes from (on A T) to (on A B), the current database is updated for the next state. However, static propositions, like (block A) indicating that A is a block, never change, and do not need to be maintained in every state. Static propositions are more efficiently stored in a separate database, which the planner can take advantage of if the user defines dynamic and static relations separately. Note, in passing, that in the proposition (block A), the term *block* is being used as a predicate. However, *block* is also a type. Predicates and types are distinguished by their context of use.

The relations of or between objects are specified according to object type, and serve as a template for the propositions that instantiate them. Binary relations are probably the most common, specifying a relation between two object types. For instance, (on block support) expresses a binary relation between blocks and supports. A unary relation expressing a property like (red block), says that blocks can be red. A trinary relation like (separates gate area area) indicates that gates separate two areas, and so on for other higher order relations specifying relations among an arbitrary number of types. Relations can even have no arguments, such as (raining), simply indicating the proposition that it is raining. The type arguments to a relation can also include the *either* construct, as in (color (either block table) hue), indicating that both blocks and tables can have a color, assuming the type hue includes object constants like red, blue, green, etc. The current limit on relation arity is four arguments. Thus, a relation like (connected node node node node node) incorporates one argument too many. All relations are fixed arity.

When a relation includes duplicate types like (separates gate area area), it is interpreted by Wouldwork as a symmetric relation. This is a convenience, since the user then does not need to worry about how

the symmetric types (ie, the 2nd and 3rd area arguments) are instantiated during planning. The user can simply include (separates ?gate ?area1 ?area2) in an action, constraint, function, etc; leaving out the reverse symmetric test for (separates ?gate ?area2 ?area1), since Wouldwork automatically includes both in any database. Whenever one proposition like (separates gate1 area1 area2) becomes true or false, the reciprocal proposition (separates gate1 area2 area1) also becomes true or false. In those special cases where the relation is not symmetric, the default symmetric assumption can be cancelled by marking the relation with a ">" directional suffix—eg, (separates> ?gate ?area1 ?area2). Now the statement will only match the ?area1 and ?area2 variables in the given order.

Relations may also include fluent types, which act as variable types, most commonly for numbers, but also for other defined user types. For example, a relation like (height block $real) might specify that blocks have a height that is a real number, possibly useful for evaluating the net height of a stack of blocks. Fluent variables are identified by their dollar-sign ($) prefix, much as logical variables like ?support are identified by their question mark (?) prefix. The type label (following the $) must be a valid user-defined or Common Lisp type.

Fluent variables, however, operate differently from their logical variable counterparts in actions. While logical variables generate object instantiations, one at a time, for testing in actions; fluent variables are instantiated by looking in the propositional database of the current state for a matching proposition. In the previous example, the relevant database is queried for a proposition matching the pattern (height block $height). If it finds a proposition like (height A 3.2), it instantiates the variable $height with the value 3.2. This instantiation is then available for further evaluation in the action as the value of the variable $height. Note that a relation with fluents is more usefully characterized as a function (being a special kind of relation). For this reason there should be only one proposition in the

database that can match the fluent pattern. Otherwise, the fluent instantiation would be ambiguous among the possible choices.

The returned value of a fluent is not limited to numerical quantities. Any user-defined type (eg, $hue) can also be a fluent (assuming the type hue includes object constants like red, green, blue, etc., and objects can have only one color at a time). A statement like (color block $color) illustrates this way of using a fluent variable. Looking at the actual problem specifications in the Appendix may help clarify the different fluent options and capabilities.

Sometimes, usually for efficiency reasons, a user may specify a relation that includes a lisp container fluent—ie, a list, vector, array, or hash-table. For example, the set of used words in the file `problem-crossword5-11.lisp` is specified as (used-words $hash-table), which records the words in a set that have been tried so far in the search. In these cases, the usual choice to represent a set will be to use a hash-table rather than a list or vector, because the containers are always tested for equality with the predicate #'equalp. The other choices, in general, will not be #'equalp even if they have the same elements. Only use one of the other choices if the order of elements in containers with the same number of elements is always the same—eg, a lexicographic ordering.

Complementary Relations

For a given relation R, the complement of R is the relation expressing the negation of R (ie, not R). Complementary relations come in pairs, such that if any propositional instance of one is true, the corresponding propositional instance other is false. For example, (on switch) and (off switch) are complements. That is, for any instantiation of switch, say switch1, whenever (on switch1) is true, then (off switch1) is false, and vice versa.

If the user stipulates which relations are complements by using the *define-complementary-relations* specification, the planner will

automatically keep track of which complementary propositions are true and which are false. Then, if an action rule concludes that switch1 is on by asserting (on switch1), the planner will automatically remove (off switch1) from the current database. Otherwise, the user must also assert (not (off switch1)) to maintain database consistency.

The previous switch example illustrates the simplest kind of complementary relation, namely one in which the arguments of both relations are the same (ie, switch). For simple complements like on/off, the planner can always work out how to deal with the assertion of any associated proposition, since (on switch1) implies (not (off switch1)), (not (on switch1)) implies (off switch1), (off switch1) implies (not (on switch1)), and (not (off switch1)) implies (on switch1).

However, some complements share only some, or even no, arguments. In the expanded blocks world problem, there is a gripper arm that moves blocks around. In this world, it is useful to know when the gripper is holding a block (in which case it cannot pickup another block) and when the gripper is free. The corresponding relations (holding block) and (not (free)) are therefore complements, since (holding block) implies (not (free)), and (not (holding block) implies (free). But the reverse direction is problematic. While (free) does imply (not (holding ?)), where ? matches anything that the gripper is currently holding, if (not (free)) were asserted, it is indeterminate what the gripper would then be holding. For this reason, all complement specifications are limited to the forward direction only. If the reverse direction is warranted, it must be included separately.

Putting all of the above examples into a specification of complementary relations would look like:

```
(define-complementary-relations
  (on switch) -> (not (off switch))
  (off switch) -> (not (on switch))
  (holding block) -> (not (free)))
```

Logical Statements, Quantifiers, & Doall

Actions, goals, constraints, functions, and other user-defined constructs are composed of logical statements that the planner uses to analyze the current active planning state. All statements must adhere to the previous type and relation definitions as specified. Many statements simply return a true or false value when executed, but others may return a user-defined value (like query function calls), or add and delete propositions from a state (like *assert* statements) . Simple examples of logical statements appear in Part 1. The complete list of admissible logical statement forms in Wouldwork is included in the later section: *The Variety of Logical Statements*.

Of particular note are the quantifier statements *forsome* or *exists* (existential quantification) and *forall* or *forevery* (universal quantification). An existentially quantified statement generates all possible values of the local variables in its parameter list, and executes the statements in its body until some instantiation returns true, at which time the entire statement immediately becomes true. If the body is not true for any instantiation, then the statement returns false. Likewise, a universally quantified statement generates all possible instantiations, but returns true only if every instantiation is true. It returns false immediately if any instantiation is false. In general, quantifier statements, can be nested to an arbitrary level. Additional details about a quantifier's parameter list are discussed in the subsequent section on *Parameter Lists*.

Experience has shown that a third "generating" logical statement form is also sometimes useful, in that it guarantees all instantiations will be processed. This is the *doall* statement, which follows the form of the quantifier statements, for example:

```
(doall (?g gate ?s switch)
  (if (and (controls ?s ?g)
           (off ?s))
    (assert (inactive ?g))
    (assert (active ?g))))
```

This statement tests to see if the switch control for a gate is on or off in the current state, and updates the state accordingly. It always returns true.

Note, however, that in this case, generating and testing all possible values of gate and switch can be expensive. If there are four gates, each of which is controlled by a switch, there are a total of 4x4 = 16 separate instantiations (<gate1,switch1>, <gate1,switch2>, …), each of which is tested in the *if* statement. Since each gate is controlled by only one switch, and a switch can only be on or off, it would be much more efficient to write a statement like:

```
(doall (?g gate)
  (if (and (controls $s ?g)
           (off $s))
    (assert (inactive ?g))
    (assert (active ?g))))
```

which uses a fluent variable ($s) as discussed later. Here, the fluent variable is bound to whatever switch controls the current generated gate ?g. Now there is only one test for each gate, for a total of 4 instantiations.

Durative Actions

For many planning problems, in which the required solution only involves the sequence of planning actions, the time to complete each action is irrelevant. In these cases the second argument in an action specification (after the name) can be 0, indicating instantaneous execution. For other problems where the duration of actions is relevant, the second argument will be a positive real number, signifying the time taken to complete the action. The duration can be specified in any arbitrary time units the user chooses, as long as the units are consistent in all of the action rules (and in any other requirements which are part of the problem specification).

One simple durative action might involve the planning agent's movement from one area to another, where the agent is named *me* below. Assuming movement only occurs between adjacent areas, and the time to move to any new adjacent area is the same, the move action could be expressed as:

```
(define-action move
  2.5  ;moving takes 2.5 time units
  ((?area1 ?area2) area)
  (and (loc me ?area1)
       (adjacent ?area1 ?area2))
  (?area1 ?area2)
  (assert (not (loc me ?area1))
          (loc me ?area2)))
```

As an aside, note that this action involves two variables which are both areas. Since the Wouldwork Planner always interprets variables as names for unique individuals, two distinct variables of the same type will always be instantiated with different objects. This convention means it is not necessary to check for equality with a statement like (not (eql ?area1 ?area2)) in the precondition above. Although this convention does violate a basic tenet of predicate logic, it is consistent with the intuitive understanding that we usually give different names to different objects in a given context to avoid confusion. It is straightforward to override this behavior, however.

Fluent Variables & Variable Binding

A fluent variable, indicated by the prefix $ (as in $support), contrasts with a generated variable, indicated by the prefix ? (as in ?support). While generated variables are instantiated in logical statements by generating all possible instances of the variable (eg, all possible supports for the variable ?support), fluent variables are instantiated by looking up a matching proposition in the current database. For example, in a statement like (on ?block $support), a previously generated value for ?block is first consulted (eg, A), and then the current state database is consulted to determine if A is on something. If it is, that something becomes the value of the fluent $support.

Fluent variables, then, provide a very efficient way of evaluating the truth of a statement, since there is no need to generate and test all possible values of a variable—a simple lookup suffices. However, to avoid ambiguity there should be only one, or no, matching proposition in the database. This limits fluents to appearing in *functional* relations like *on*—a block can only be on one support at a time. When there is a choice between using a generated or fluent variable, the fluent representation is much more efficient.

The object variables in the blocks world problem (namely, ?block and ?support) are discrete variables, in that they can only take on discrete values, such as A, B, C, or T. However, for other problems it is useful to allow continuous variables as well, perhaps representing the height and weight of a block. As discussed previously, in addition to relations like (on ?block $support), relations like (weight ?block $w) or (height ?block-support $h) could also be used in action preconditions, say to account for limitations in the agent's ability to move certain blocks that weigh too much or are stacked too high. The continuous variables $w and $h, distinguished by their required $ prefixes, are technically known as numerical fluents, and in this case can take on positive real number values. Numerical fluents are instantiated by the same lookup procedure as for discrete fluents.

Sometimes there is a choice of whether to represent a variable as a fluent ($) or as a discrete generated (logical) variable. Generally speaking, fluents are most useful for representing data (eg, the height of a block), while logical variables represent symbolic values (eg, individual blocks and supports). But data can also be made discrete if there are a modest number of possible values, say <100. In this case the discrete values must be defined as a type—eg, (define-type height (1 2 3 … 100)) so the action rules can generate the values in parameter lists. Note that processing will slow down somewhat if equality between the values cannot be tested with #'eql (eg, if the values are lists or vectors). Often it is better to simply use a fluent for such values—eg, (define-dynamic-relation (locations $list))—where $list

represents a list of coordinates like ((0 0) (3 1) …). Coordinates can then be dynamically added to or removed from the current locations list in the action rules.

As a more complete example, consider the specification of the classic "jugs" problem. Two jugs of different size are used to measure out a specific amount of water taken from a large reservoir. There are no markings on the jugs to indicate graded amounts. A jug can be filled or emptied completely at the reservoir, or emptied into the other jug to the point where the other jug is full. The goal is to get some reservoir water and pour it back and forth between jugs until a specific target amount of water remains. The first part of the problem specification might look like:

```
(define-types
  jug (jug1 jug2))

(define-dynamic-relations
  (contents jug $integer))

(define-static-relations
  (capacity jug $integer))
```

Here, the relations *contents* (representing the current amount of water in a jug) and *capacity* (representing the maximum amount of water a jug can hold) take a discrete argument (one of the jugs) and a fluent argument (an integer, indicated by the required $ prefix). And both relations are clearly functional, since a jug has only one contents and capacity at any time. The action of filling a jug with water from the reservoir then takes the form of the following rule:

```
(define-action fill
  1
  (?jug jug)
  (and (bind (contents ?jug $amt))
       (bind (capacity ?jug $cap))
       (< $amt $cap))
  (?jug $cap)
  (assert (contents ?jug $cap)))
```

The action uses the fluents $amt and $cap to represent the current amount in a jug and its capacity, respectively. The rule says that if a jug presently has some amount of water in it (possibly 0), and the current amount is less than its capacity (otherwise it is already full), then after filling, its current amount will be its capacity. Other actions such as emptying a jug completely, or pouring the water in one jug into the other jug until it is either empty or the other jug is full (leaving some remaining in the first jug) are left for the complete Appendix jug problem example. The *bind* operator basically tells the planner to look up and assign the current values to all fluent variables in a statement. Those values are then available for use in subsequent statements.

Note that any statement which involves database *lookup*—eg, (loc ?obj $x $y)) in a precondition, query function, if statement, etc—must not include in-line computations: eg, (bind (loc ?obj (1+ $x) $y)). In the former statement the location of an object is being looked up in the database and its retrieved location values assigned to the variables $x and $y. The expression (1+ $x) is not a variable, and thus cannot receive an assignment. But a statement like (assert (loc ?obj (1+ $x) $y)) is fine, since the x coordinate is being computed before *adding* the proposition to the database.

Other local variables can be introduced in any Wouldwork function using the usual *let* form from Common Lisp. If they are special variables (accessible from subfunctions), then simply declare them special as discussed later. Note that local variables appearing in pure Common Lisp expressions or functions should not have the prefix $. As a general rule in Wouldwork statements, always initialize fluent variables using the $ prefix notation—for example, with a *setq* (setq $var 3) or *bind* (bind (blocks $blocks) form. But in Common Lisp statements—such as variable initializations in *loop* or *let*—never use prefixes, or Wouldwork will complain about uninitialized variables.

Global Variables

On some occasions it may be useful to define one or more global variables at the beginning of a problem specification file. Global variables are often convenient, because they are visible from anywhere in the problem specification, and in all threads. Use the normal Lisp operator *defparameter* for single-threaded programs and for programs in which the value of the parameter does not change. Otherwise, to define a global variable which may change or to enable multi-threaded program runs, put in a declaration like the following:

```
(sb-ext:defglobal var-name val-form doc-string)
```

The global var-name can then be safely updated by using the built in SBCL atomic parallel operations such as

```
(sb-ext:atomic-incf var-name delta)     for fixnums, or
(sb-ext:atomic-push object var-name),   for lists
(sb-ext:atomic-pop var-name)
```

See the SBCL manual for further details.

Dynamic (aka Special) Variables

Along with global variables, Common Lisp has a built-in facility for handling dynamic variables. Typically, the variables appearing in a user's problem specification will be *local* (ie, non-dynamic) to each action rule or function, and therefore have no meaning to Wouldwork outside of that rule or function. However, user variables declared as *dynamic* are not so restricted. Wouldwork can use dynamic variables to relay variable values between a parent function and any of its subfunctions. For example, if one function initializes a counter that several other functions subsequently increment depending on various conditions, the incremented values will be available to the initializing function for further processing. Here is a simple example:

```
(define-query bag-item ($item)
  (let ()
    (declare (special $net-price))
    (incf $net-price
          (case $item
            (peas 1.80)
            (carrots 2.25)
            (mangos 3.00)))
    (do-some-other-stuff)))

(define-query purchase-price? ()
  (let (($net-price 0))
    (declare (special $net-price))
    (bag-item 'peas)
    (bag-item 'carrots)
    $net-price))  ;returns the total price
```

Declaring $net-price as dynamic (using the Common Lisp keyword *special*) allows its current value to be shared among the functions that use it. Any special variable must be so declared inside of a *let* or *do* statement.

Parameter Lists

Each action precondition and effect, as well as every *quantified* logical statement has a parameter list which tells the planner how to process the variables appearing in those statements. In the jugs example above, the fill action has a parameter list for the precondition, namely (?jug jug), and for the effect (?jug $cap). The presentation format in a precondition parameter list is always zero or more variables followed by their type. The presentation format in an effect parameter list is always a simple list of zero or more variables. An empty precondition parameter list means execute the body of the action only once, if the precondition is simply (always-true). The order of variable-type presentations in a precondition or quantified statement is arbitrary, but the order of variables in an effect determines the order of the variable instantiations printed out in the steps of a solution. The printout of a step like (fill jug1 5) means fill jug1 to capacity 5, because that is the user-specified order and meaning of variables in the effect parameter list above. Note that the variables appearing in an effect parameter list can come from either the precondition or the effect.

Complex types expressed via the *either* construct are also allowed in precondition parameter lists—eg, ((?pet1 ?pet2) (either dog cat pig) ?owner owner).

As a general rule, every generated variable (?) must be parameterized before it is subsequently used in a logical statement. Thus, a precondition parameter list establishes a *scope* in which its variables can appear in an action precondition, effect, or quantified formula. Fluent variables ($) will never appear in a precondition or quantified formula parameter list, since their values are not generated. Their binding is always established through the use of the *bind* operator. However, fluent variables can appear in action effect parameter lists to control how solution steps are printed, as mentioned above.

In the case of overlapping types, as in a blocks world parameter list like (?block block ?support support), some supports are blocks (all supports except the table T) and all blocks are supports. For this situation, the default is to generate only relevant combinations of a block and a support—eg, (A B) or (A T), but not (T A) or (T T) or (A A)—for checking in an action rule. For example, in the default case, there is no need to include a statement like (different ?block ?support) in the body of a rule.

Whenever the precondition of an action is satisfied for a particular instantiation of its variables, those instantiations are then available for use in the action's effect. In the jugs example, all of the precondition variable instantiations for ?jug, $amt, and $cap are automatically passed to the effect, and therefore do not need to appear in the effect parameter list, unless needed to appropriately print out solution steps.

Wouldwork has several available strategies for generating variables, depending on the requirements of the problem. The default strategy, previously summarized in Part 1: *Specifying Possible Actions*, effectively generates a list of all possible instantiations of each variable, but culls the list to include only unique instantiations. That is,

no two variables will be instantiated with the same value. This comports with the common intuition that we give different names (in this case variable names) to different objects (as values). For example if the parameter list is (?block block (?block-support ?support) support), there are three generating variables—?block, ?block-support, and ?support—each of which can be instantiated with any of the instances from their respective types—block (A B C), block-support (A B C T), and support (A B C T). Therefore by default, instantiations for the three variables are generated from the full cross-product of (block X block-support X support)minus any duplication of values = ((A B C) (A B T) (A C B) (A C T) (A T B) (A T C) (B A C) (B A T) (B C A) (B C T) (B T A) (B T C) (C A B) (C A T) (C B A) (C B T) (C T A) (C T B)). The action rule or quantified formula is then checked for each such instantiation of the three (?) variables.

For some problems it may be possible to reduce the number of instantiations to be checked by considering only different *combinations* of the variable types. To generate combinations, insert it as a keyword at the front of the parameter list like so: (combinations ?block1 block ?block2 block). Now only different combinations of instances will be generated—namely, ((A B) (A C) (B C), greatly reducing the number of instantiations to be checked in a rule. Specifying *combinations* usually only makes sense if there are at least two variables of the same type.

Alternately, the full cross-product of all variables can always be specified with the keyword *products* at the front of the parameter list like so: (products ?digit1 digit ?digit2 digit). Now, all possible combinations of two digits, including when ?digit1 = ?digit2, will be generated and checked in the rule.

On other occasions, it is useful to simply generate *dot-products* from the variable types, such that one instance is selected progressively from each type. Progressive generation is often useful for associating items with attributes. Use the keyword *dot-products* to specify this generation strategy: (dot-products ?obj object ?pos position). Here,

each object will be paired with a corresponding position—eg, ((OBJ1 (3 1)) (OBJ2 (1 1) (OBJ3 (2 3)). Note that to generate dot-products the number of objects must be the same as the number of positions, since they are paired up one-to-one.

Finally, the user can specify that one or more variables are to be generated dynamically—ie, every time an action rule is invoked. For this case, specify a variable type as a query function call that returns the list of values for that variable. For example, instead of specifying a parameter list as (?field field ?word word), specify it as (?field (get-remaining-fields?) ?word (get-remaining-words?)). Then write define-query functions named get-remaining-fields? and get-remaining-words? which return the remaining fields and words for instantiation in the action rule. This may progressively reduce the number of instantiations for multiple invocations of the rule, but is ineffective if the rule is only executed once.

Goals

A planning goal is a logical statement that evaluates to true or false (ie, T or NIL) when applied to a state. Often it will simply consist of a conjunction of literals, all of which must be true to satisfy the goal. Alternately, it can be a complex logical statement or query function which expresses the conditions for recognizing when a situation is true. Every state explored by the planner is checked against the goal condition to see if it satisfies the goal. If it does, the planner stores that final state along with the path to that state as a solution. Depending on whether the user is looking for the first solution or additional solutions, planning will either exit or continue the search, respectively.

When solving an optimization problem that searches for a *min-value* or *max-value* state, a goal specification may be superfluous, since progressively better solutions are produced whenever a new state's $objective-value is better than that of any older state. Leave out the goal specification to tell Wouldwork there is no goal in such a case.

But if a goal is specified, min or max values will only be stored for goal states. Solutions and best states are recorded separately, and progressively. While solutions are recorded in *solutions*, best states are recorded in *best-states*.

Global Constraint

The user may optionally specify a global constraint (or multiple constraints AND-ed or OR-ed together). A global constraint places unconditional restrictions (serving as a kill switch) on planning actions. If a global constraint is ever violated in a state encountered during the planning process, it means that state cannot be on a path to a solution, and the planner must keep looking for some other path to the goal.

A constraint violation occurs when the constraint condition evaluates to NIL (false). In other words, define the constraint to evaluate to T (true), when the constraint is not violated. For example, to have an agent avoid any area of toxic gas, the user could include a constraint like:

```
(define-constraint
  (not (exists (?gas gas ?area area)
         (and (loc me ?area)
              (atmosphere ?gas ?area)
              (toxic ?gas)))))
```

Constraints thus use the same format for expressions as goals. Since constraints are evaluated independently of context, any variables in the constraint must be bound (ie, no free variables). In general, it is more efficient to place constraints locally in the preconditions of individual action rules, since a global constraint is checked in every trial state generated by the planner. However, global constraints can avoid excessive redundancy, and are usually simpler to specify and debug than action preconditions.

Query, Update, and Lisp Functions

One very flexible kind of statement that can appear pretty much anywhere is a function call, used to query or update the current planning state. In the simplest cases, arbitrary built-in Common Lisp function calls may appear along with other Wouldwork logical statements. For example, an expression in a precondition like (< $height1 $height2) evaluates to T (true) if the previously bound value of $height1 is less than $height2. User-defined Common Lisp functions are specified with the usual *defun* expression. In standard programming style, function calls pass the current values of their arguments on to built-in Common Lisp or user-defined functions, which perform some computation before passing the result back to the calling statement. This result then is available for subsequent use in the action rule.

A Wouldwork query or update function provides a convenient way to encapsulate and hide the details of a complex analysis, thus making the problem specification more readable. And since functions are called from action rules or other functions at run time, they can incorporate recursive calls. As usual, function calls support an arbitrary number of arguments, the values of which are passed to the function in the same ordered sequence. Functions must be defined before they are used in a problem specification file.

Query function calls can be used to assign computed values to fluent arguments like $height1 and $height2. For example, the following blocks world query function takes a support as an argument, and computes the elevation of that support. Since blocks can be stacked, the elevation of a block is recursively computable from the height of that block plus the elevation of the block beneath it. If the expression (setf $elev (elevation? state $support)) appears in the precondition of an action, where $support is already instantiated, then the resulting value of $elev after evaluation will be the elevation of that support. The *setf* operator simply assigns the fluent variable to the value returned by the function elevation?. This value might then be used

subsequently in an expression like (< $elev 10) to check whether the total elevation of that support is less than 10.

```
(define-query elevation? ($support)
  (do (bind (height $support $h))
      (bind (on ?support $s))
      (if (not (support $s))
        $h
        (+ $h (elevation? $s)))))
```

The above function first gets the height ($h) of the support ($support). Next, if there is some other support, $s, which $support is on, then it returns $h plus the elevation of $s as the net elevation of $support. Otherwise, it just returns $h as the elevation of $support. The calculation of elevation thus recurses down a stack of blocks, adding in the height of each, until finally the table's height is added. The *bind* statement is used to bind $s to the support that $support is on, if it is currently on a support, or to NIL (meaning the statement is false), if it is not on any support. The *do* operator simply collects the statements into a sequence. The *do* operator is required here, because the body of a Wouldwork function must consist of only one statement (in distinction from a Common Lisp function).

The syntax for a function definition must include a name for the function (eg, elevation?), an argument list of $ and ? variables which are passed into the function (eg, $support), and a body consisting of a single logical statement (possibly composed of multiple sub statements) that returns a value when the function is executed.

As alluded to, there are two basic kinds of user function specifications, depending on whether the function call appears in the precondition or the effect part of an action. The *define-query* specification (eg, elevation?) is for use in preconditions and conditional *if* statements, and returns a value that can be assigned to a fluent variable. The *setf* statement above illustrates this kind of use. However, a more common query use is simply for returning a boolean true or false value, which does not get explicitly assigned. A statement like

```
(if (stable? state ?support)
  (assert (on ?block ?support)))
```

uses the query function stable? to check if a ?support is stable before putting a block on it, where stable? is defined by the user with *define-query*.

Alternately, a *define-update* specification is for functions appearing in an effect, and can be used for analysis leading to changes to the current database. For example, an effect statement like

```
(deactivate-receiver! state ?receiver)
```

might consolidate a number of database conditions and updates associated with deactivation. (While query functions are optionally distinguished by the ? suffix, update functions are optionally distinguished by the ! suffix.) Update functions also return values, but those values are technical updates to the current state (ie, propositions), rather than user specified values. Wouldwork automatically collects the implicit changes, and returns them to the update function's caller. It is up to the action rule to *assert* all changes. In the following update function, the changes appear in the *then* part of the *if-then* statements:

```
(define-update deactivate-receiver! ($receiver)
  (if (active $receiver)
    (do (not (active $receiver)))
        (doall (?g gate)
          (if (controls $receiver ?g)
            (active ?g))))))
```

The *do* statement simply groups several statements together as the *then* part of the *if-then* statement. The propositions (not (active $receiver)) and (active ?g) are updates to the current state.

Next & Finally

By default, all statements in the effect part of an action rule are executed in the context of the variable values established during precondition execution. The order of effect statements therefore normally does not matter, since the state changes resulting from those statements are effectively processed as a group. However, some occasions require a strict sequence of effect actions, for which the operators *next* and *finally* may be appropriate. A common use is for when the initial actions of a rule can automatically trigger additional follow-up actions. In the blocks world, an example might be triggering the collapse of a stack of blocks after a block has been placed on a stack exceeding a certain height. In this case, the last effect statement could look like (finally (assess-stack-stability! ?support)), whose execution is delayed until the preceding effect statements are completed.

Any number of follow-up actions are allowed, each of which will be executed in strict sequence. The convention for including multiple follow-up actions is to label the initial statements with *next*, reserving *finally* for the last statement, although *next* and *finally* are internally processed as synonyms. Also, *next* and *finally* can only take one argument, which must be an update function call, not a generic logic statement.

Assert Statements

Propositions are added to (or retracted from) the current state using the *assert* operator. For example, (assert (on ?block ?support)) will, depending on the current values of ?block and ?support, add a proposition like (on block1 block2). Alternately, a statement like (assert (not (on ?block ?support))) will, depending again on current values, delete a proposition like (on block1 block2) from the current state.

An assert statement establishes a *scope* for other statements that update the current state. And often there are a number of statements that get asserted by an assert statement in the order specified by the user. Each assert statement generates its own update to the current state, which is then stored by Wouldwork for future follow-on processing. This means that the effect of an action rule can contain several assert statements, each with its own scope, and each of which will generate a follow-on state. Thus multiple follow-on states can be conveniently generated by different effects as well as different preconditions in action rules. See `problem-triangle-xyz-one.lisp` for an example of multiple effects all condensed into one large action rule (as opposed to many smaller rules). If the problem is one of optimization using either *min-value* or *max-value*, make sure the variable $objective-value is calculated in each assert statement.

Exogenous Events

Exogenous events are events that happen in the planning environment independent of the planning agent's actions. Typically, the agent must react to or otherwise take into account such happenings along the way to achieving the goal. The user specifies exogenous events before planning begins in a program schedule, which becomes part of the problem specification. As these events are prespecified by the user, they are technically known as *timed initial literals*. The planner uses the schedule to automatically update the state of the environment at the appropriate time. For example, below is a schedule for an automated sentry, named sentry1, which is continuously patrolling three adjacent areas in sequence.

```
(define-happening sentry1
  :events
  ((1 (not (loc sentry1 area6))
      (loc sentry1 area7))
   (2 (not (loc sentry1 area7))
      (loc sentry1 area6))
   (3 (not (loc sentry1 area6))
      (loc sentry1 area5))
   (4 (not (loc sentry1 area5))
      (loc sentry1 area6)))
  :repeat t)
```

Starting in area6 at time = 0 (which is specified separately in the initial state), the planner updates the sentry's location to area7 at time 1. Subsequently, the sentry's location is updated at each time index, to area6 (at time = 2), area5 (at time = 3), back to area6 (at time = 4), area7 (at time = 5), area6 (at time = 6), etc. The keyword :repeat (where t means true) indicates that the sequence is repeated indefinitely. If the keyword :repeat is not included (or is NIL), the exogenous events end with the last event listed. The happenings at each indicated time are simply a list of all state changes occurring at that time.

In this automated sentry problem example (presented fully in the Appendix as the Sentry Problem) the agent must avoid contact with the sentry, but can pass through a patrolled area as long as the sentry is in a different area at the time. Exogenous events often place global constraints on the planning agent, which can be added to the problem specification:

```
(define-constraint  ;avoid kill situation
  (not (exists (?s sentry ?a area)
    (and (loc me ?a)
         (loc ?s ?a)
         (not (disabled? ?s))))))
```

This constraint determines whether there is a sentry and an area, such that the agent (labeled *me*) and the sentry are both located in that area, and the sentry is not disabled (a kill situation). If the constraint is ever not satisfied during planning (ie, evaluates to NIL in any state),

then the constraint is violated, and the planner must backtrack and explore a different path.

There is also a means to temporarily interrupt scheduled happenings, and to dynamically change the sequence of happening events based on certain conditions. The previous constraint shows that whenever a sentry is disabled (eg, by jamming, destruction, or other deactivation), then the constraint is satisfied, and it will be safe to enter the sentry's current area. But becoming disabled means the sentry's program schedule is temporarily or permanently suspended during planning. The user can optionally indicate when such an interruption occurs by specifying interrupt conditions, signaled by the keyword :interrupt. Adding this specification to the happenings then yields the full definition:

```
(define-happening sentry1
  :events ((1 (not (loc sentry1 area6))
              (loc sentry1 area7))
           (2 (not (loc sentry1 area7))
              (loc sentry1 area6))
           (3 (not (loc sentry1 area6))
              (loc sentry1 area5))
           (4 (not (loc sentry1 area5))
              (loc sentry1 area6)))
  :repeat t
  :interrupt (exists (?j jammer)
               (jamming ?j sentry1)))
```

Note that if exogenous events are included in a problem specification, then a graph search strategy cannot be used. This is because states cannot be permanently closed. Entering a previously dead state, may no longer be dead, if it is reached at a different time. Use tree search instead.

Waiting

When there are exogenous events happening in the planning environment, it may sometimes become expedient for an agent to simply wait for the situation to change. For example, in the patrolling sentry problem, the planning agent needs to wait for a time interval

before moving to a patrolled area, to allow the automated sentry to move away from the area. In Wouldwork, waiting is implemented as an action rule, which can be added to the other potential actions included in the problem specification.

```
(define-action wait
  0
  ()
  (always-true)
  ()
  (assert (waiting)))
```

The wait duration is always specified as 0 time units, but will vary during planning analysis, depending on how long the agent must wait in the current situation for the next exogenous event to occur. There are no precondition parameters, since waiting is always a possibility in any situation. Accordingly, the precondition (always-true) will always be satisfied, since that proposition is true in every state. (It is automatically included in the initial conditions for the starting state, and is never subsequently removed.) Likewise in the effect, there is only one update to the current state as a result of executing the wait action, namely (waiting). This proposition will be true during a wait action, but is removed before the next non-wait action is executed.

Since the planner considers action rules in the order presented, the wait action, if it occurs at all, will typically appear as the last action in the problem specification, where it will be given the lowest priority. Although all possible actions are considered on each planning cycle, waiting should probably be considered last to minimize the number of plans containing many waits, all of which are technically acceptable, but possibly inefficient. However, any action can be given priority over the others by moving it up in the list of actions.

Initialization Actions

As already discussed in Part 1, the database for the starting state is initialized via a straightforward listing of true propositions appearing in a *define-init* specification. Similarly, an initialization action, as

defined in a *define-init-action* specification, is an action which is taken only once, also at initialization. An initialization action is useful for adding numerous default propositions to the starting state, in lieu of listing each one individually in a *define-init* specification. For example, the following rule specifies that all sentries are active at initialization:

```
(define-init-action activate-sentries
  0
  (?sentry sentry)
  (always-true)
  ()
  (assert (active ?sentry)))
```

The duration in an initialization action is always set to 0 (actually, any other value is ignored), but otherwise it looks and functions like a normal action. Also, the effect parameter list in an initialization action is usually set to (), since Wouldwork does not printout initialization actions as steps in the planning process.

On rare occasions it also useful to execute a normal action rule one or more times, and then delete it. The action then will no longer take part in generating new states. This allows a form of conditional pre-processing—eg, to generate several possible initial states from the *start-state*—which the other actions can subsequently process. For example, assume there is such an action rule, named add-first-peg, that places a peg on a peg board at each possible initial position. (See `problem-triangle-backwards.lisp`, which solves a triangle peg puzzle backwards.) But after this initial move, the rule is superfluous, since another action takes over placing pegs adjacent to those already on the board. Adding an action rule to delete the now superfluous action will speed up the search:

```
(define-action delete-add-first-peg
  0
  ()
  (always-true)
  ()
  (delete-actions 'add-first-peg 'delete-add-first-peg))
```

This rule will only run once, since it deletes itself as well as the target add-first-peg action. The function named *delete-actions* is reserved and pre-defined in Wouldwork to be available for this purpose. Its arguments are the names of action rules to be deleted.

Multiple Action Effects

A typical action rule has one precondition and one effect. That is, if the precondition is satisfied for some instantiation of its variables, the effect assertions are executed with that instantiation, giving rise to the next state. But it may be convenient, and more efficient, on some occasions to specify several possible effects for each instantiation in one action rule. Then if a precondition is satisfied, all of the intended effects will be independently executed with the precondition instantiation. Each of the multiple effects will thus produce a follow-on state.

Multiple effects are useful for consolidating several action rules into one action, when those several actions have the same precondition. The consolidated action specification is more complex, since it now specifies the previous effects as alternates in the consolidated action. But then the separate actions are redundant, and can be removed. Processing efficiency is also enhanced, since the precondition only needs to be checked once to allow all possible alternate effects.

The simplest way to specify the alternates is to list them with a *do* statement in the consolidated effect like so:

```
(do (if …
      (assert …))
    (if …
      (assert …))
     …)
```

where each set of assertions specifies a state update (if the condition is satisfied). In other words, all update statements within the scope of an *assert* are processed together, separately from other sets of

assertions. But only use *assert* in action rules, not in update functions called from action rules. An example of a multi-effect action rule is provided in the problem specification file named `problem-triangle-heuristic.lisp`.

The Variety of Logical Statements

This section outlines the kinds of logical statement that may appear in actions, goals, initialization actions, functions, exogenous interrupt conditions, and constraints. All planning analysis is done in terms of logic statements, each of which is ultimately translated into the implementation language Common Lisp. Logic statements typically can be nested within other logic statements to an arbitrary extent, although there are some limitations.

```
(loc ?sentry ?area)
```

This is probably the most common form of statement, containing locally generated (?) variables. It deals with the location of a sentry. This basic statement will become instantiated with particular values for ?sentry and ?area during execution (eg, sentry3 and area2) forming a proposition. In a precondition the statement tests whether or not a proposition is present in a state database (returning T or NIL). If true, processing of subsequent precondition statements continues (typically because all the precondition statements are AND-ed together into a conjunction), but if false, the entire conjunction fails. In an effect, such an instantiated statement is instead interpreted as a proposition in the current state database.

```
(assert (loc ?sentry ?area)), or
(assert (not (loc ?sentry ?area))
        (loc me ?area))
```

Assertions are used in action effects to add or retract propositions from the current database. If ?sentry and ?area have been previously instantiated with sentry3 and area2, respectively, then the proposition (loc sentry2 area2) will be either added to or removed from the

database. All unconditional statements within the scope of an *assert* will be asserted.

```
(loc sentry1 ?area)
```
A partially instantiated statement. The operation of partially instantiated statements is as described above. If a statement is fully instantiated, as in (loc sentry1 area3), it is already a proposition, and can be checked in the database directly.

```
(not (loc ?sentry ?area))
```
The negation of a statement in a precondition tests whether an instantiation is explicitly not in the database. In an action effect it means delete the proposition, if present. In the latter case there is no change if the proposition is not present.

```
(loc ?sentry $area)
```
A statement containing fluents (eg, $area), which can be matched with the current variable bindings is in the database. If not found in the database, the statement is false, otherwise true. Note that fluents should only appear in *functional* relations (ie, relations exhibiting only one instantiation in a database at a time). Location is one such relation, since an object normally cannot be located in more than one place at a time.

```
(bind (loc ?sentry $area1))
(not (bind (loc ?sentry $area1)))
```
The bind operator on a fluent statement binds its fluent variables by looking up their values in the current database. The bindings are then available for subsequent use. If a corresponding proposition is not found in the database, the proposition returns false, otherwise true. It is normally used in a precondition or effect *if* statement.

```
(assert (if (active ?sentry)
          (dangerous ?sentry)
          (benign ?sentry)))
```
A conditional *if* statement performs a test and depending on the true/false result selects one or more statements to assert. Its full format consists of three clauses (if <test> <then> <else>), although the <else> clause is optional. In an effect the test is performed on the

current successor state, which is being updated. The judicious use of *if* statements in effects can substantially reduce the total number of action rules required to cover a problem domain.

`(cleartop? ?block)`
A statement that calls a query function is indicated by an optional postfix question mark (?) on a predicate or other computation. The sequence of argument variables must correspond to the order of those in the query function's definition. The statement's value is the value returned by the function.

`(ww-loop for ($x $y) in $positions do …)`
An iterative statement like *loop* in Common Lisp. Allows binding of local variables and respects all the normal loop keywords and constructs like *do*, *collect*, *sum*, etc. Using *ww-loop* as opposed to *loop* allows Wouldwork logical statements to be embedded within the *ww-loop* statement. Note that, in general, Wouldwork logical statements cannot be embedded in Common Lisp statements.

`(setf $elevation (elevation? ?support))`
In this case the query function elevation? takes one argument, ?support, computes the elevation of the support, and returns a value which is stored in the variable $elevation. The *setf* operator performs fluent variable assignment. Subsequent uses of the variable $elevation will refer to this value.

`` `(count ,(1+ *n*)) ``
This is an example of a backquoted statement, indicated by the initial backquote character (`) and one or more commas (,) or comma-ats (,@) internal to the statement. The statement format follows the Common Lisp standard for backquoted expressions. It offers a flexible way to evaluate any Lisp expression following a comma, the results of which are inserted directly into the statement. For example, if the value of *n* has previously been set to 100, then the above statement translates to (count 101). See the define-init section of the `problem-knap19.lisp` file for an example that initializes the capacity of a knapsack.

```
(climbable> ?ladder ?area1 ?area2)
```

When a statement contains two or more generated variables of the same type (viz, ?area1 and ?area2 are both types of area), the planner assumes by default that the predicate and its arguments express a symmetric relation. A symmetric interpretation of this statement means that you can climb from ?area1 to ?area2 using the ?ladder, and from ?area2 to ?area1 using the same ?ladder. (Here, the ladder goes over a wall separating the two areas.) But this default interpretation is probably incorrect in this case. To specify that climbable is not symmetric, attach the direction marker (>) to the predicate, climbable>. Now the agent can use the ladder to go only from ?area1 to ?area2.

```
(exists (?sentry sentry)
  (and (bind (loc ?sentry $area))
       (active ?sentry)))
```

An existential statement operates much like a scaled down action rule, but stops executing after it finds the first instantiation satisfying its conditional part (ie, the *and* statement above). The parameter list can have one or more generator variables, and the condition is tested for each instantiation set. If any instantiation of the condition is true, the entire existential statement is true, otherwise it is false.

```
(forall ((?sentry1 ?sentry2))
  (if (and (bind (loc ?sentry1 $area1))
           (bind (loc ?sentry2 $area2))
           (eql $area1 $area2))
    (assert (conflict ?sentry1 ?sentry2))))
```

A universal statement is the quantified counterpart to an existential statement, and stops executing as soon as any instantiation does *not* satisfy the conditional part. It returns true only if all instantiations of its conditional part are true.

```
(doall (?g gate ?s switch)
  (if (and (controls ?s ?g)
           (off ?s))
    (assert (inactive ?g))
    (assert (active ?g))))
```

A *doall* statement executes its body for all instantiations of its parameter list. It simply returns true when finished.

```
(do (activate-transmitter! ?transmitter)
    (activate-receiver! ?receiver))
```
The *do* operator simply combines two or more statements into a single statement. This statement is common in functions, where the body must be expressed as a single statement.

```
(different ?block ?support)
```
The keyword *different* is a built-in predicate for testing whether two variable instantiations are distinct. Since blocks are also supports, the instantiations for each could be the same—eg, ?block = block1, ?support = block1. In cases like this, it may be useful to test for distinct instantiations using *different*. Here, *different* is synonymous with (not (eql ?block ?support)).

```
(setq $objective-value (+ $current-value $additional-value))
```
A statement like this informs the planner what the net value of a state is when the *solution-type* is either *min-value* or *max-value*. Wouldwork will then search for a state minimizing or maximizing that value. Put this statement inside an assert statement, so it becomes part of the next updated state.

```
(delete-actions 'add-first-peg 'delete-add-first-peg))
```
This statement will delete the specified action rules from further consideration, once the statement is executed. The search then continues without those rules.

```
(always-true)
```
This statement, as it indicates, is always true. It can be used to satisfy a precondition for any and all action rule parameters, which are then passed to the effect for producing assertions.

```
(print ?area), (< $height1 $height2), etc.
```
Any valid Common Lisp function or special form can also appear as the relation in a statement. However, the operator's arguments are limited to other Common Lisp expressions, or previously defined planning parameters.

Plan Monitoring & General Debugging

Even the best laid plans can go awry for unexpected reasons. And it can be difficult to write a logically valid problem specification free from error. Accordingly, Wouldwork offers several options for tracking down mistakes in logic or programming, or just monitoring the planning process.

The global variable named *debug* (in the :ww package) controls the level of information output to the terminal during planning, and for nonparallel search can take a value of 0, 1, 2, 3, 4, or 5 (initially 0 for no debugging output). Level 1 simply outputs the complete search tree of actions attempted, which may be useful for determining where a plan goes wrong. Level 2 outputs level 1 plus the state after each planning step is taken, which may help determine why a particular action failed. Level 3 outputs information about the sequence of steps. Level 4 outputs detailed information about each step. Level 5 outputs the same as Level 4, but temporarily halts program execution after each step, allowing the user to carefully examine each possible action before continuing to the next step. For parallel search there are only two valid values of *debug*, namely 0 (no output) and 1 (basic thread output). Debugging, naturally, slows down the search process as it gathers information about the individual planning states.

Debugging output is displayed in the SBCL REPL window, along with normal output. Since the output can be lengthy, you can capture it to a file for subsequent analysis in a text editor by executing (dribble <filename>)—eg, (dribble "D:/temp.txt")—before executing (run “problem-name”). After the program completes, execute (dribble) to close the file.

To assist with debugging individual actions, constraints, goals, functions, etc, you can also insert arbitrary Common Lisp code among logic expressions—for example to print variable bindings as they are assigned during execution. The following action from the blocks world

problem will print the bindings for two variables of interest using the utility (ut::prt <variable names>) during rule execution:

```
(define-action put
 1
 (?block block ?support support)
 (and (cleartop? ?block)
      (cleartop? ?support)
      (ut::prt ?block ?support))
 (?block block ?support support)
 (assert (on ?block ?support)
         (if (bind (on ?block $s))
           (not (on ?block $s)))))
```

Note that a ut::prt statement always returns the value of its last argument, just like *print* in Common Lisp.

Here is a list of some other potentially useful SBCL commands to load, inspect, or run the Wouldwork data structures:

```
(ww-set *debug* 5)

(ut::show (first *actions*))

(pprint 'goal-fn)  ;or query, happening name, etc

(ut::show *types*)

(ut::show *relations*)

(ut::show *db*)  ;the initial dynamic database
```

As a final resort, there is also a facility for interrupting the search at a particular step (ie, action) in the program. This allows checking whether that step is being performed correctly. For example, specifying (ww-set *probe* (move (area4 area5) 12)), will interrupt the search just after the action of moving from area4 to area5, where the current depth is 12. To tell Wouldwork to interrupt after the third such move action encountered, specify (ww-set *probe* (move (area4 area5) 12 3).

Debugging with Runtime Invariants

An invariant in programming is a statement that, when executed, will check that some condition is true, and signal an error if it is not. Including invariants in a program can often detect subtle mistakes in programming that logical or otherwise careful analysis would overlook. For example, if the program involves initializing a lot of data, a typo that inputs 7 instead of 6 may allow the program to run fine to completion, but perform a faulty analysis.

In Wouldwork, an invariant can check for its condition in every state generated by a rule during planning. An invariant is expressed in Wouldwork as update function (included as a *define-update*

specification in the problem.lisp file), and called in a *next* or *finally* clause in an action rule. As an example, consider the *check-emptys* invariant function in the problem-tiles-7a-heuristic.lisp file:

```
(define-update check-emptys ()
  (do (bind (emptys $emptys)) ;get the value of $emptys
      (unless (alexandria:setp $emptys :test #'equal)
        (troubleshoot "$emptys is not setp: ~A" $emptys))
      (unless (alexandria:sequence-of-length-p $emptys 9)
        (troubleshoot "$emptys length is not 9: ~A"
                      (length $emptys)))))
```

Here, the variable $emptys is always expected to be a list with no duplicates (checked with alexandria:setp) and to have a length of 9 (checked with alexandria:sequence-of-length-p). If either invariant is ever violated, a troubleshooting environment is entered, to help the user determine what led to the error. (Note that the arguments to troubleshoot will be displayed to explain which error was encountered.)

Subsequent to entering the troubleshooting environment, if the user enters 0, Wouldwork displays the immediately preceeding state and the followon states (one of which contains the error). Comparing the two states should highlight what led to the specific error. Once all such errors have been eliminated, the *next* or *finally* clauses in the action can be removed to allow more efficient processing.

Some Useful REPL commands

`sbcl --dynamic-space-size 24000`
Opens SBCL with more than the default 1K of memory.

`(run-test-problems)`
Runs through a varied collection of problems and their solutions, useful for debugging any changes to Wouldwork.

`(list-all)`
Displays the names of all problem specifications in the src directory.

```
(ut::show (first *actions*))
```
Prints the lisp code for a Wouldwork action rule (for debugging).

```
(pprint (get '*goal* 'fn))
```
Prints the lisp code for a Wouldwork function (for debugging).

```
(ut::show *types*)
```
Prints all current type structures as defined in problem.lisp.

```
(ut::show *relations*)
```
Prints all current dynamic relation structures as defined in problem.lisp

```
(ut::show *static-relations*)
```
Prints all current static relations.

```
(ut::show *db*)
```
Prints the initial dynamic propositional database before solving begins.

```
(ut::show *static-db*)
```
Prints the initial static propositional database.

```
(ut::prt s-expr s-expr …)
```
Prints the values of any lisp variables or s-expressions.

```
(ut::profile)
```
Profiles the solution of a problem, to determine which functions are taking the most time. If the search is interrupted midway (eg, by entering CTRL-C) it will be necessary to execute (sb-profile::report) to retrieve the results.

```
(unique-states *best-states* &optional n)
```
Culls the *best-states* (or *solutions*) by removing multiple instances of the same state reached by different paths. Returns the first n such unique states.

Some Useful User Functions to Use in Rules

There are also a few pre-defined Common Lisp functions that the user can include in rules and functions. These functions are simply provided as a convenience so the user doesn't have to write them, if needed. For example, if you want to use a hash table as a representation for a set of elements, there are functional set operations provided such as union, difference, copy, etc. Others may be added later.

```
(different sym1 sym2)
  "Determines whether two symbols represent different objects
   rather than the same object."

(delete-actions &rest names)
  "Deletes named actions from *actions* at run-time."

(get-state-codes)
  "User calls this after finding backwards *solutions* to a
   Problem in order to generate forward paths to those
   solutions."

(backward-path-exists state)
  "Use in a forward search goal to check for the existence of a
   pre-computed backward path."

(vectorize lists)
  "Turns a list of lists into vector of vectors."

(make-ht-set &rest args
             &key (initial-contents nil initial-contents-p)
             &allow-other-keys)
  "Makes a wouldwork hash-table that works as a set container
   for the user."


(copy-ht-set set-ht)
  "Copy a set hash table (with t values)."

(union-ht-set &rest set-hts)
  "Unions hash tables keys. Assumes values are all t and have
   the same :test function."

(set-difference-ht-set ht1 ht2)
  "Returns a new hash table that represents the set difference
   of ht1 and ht2."
```

PART 3: PROBLEM SOLVING STRATEGIES

Solving difficult problems typically involves experimenting with different problem solving strategies. What works for one problem may not work for another. Accordingly, Wouldwork supports a number of strategies that can be tried, ranging from the relatively simple to the relatively complex. These strategies can be used individually, or often in combination, to facilitate the search for one or more solutions.

Brute-Force Search

Implementing a brute-force search is the simplest and easiest problem specification to write. In fact, all problem solving strategies depend on an underlying depth-first brute-force search, so this should always be the first problem specification to develop and validate on a simple version of the main problem.

The principle drawback is that the time required to find a solution is generally exponential in the depth of a solution. Exploring to greater depths, therefore, may be infeasible. A rough estimate of the total size of the search space is available by running a brute-force search on the main problem, and noting the branching factor, b. Combining b with an estimate of the number of steps or depth, d, required to find a solution gives an estimate of the total search space size: b^d. If the size is greater than, say 1 billion, and the number of possible paths to solutions are relatively sparse, then brute-force will likely take too long to be of use.

Parallel Search

Probably the simplest improvement to implement is to use some of the available core processors in your CPU. The search uses only one core by default, but this can be increased as discussed below. Parallel processing probably works best when using a tree (as opposed to a

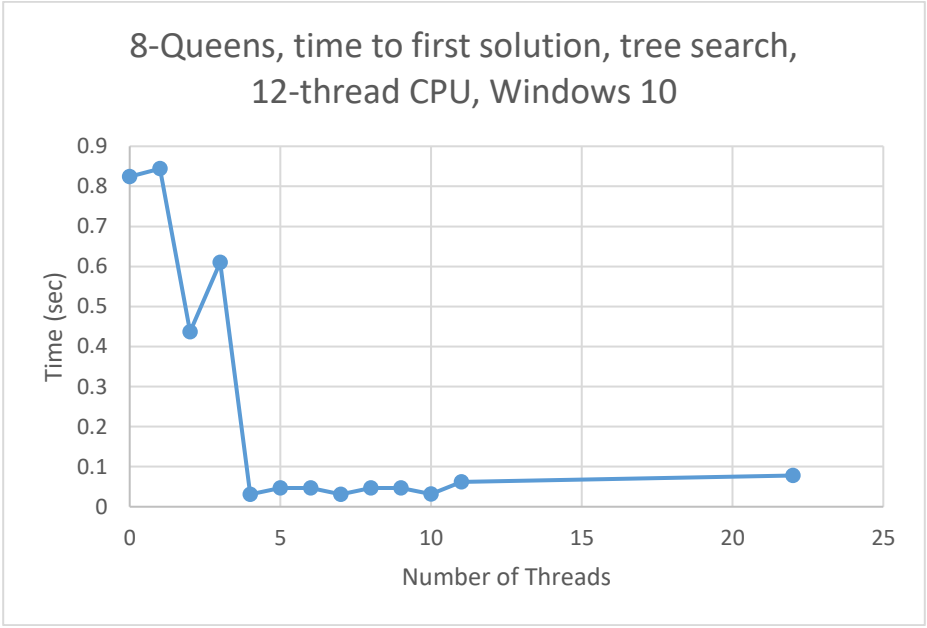
graph) search, because many problem states may be redundantly processed in different threads, thereby reducing the effectiveness of graph search. Also, parallel search will be less efficient than serial search for shallow search trees, because of the constant thread switching overhead. But actual performance tradeoffs will depend on the problem. Parallel search can be combined with any of the other strategies.

During the initial phase of problem development, testing, and debugging, it is prudent to validate simple solutions with serial processing. But for problems with a large search space, parallel processing may be the only way to find any solution. The global variable *threads*, located at the beginning of the file `ww-settings.lisp`, controls how many threads will be allocated to the search. (See Part 1: Program Control Settings for details.)

Wouldwork uses a pool of *threads* to manage parallelism, where each thread in the pool can perform a depth-first search over a subproblem of the originally specified problem. Initially, the main problem is given to the first thread. Then, if there are any idle threads, a subproblem is split off from the current problem, to be picked up by any idle thread. Processing continues on the current problem, with periodic checks for any more idle threads and subsequent splits, until no threads are idle. Each thread may split off a subproblem whenever an idle thread is detected. A thread becomes idle when its search tree is exhausted. Solutions are recorded globally when a thread detects a goal state.

There are a number of different techniques (discussed in the literature on parallel depth-first search) for deciding how to split off a subproblem. Wouldwork always splits off (copies into a subproblem) the top-most state in a search tree. This top state constitutes the shortest unexplored linear path back to the start state. Subsequent state expansions in the subproblem will continue from the last state in this path. This strategy (ie, top linear split) seems to work well for most problems, although massively parallel simulations have shown

that a more complex multi-level strategy (ie, stack split) is generally more efficient overall. In Wouldwork the best parallel speedups seem to correlate with tree (as opposed to graph) search. But note that too many threads, or even *any* multiple threads, can slow down the search for some problems, since there is an overhead for managing parallelism. Experiment on a small version of a problem to determine the optimal number of threads. The following diagram illustrates the parallel speedup for the 8-Queens problem, which maxes out at 4 threads.



Subgoal Search

Oftentimes a problem can be broken down into two (or more) subproblems, such that a solution to the first is a good starting point for solving the second. The goal of the first problem thus becomes the starting state of the second problem. The total solution then is each partial solution in sequence, although the total solution may not be optimal. The objective is to substantially reduce the total depth of the search for each subproblem. See the files `problem-socrates1.lisp`

and `problem-socrates2.lisp` for an example of dividing a problem into two parts, whose individual solutions each require 14 steps, for a total of 28 steps.

Heuristic Search

A heuristic function is a user-defined function that can focus the search for a solution. Adding a heuristic function to Wouldwork may be useful, if the baseline unfocused depth-first search fails to find a solution within a reasonable amount of time. The heuristic function essentially tells Wouldwork to explore the next possible problem states in a best-first, rather than arbitrary, order. Given a current state and the set of all possible states reachable from the current state, the lower-valued next-states will be explored before the higher-valued ones. An effective heuristic function may lead the search to the first solution quickly, although it may not be an optimal solution. But sometimes even a relatively poor heuristic may be enough to make an intractable problem solvable. Note that Wouldwork's heuristic strategy results in a so-called "beam" search, and not an optimal A* search.

The heuristic function analyses a state, and returns a real number indicating how promising that state is for leading to a solution. A lower value indicates a more promising state than a higher value. In general, a lower value may correspond to a state which is closer to a goal. But not always. It is probably better to think of the heuristic value as an indicator of how likely a state is to eventually lead to a solution—ie, its promise.

A heuristic is specified in Wouldwork as a normal *query function* with no arguments, but with the reserved name "heuristic?". (The current state is automatically filled in by Wouldwork.) The typical heuristic query function then looks like the following:

```
(define-query heuristic? ()
  . . .                          ;compute heuristic $value
  (return-from heuristic? $value))
```

See the file `problem-triangle-heuristic.lisp` for an example of a heuristic function.

Note that a heuristic function currently only works with serial, as opposed to parallel, processing.

Optimization Problems

Many problems can be solved using straightforward search techniques, namely, look for the first solution satisfying a goal, or look for every possible solution. However, an optimization problem requires finding a "best" solution; for example, a solution minimizing the number of steps or time taken to achieve a goal, or maximizing or minimizing some user-defined value (eg, net worth or net cost).

To optimize on the number of steps, set solution-type to *min-length* by including (ww-set *solution-type* min-length) in the problem specification file. To optimize on time, set solution-type *to min-time*, and specify for each action rule the time taken to complete that action.

To optimize on some other user-defined value (either minimizing or maximizing the value), set solution-type to either *min-value* or *max-value*, calculate the current value in the relevant action rule, and assign it to the special fluent variable named $objective-value in each assert statement. Once the current value (any real number value) is assigned to $objective-value, the planner will automatically optimize over all planning states. See the knapsack problem in the Appendix for an example of value optimization.

If the search space for a value optimization problem (ie, one involving a solution-type of *min-value* or *max-value*) turns out to be very large, it may become necessary to write a special user-defined "bounding" function to effectively prune suboptimal states. The function is optional, but may considerably shorten the time required for the planner to find the best solution. It should be specified in the

`problem.lisp` file as a *define-query*, and given the name *get-best-relaxed-value?* like so:

```
(define-query get-best-relaxed-value? ()
  ;calculate the best relaxed value here
  ;eg, (setq $relaxed-value (+ $current-value $added-value))
  ;and return the new value
```

The current state is implicitly passed to this function, and the $relaxed-value will be associated with this state. As the name suggests, the function body should efficiently compute a so-called "relaxed" best estimated value for the state, for example using a greedy algorithm. To gain any advantage from using a relaxed value estimate, the function's computational complexity should be less than exponential (preferably linear), since the normal unrelaxed search is typically exponential.

Constraint Satisfaction Problems

A constraint satisfaction problem (CSP) concerns finding a value for each of a number of variables that together satisfy a number of constraints. One popular type of CSP are logic problems, as illustrated in the file problem-captjohn.lisp, where there are multiple constraints on the relative positions of objects. The n-queens problem (eg, problem-8-queens.lisp) is also a CSP, although it is readily formulated here as a simple planning problem with only one constraint.

Wouldwork was originally designed to solve planning problems, in which an agent typically can perform any number of actions in the current situation. The various actions available in various situations are encoded in the action rules, all of which are analyzed by Wouldwork in the current situation. However in a constraint satisfaction problem, it is usually possible to select the next best constraint (ie, action) to consider next, without considering all other constraints, thereby simplifying the search for a solution. CSP problems are thus distinguished from pure planning problems in Wouldwork according to how the constraints are analyzed. But all the constraints are still specified in individual action rules.

Setting *problem-type* to csp—ie, (ww-set *problem-type* csp) — tells Wouldwork to apply this simpler stragegy, as opposed to the default (ww-set *problem-type* planning). Also (ww-set *tree-or-graph* tree) since states are not repeated in constraint satisfaction problems; and leave *depth-cutoff* at 0, since search to a depth equal to the number of rules is required.

Searching with Macro Operators

A macro operator is an action rule that combines two (or more) individual rules into one rule. For example, instead of taking two separate steps in two separate actions, take one big step to end up at the same place. The objective is to reduce the depth and time of the search, since a macro action essentially compresses multiple actions into one.

Macro actions should be added to the individual actions, not replace them. Place all macro actions before any of the baseline individual actions, so they will be considered first during the planning process. Macro actions should be added judiciously, however, since each new action rule adds to the processing overhead for each state. It is something of an art to select useful macros that will quickly lead to a solution, from the myriad potentially useful macros for a problem. (After all, it is even possible to envision a direct solution from the start state to the goal in one step, given prescient knowledge of the search space.)

Examples are provided in the files labeled `problem-triangle-macros.lisp` and `problem-triangle-macros-one.lisp`. The six macros included in these files were added to the baseline triangle peg problems labeled `problem-triangle-xyz.lisp` and `problem-triangle-xyz-one.lisp`, respectively. Each macro jumps two pegs, rather than one at a time. The time savings is achieved by interspersing double jumps among the usual single jumps whenever possible.
A utility program, named *freq*, is provided for analyzing the possible macros associated with a problem. If the problem specification file is set up to find *every* solution to a simple version of the (difficult) target

problem, then the utility program can extract all possible macros that lead to those solutions. The arguments to freq are the macros of the desired length. For example, (freq 2 3) will return all macros of length 2 and length 3, ordered by frequency. The most frequently found macro actions are the ones most likely to be of use in the final problem specification. But intuitions can be deceptive.

Bi-Directional Search

The size of (and time to search) a problem's search space is typically an exponential function of the depth-cutoff (d) and the problem's average branching factor (b): $O(b^d)$. Reducing either b or d can significantly decrease the search space. One way to reduce d is to perform two searches, one forward from a starting state, and one backward from a goal state. If the searches meet in the middle, each search will have searched to a depth of d/2. This is a significant savings, since $2b^{d/2} << b^d$.

But performing a bi-directional search involves extra programming effort. In addition to developing a specification file for the normal forward search, a second specification is required for performing a backward search. Much of the two specifications will be the same, but the action rules will be different, since each of the possible forward actions are now reversed. (However, note that for some problems it may be impossible to reverse the actions.) The depth-cutoff for each direction may also be different. That is, the forward and backward searches will each examine a different portion of the total search space, say to depths d_1 (forward from a starting state) and d_2 (backward from a goal state), but they must meet in the middle somewhere, so $d_{total} = d_1 + d_2$.

In Wouldwork the two problem specifications are run independently, but are coordinated via a user-defined function for efficiently encoding problem states. The general procedure involves 1) running a backward search to some depth d_2 to collect all possible states at that depth reachable from a goal state, 2) running a user-defined function

to efficiently encode all of those states, and 3) running a forward search to a depth $d_1 = d_{total} - d_2$ that hopefully matches one (or more) of the encoded backward states. Wouldwork then joins together the solutions from the forward and backward directions to derive the net solutions.

The following outlines in more detail the procedure for setting up and running a bi-directional search:

- First, see if the main (difficult) problem is amenable to solving with a simpler strategy than bi-directional search. Some of the simpler methods Wouldwork supports include one or more combinations of: brute-force search, sub-goaling, bounded search, macro operators, heuristic search, parallel search, and backwards search.
- The bi-directional approach in Wouldwork begins by developing a basic specification file for the usual forward search. Validate the program on one or more *simple* (ie, reduced) versions of the main problem to make sure the forward search is working correctly and finding all solutions to a specified goal state. (See `problem-triangle-forward6.lisp` for an example.)
- Next, develop the basic specification file for a backward search, where the action rules reverse the actions in the forward action rules. Again, test the backward search from a goal state to a start state on the simple versions of the main problem. The solutions should correspond (in reverse order) to those found in the forward search. (See `problem-triangle-backward6.lisp` for an example.)
- Setup the backwards search specification for the main problem, setting the *solution-type* to *every*. But set the *depth-cutoff* to a small value, and perform a backwards search only to that depth. Modify the goal to detect a solution whenever that depth is reached. Then steadily increase the depth-cutoff until all available memory is used up with every solution recorded at the depth-cutoff. You want to set the final depth-cutoff at the maximum value, d_2, before running out of memory. Run the

backward search with this maximum depth-cutoff to collect all states at that depth (recorded in **solutions**).

- Add a standard Lisp function called *encode-state* to the backward search specification—ie, (defun encode-state (propositions) …), which takes a list of propositions and returns a coded representation of those propositions. The propositions are the list of current (dynamic) propositions defining any state. The encode-state function must return a *unique* code for each state. The most efficient code will be an integer representing the state that distinguishes it from any other state. (See problem-triangle-backward6.lisp for an example.)
- Run the built-in function called *get-state-codes*—ie, execute (get-state-codes). This function will use your encode-state function to generate a code for every state recorded as a solution during the previous backward search at the maximum depth-cutoff. The code and the path to that state are stored in a hash table. (The subsequent forward search will consult the hash table to determine if a forward state matches a backward state leading directly to a goal.)
- Setup the forward search specification for the main problem. Set the depth-cutoff, d_1, so that the forward search will meet any states recorded in the hash table. Given the total depth to achieve a solution is d_{total}, then set $d_1 = d_{total} - d_2$. Also set the goal to record a solution when the forward search reaches d_2 and a backward path exists in the hash table. For the latter test, use the function call (backward-state-exists state), which will return T (ie, true) when there is such a backward state. (Whether or not the forward search matches a backward search state, the forward search will not go deeper than d_1.) Wouldwork will check the current state at depth d_1 to see if it is in the hash table, and paste together the forward path with the backward path as a solution if it is.
- The forward search may benefit from a heuristic or other optimizing strategy to better guide the search toward a

solution--but not in the backward search, since every state at the given depth must be visited.

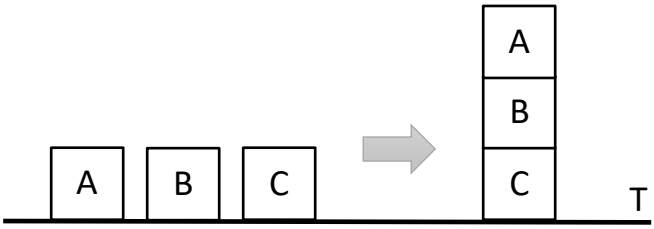
The Wouldwork Search Algorithm

This final section provides a description of the particular algorithm that Wouldwork uses to perform state-space search. It is a variation on the "ordered-search algorithm" developed by Nills Nilsson in "Problem Solving Methods in Artificial Intelligence" (1971), p.55. It offers more options for variations than some of the more modern algorithms:

(1) Put the start node s on a list called OPEN and compute f(s) [the value of a node].
(2) If OPEN is empty, exit with failure, otherwise continue.
(3) Remove from OPEN that node whose f value is smallest and put it on a list called CLOSED. Call this node n. Resolve ties for minimal f values arbitrarily, but always in favor of any goal node.
(4) If n is a goal node, exit with the solution path obtained by tracing back through the pointers, otherwise continue.
(5) Expand node n, generating all of its successors. If there are no successors, go immediately to (2). For each successor n_i, compute f(n_i).
(6) Associate with the successors not already on either OPEN or CLOSED, the f values just computed. Put these nodes on OPEN and direct pointers from them back to n.
(7) Associate with those successors that were already on OPEN or CLOSED, the smaller of the f values just computed and their previous f values. Put on OPEN those successors on CLOSED whose f values were thus lowered and redirect to n, the pointers from all nodes whose f values were lowered.
(8) Go to (2).

APPENDIX: SAMPLE PROBLEMS

1. Blocks World Problem



Develop a plan to stack three blocks on a table.

Blocks Problem Specification:

```
;;;; Filename: problem-blocks3.lisp


;;; Problem specification for a blocks world problem:
;;; stack blocks named A, B, and C on a table named T.


(in-package :ww)  ;required


(ww-set *problem* blocks3)

(ww-set *solution-type* every)

(ww-set *tree-or-graph* tree)


(define-types
    block (A B C)
    table (T)
    support (either block table))


(define-dynamic-relations
    (on block support))
```

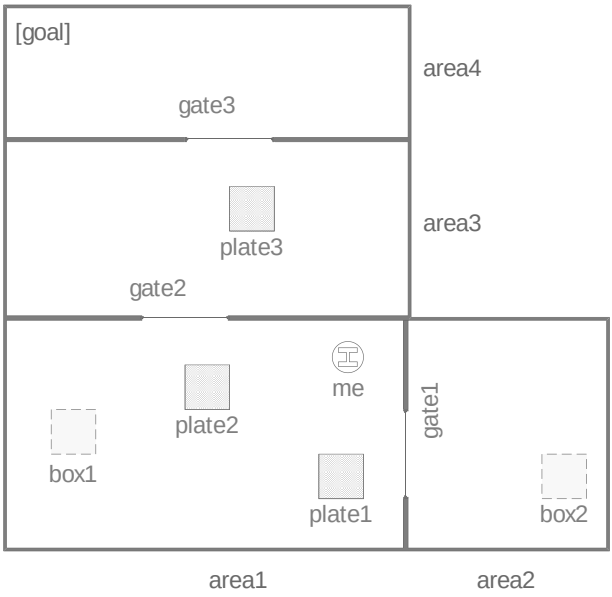
```
(define-static-relations
    (height support $real))


(define-query cleartop? (?block)
  (not (exists (?b block)
         (on ?b ?block))))


(define-action put
    1
  (?block block (?block-support ?support) support)
  (and (cleartop? ?block)
       (on ?block ?block-support)
       (cleartop? ?support)
       (different ?block ?support))
  (?block block ?support support)
  (assert (on ?block ?support)
          (not (on ?block ?block-support))))


(define-init
  (on A T)
  (on B T)
  (on C T))


(define-goal
  (and (on C T)
       (on B C)
       (on A B)))
```

Blocks Problem Solution:

```
Proceed with testing problem-blocks3.lisp?: (y or n) y

working...

New path to goal found at depth = 3

New path to goal found at depth = 2
```

```
In problem BLOCKS3, performed TREE search for EVERY
solution.

Search process completed normally.

Examined every state up to the depth cutoff.

Depth cutoff = 0

Maximum depth explored = 4

Total states processed = 29

Program cycles (state expansions) = 14

Average branching factor = 5.4705844

Start state:
((ON A T) (ON B T) (ON C T))

Goal:
(AND (ON C T) (ON B C) (ON A B))

Total solutions recorded = 2
(Check *solutions* for list of all solution records.)

Number of steps in a minimum path length solution = 2

A minimum length solution path from start state to goal
state:
(1.0 (PUT B C))
(2.0 (PUT A B))

Final state:
((ON A B) (ON B C) (ON C T))


A shortest path solution is also a minimum duration
solution.

Evaluation took:
  0.010 seconds of real time
  0.000000 seconds of total run time (0.000000 user,
0.000000 system)
  0.00% CPU
  34,993,929 processor cycles
  130,896 bytes consed
```

2. Boxes Problem



Move from area1 to area4 by placing boxes on pressure plates, which open the gates.

Boxes Problem Specification:

```
;;;; Filename: problem-boxes.lisp


;;; Problem specification for using boxes to move to an
;;; area through a sequence of gates controlled by
;;; pressure plates.


(in-package :ww)  ;required

(ww-set *problem* boxes)

(ww-set *depth-cutoff* 10)

(ww-set *solution-type* min-length)
```

```
(define-types
  myself     (me)
  box        (box1 box2)
  gate       (gate1 gate2 gate3)
  plate      (plate1 plate2 plate3)
  area       (area1 area2 area3 area4)
  object     (either myself box plate))


(define-dynamic-relations
  (holding myself box)
  (loc (either myself box plate) area)
  (on box plate))


(define-static-relations
  (controls plate gate)
  (separates gate area area))


(define-query free? (?me)
  (not (exists (?b box)
         (holding ?me ?b))))


(define-query cleartop? (?plate)
  (not (exists (?b box)
         (on ?b ?plate))))


(define-query open? (?gate ?area1 ?area2)
  (and (separates ?gate ?area1 ?area2)
       (exists (?p plate)
         (and (controls ?p ?gate)
              (exists (?b box)
                (on ?b ?p))))))


(define-action move
    1
  ((?area1 ?area2) area)
  (and (loc me ?area1)
       (exists (?g gate)
         (open? ?g ?area1 ?area2)))
  ((?area1 ?area2) area)
  (assert (not (loc me ?area1))
          (loc me ?area2)))
```

```
(define-action pickup
    1
  (?box box ?area area)
  (and (loc me ?area)
       (loc ?box ?area)
       (free? me))
  (?box box ?area area)
  (assert (not (loc ?box ?area))
          (holding me ?box)
          (exists (?p plate)
            (if (on ?box ?p)
              (not (on ?box ?p))))))


(define-action drop
    1
  (?box box ?area area)
  (and (loc me ?area)
       (holding me ?box))
  (?box box ?area area)
  (assert (loc ?box ?area)
          (not (holding me ?box))))


(define-action put
    1
  (?box box ?plate plate ?area area)
  (and (loc me ?area)
       (holding me ?box)
       (loc ?plate ?area)
       (cleartop? ?plate))
  (?box box ?plate plate ?area area)
  (assert (loc ?box ?area)
          (not (holding me ?box))
          (on ?box ?plate)))

(define-init
  ;dynamic
  (loc me area1)
  (loc box1 area1)
  (loc box2 area2)
  ;static
  (loc plate1 area1)
  (loc plate2 area1)
  (loc plate3 area3)
  (controls plate1 gate1)
  (controls plate2 gate2)
```

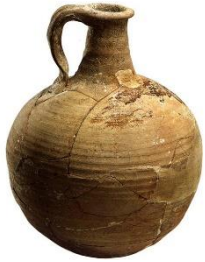
```
  (controls plate3 gate3)
  (separates gate1 area1 area2)
  (separates gate2 area1 area3)
  (separates gate3 area3 area4))


(define-goal
  (loc me area4))
```

Boxes Problem Solution:

```
Proceed with testing problem-boxes.lisp?: (y or n) y

working...

New path to goal found at depth = 10


In problem BOXES, performed GRAPH search for MIN-LENGTH
solution.

Search process completed normally.

Examined only worthwhile states up to the depth cutoff.

Depth cutoff = 10

Maximum depth explored = 10

Total states processed = 59

Repeated states = 22, ie, 37.288136 percent

Program cycles (state expansions) = 27

Average branching factor = 3.2366946

Start state:
((LOC BOX1 AREA1) (LOC BOX2 AREA2) (LOC ME AREA1) (LOC
PLATE1 AREA1) (LOC PLATE2 AREA1) (LOC PLATE3 AREA3))

Goal:
(LOC ME AREA4)
```

```
Total solutions recorded = 1
(Check *solutions* for list of all solution records.)

Number of steps in a minimum path length solution = 10

A minimum length solution path from start state to goal
state:
(1.0 (PICKUP BOX1 AREA1))
(2.0 (PUT BOX1 PLATE1 AREA1))
(3.0 (MOVE AREA1 AREA2))
(4.0 (PICKUP BOX2 AREA2))
(5.0 (MOVE AREA2 AREA1))
(6.0 (PUT BOX2 PLATE2 AREA1))
(7.0 (PICKUP BOX1 AREA1))
(8.0 (MOVE AREA1 AREA3))
(9.0 (PUT BOX1 PLATE3 AREA3))
(10.0 (MOVE AREA3 AREA4))

Final state:
((LOC BOX1 AREA3) (LOC BOX2 AREA1) (LOC ME AREA4) (LOC
PLATE1 AREA1) (LOC PLATE2 AREA1) (LOC PLATE3 AREA3) (ON
BOX1 PLATE3)
 (ON BOX2 PLATE2))

Evaluation took:
  0.030 seconds of real time
  0.000000 seconds of total run time (0.000000 user,
0.000000 system)
  0.00% CPU
  106,012,865 processor cycles
  261,616 bytes consed
```

3. 2-Jugs Problem



2-gallon jug 5-gallon jug reservoir

Fill jugs at reservoir, pour water between jugs until exactly 1 gallon remains. This is set up as a min-time problem, requiring 6 steps, rather than a min-length problem, which requires only 4 steps . Note that it takes a long time to empty a jug (10 time units), since you must walk some distance to do it.

2-Jugs Problem Specification:

```
;;;; Filename: problem-jugs2.lisp


;;; Fluent problem specification for pouring between jugs
;;; to achieve 1 gal given 2-gal jug & 5-gal jug.


(in-package :ww)  ;required

(ww-set *problem* jugs2)

(ww-set *depth-cutoff* 7)  ;set to expected # steps to
goal

(ww-set *solution-type* every)

(ww-set *tree-or-graph* tree)


(define-types
    jug (jug1 jug2))
```

```
(define-dynamic-relations
    (contents jug $integer))

(define-static-relations
    (capacity jug $integer))


(define-action fill
    2
  (?jug jug)
  (and (bind (contents ?jug $amt))
       (bind (capacity ?jug $cap))
       (< $amt $cap))
  (?jug jug $cap fluent)
  (assert (contents ?jug $cap)))


(define-action empty
    10
  (?jug jug)
  (and (bind (contents ?jug $amt))
       (> $amt 0))
  (?jug jug)
  (assert (contents ?jug 0)))


(define-action pour  ;A into B
    3
  ((?jugA ?jugB) jug)
  (and (bind (contents ?jugA $amtA))
       (> $amtA 0)
       (bind (contents ?jugB $amtB))
       (bind (capacity ?jugB $capB))
       (< $amtB $capB))
  (?jugA jug $amtA fluent ?jugB jug ($amtB $capB) fluent)
  (if (<= $amtA (- $capB $amtB))
    (assert (contents ?jugA 0)
            (contents ?jugB (+ $amtB $amtA)))
    (assert (contents ?jugA (- (+ $amtA $amtB) $capB))
            (contents ?jugB $capB))))


(define-init
    (contents jug1 0)
    (contents jug2 0)
    (capacity jug1 2)
    (capacity jug2 5))
```

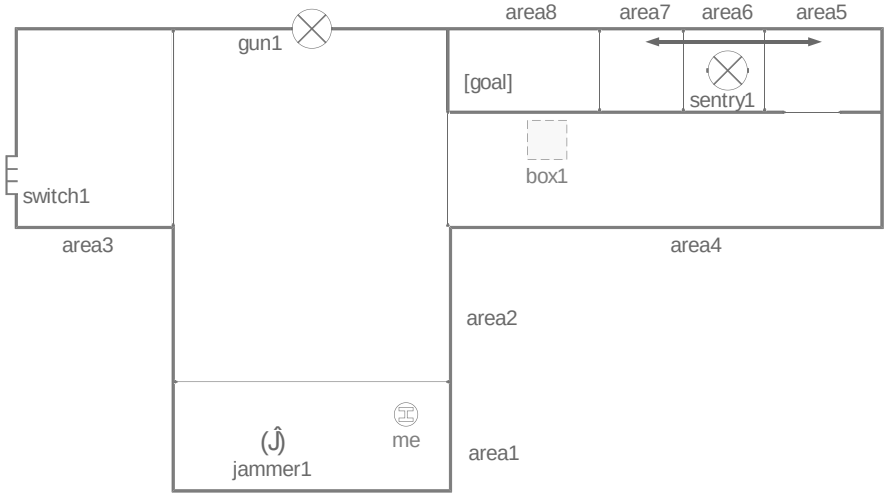
```
(define-goal
    (or (contents jug1 1)
        (contents jug2 1)))
```

2-Jugs Problem Solution:

```
Proceed with testing problem-jugs2.lisp?: (y or n) y

working...

New path to goal found at depth = 6

New path to goal found at depth = 6

New path to goal found at depth = 6

New path to goal found at depth = 4


In problem JUGS2, performed TREE search for EVERY
solution.

Search process completed normally.

Examined every state up to the depth cutoff.

Depth cutoff = 7

Maximum depth explored = 7

Total states processed = 126

Program cycles (state expansions) = 41

Average branching factor = 4.058751

Start state:
((CONTENTS JUG1 0) (CONTENTS JUG2 0))

Goal:
(OR (CONTENTS JUG1 1) (CONTENTS JUG2 1))


Total solutions recorded = 4
(Check *solutions* for list of all solution records.)
Number of steps in a minimum path length solution = 4
```

```
A minimum length solution path from start state to goal
state:
(2.0 (FILL JUG2 5))
(5.0 (POUR JUG2 5 JUG1 0 2))
(15.0 (EMPTY JUG1))
(18.0 (POUR JUG2 3 JUG1 0 2))

Final state:
((CONTENTS JUG1 2) (CONTENTS JUG2 1))



Duration of a minimum time solution = 15.0

A minimum time solution path from start state to goal
state:
(2.0 (FILL JUG1 2))
(5.0 (POUR JUG1 2 JUG2 0 5))
(7.0 (FILL JUG1 2))
(10.0 (POUR JUG1 2 JUG2 2 5))
(12.0 (FILL JUG1 2))
(15.0 (POUR JUG1 2 JUG2 4 5))

Final state:
((CONTENTS JUG1 1) (CONTENTS JUG2 5))

Evaluation took:
  0.024 seconds of real time
  0.000000 seconds of total run time (0.000000 user,
0.000000 system)
  0.00% CPU
  86,286,413 processor cycles
  392,688 bytes consed
```

4. Sentry Problem



Move through an area guarded by an automatic laser gun, so as to jam an automated patrolling sentry, and move to the goal area. Gun1 sweeps area2. Switch1 turns gun1 on/off. Movable jammer1 can jam gun1 or sentry1. Sentry1 repeatedly patrols area5, area6, area7.

Sentry Problem Specification:

```
;;;; Filename: problem-sentry.lisp


;;; Problem specification for getting by an automated
;;; sentry by jamming it.


(in-package :ww)  ;required

(ww-set *problem* sentry)

(ww-set *tree-or-graph* tree)

(ww-set *depth-cutoff* 16)


(define-types
  myself     (me)
  box        (box1)
```

```
  jammer     (jammer1)
  gun        (gun1)
  sentry     (sentry1)
  switch     (switch1)
  red        ()  ;red & green are predicates
  green      ()
  area       (area1 area2 area3 area4 area5 area6 area7
              area8)
  cargo      (either jammer box)
  threat     (either gun sentry)
  target     (either threat))


(define-dynamic-relations
  (holding myself cargo)
  (loc (either myself cargo threat target switch) area)
  (red switch)
  (green switch)
  (jamming jammer target))


(define-static-relations
  (adjacent area area)
  (los area target)    ;line-of-sight exists
  (visible area area)  ;area is visible from another area
  (controls switch gun)
  (watches gun area))


(define-query free? (?myself)
  (not (exists (?c cargo)
               (holding ?myself ?c))))


(define-query passable? (?area1 ?area2)
  (adjacent ?area1 ?area2))


(define-query active? (?threat)
  (not (or (exists (?j jammer)
             (jamming ?j ?threat))
           (forall (?s switch)
             (and (controls ?s ?threat)
                  (green ?s))))))
```

```
(define-query safe? (?area)
  (not (exists (?g gun)
         (and (watches ?g ?area)
              (active? ?g)))))


(define-happening sentry1
  :inits ((loc sentry1 area6))
  :events
  ((1 (not (loc sentry1 area6)) (loc sentry1 area7))
   (2 (not (loc sentry1 area7)) (loc sentry1 area6))
   (3 (not (loc sentry1 area6)) (loc sentry1 area5))
   (4 (not (loc sentry1 area5)) (loc sentry1 area6)))
  :repeat t
  :interrupt (exists (?j jammer)
               (jamming ?j sentry1)))


(define-constraint
  ;Constraints only needed for happening events that can
  ;kill or delay an action. Global constraints included
  ;here. Return t if constraint satisfied, nil if
  ;violated.
  (not (exists (?s sentry ?a area)
         (and (loc me ?a)
              (loc ?s ?a)
              (active? ?s)))))


(define-action jam
    1
  (?target target ?area2 area ?jammer jammer ?area1 area)
  (and (holding me ?jammer)
       (loc me ?area1)
       (or (los ?area1 ?target)
           (and (loc ?target ?area2)
                (visible ?area1 ?area2))))
  (?target target ?jammer jammer ?area1 area)
  (assert (not (holding me ?jammer))
          (loc ?jammer ?area1)
          (jamming ?jammer ?target)))
```

```
(define-action throw
    1
  (?switch switch ?area area)
  (and (free? me)
       (loc me ?area)
       (loc ?switch ?area))
  (?switch switch)
  (assert (if (red ?switch)
              (do (not (red ?switch))
                  (green ?switch))
              (do (not (green ?switch))
                  (red ?switch)))))


(define-action pickup
    1
  (?cargo cargo ?area area)
  (and (loc me ?area)
       (loc ?cargo ?area)
       (free? me))
  (?cargo cargo ?area area)
  (assert (not (loc ?cargo ?area))
          (holding me ?cargo)
          (exists (?t target)
            (if (and (jammer ?cargo)
                     (jamming ?cargo ?t))
              (not (jamming ?cargo ?t))))))


(define-action drop
    1
  (?cargo cargo ?area area)
  (and (loc me ?area)
       (holding me ?cargo))
  (?cargo cargo ?area area)
  (assert (not (holding me ?cargo))
          (loc ?cargo ?area)))
```

```
(define-action move
    1
  ((?area1 ?area2) area)
  (and (loc me ?area1)
       (passable? ?area1 ?area2)
       (safe? ?area2))
  ((?area1 ?area2) area)
  (assert (not (loc me ?area1))
          (loc me ?area2)))


(define-action wait
    0  ;always 0, wait for next exogenous event
  (?area area)
  (loc me ?area)
  ()
  (assert (waiting)))


(define-init
  ;dynamic
  (loc me area1)
  (loc jammer1 area1)
  (loc switch1 area3)
  (loc box1 area4)
  (red switch1)
  ;static
  (always-true)
  (watches gun1 area2)
  (controls switch1 gun1)
  (los area1 gun1)
  (los area2 gun1)
  (los area3 gun1)
  (los area4 gun1)
  (visible area5 area6)
  (visible area5 area7)
  (visible area5 area8)
  (visible area6 area7)
  (visible area6 area8)
  (visible area7 area8)
  (adjacent area1 area2)
  (adjacent area2 area3)
  (adjacent area2 area4)
  (adjacent area4 area5)
  (adjacent area5 area6)
  (adjacent area6 area7)
  (adjacent area7 area8))
```

```
(define-goal
  (loc me area8))
```

Sentry Problem Solution:

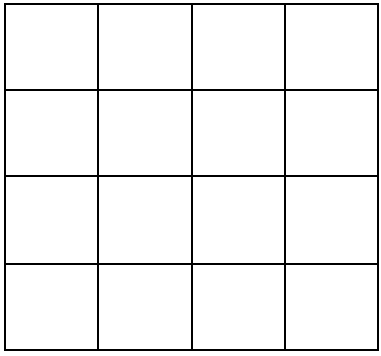
```
Proceed with testing problem-sentry.lisp?: (y or n) y

working...

New path to goal found at depth = 16


In problem SENTRY, performed TREE search for FIRST
solution.

Search ended with first solution found.

Depth cutoff = 16

Maximum depth explored = 16

Total states processed = 2,023

Program cycles (state expansions) = 629

Average branching factor = 9.153569

Start state:
((LOC BOX1 AREA4) (LOC JAMMER1 AREA1) (LOC ME AREA1) (LOC
SWITCH1 AREA3) (RED SWITCH1))

Goal:
(LOC ME AREA8)


Total solutions recorded = 1
(Check *solutions* for list of all solution records.)

Number of steps in first solution found: = 16

Duration of first solution found = 16.0
```

```
Solution path of first solution found from start state to
goal state:
(1.0 (PICKUP JAMMER1 AREA1))
(2.0 (JAM GUN1 JAMMER1 AREA1))
(3.0 (MOVE AREA1 AREA2))
(4.0 (MOVE AREA2 AREA3))
(5.0 (THROW SWITCH1))
(6.0 (MOVE AREA3 AREA2))
(7.0 (MOVE AREA2 AREA1))
(8.0 (PICKUP JAMMER1 AREA1))
(9.0 (MOVE AREA1 AREA2))
(10.0 (MOVE AREA2 AREA4))
(11.0 (WAIT 1.0))
(12.0 (MOVE AREA4 AREA5))
(13.0 (JAM SENTRY1 JAMMER1 AREA5))
(14.0 (MOVE AREA5 AREA6))
(15.0 (MOVE AREA6 AREA7))
(16.0 (MOVE AREA7 AREA8))

Final state:
((GREEN SWITCH1) (JAMMING JAMMER1 SENTRY1) (LOC BOX1
AREA4) (LOC JAMMER1 AREA5) (LOC ME AREA8) (LOC SWITCH1
AREA3))

Evaluation took:
  0.141 seconds of real time
  0.125000 seconds of total run time (0.093750 user,
0.031250 system)
  [ Run times consist of 0.047 seconds GC time, and 0.078
seconds non-GC time. ]
  88.65% CPU
  508,526,133 processor cycles
  163,691,936 bytes consed
```

5. 4-Queens Problem

rows: 1, 2, 3, 4
columns: 1, 2, 3, 4

queen1 queen2 queen3 queen4



Place four queens on the board, so that no two queens are attacking each other. Place the first queen on row 1, the second on row 2, etc, until all queens are properly placed.

4-Queens Problem Specification:

```
;;;; Filename: problem-queens4.lisp

;;; Problem specification for 4-queens.

(in-package :ww)  ;required

(ww-set *problem* queens4)

(ww-set *depth-cutoff* 4)

(ww-set *solution-type* every)

(ww-set *tree-or-graph* tree)

(define-types
  queen   (queen1 queen2 queen3 queen4)
  column  (1 2 3 4))
```

```
(define-dynamic-relations
  (loc queen $integer column)
  (placed queen)
  (next-row $integer))


(define-action put
    1
  (?queen queen ?column column)
  (and (not (placed ?queen))
       (bind (next-row $row))
       (not (exists (?q queen ?c column)
              (and (placed ?q)
                   (bind (loc ?q $r ?c))
                   (or (= $r $row)
                       (= ?c ?column)
                       (= (- $r $row) (- ?c ?column))
                       (= (- $r $row)
                          (- ?column ?c)))))))
  (?queen queen $row fluent ?column column)
  (assert (loc ?queen $row ?column)
          (placed ?queen)
          (next-row (1+ $row))))


(define-init
    (next-row 1))


(define-goal
  (next-row 5))
```

4-Queens Problem Solution:

There are exactly 48 unique solutions to the 4-queens problem taking into account all possible successful arrangements of four distinct queens labeled queen1, queen2, queen3, queen4. Considering only the arrangements of unlabeled queens on the board, however, there are only two distinct successful arrangements. The shortest solution path listed below gives one of the 48 possible solutions. Each step in the solution below corresponds to placing a queen in successive rows 1-4. Thus, the first action labeled (PUT QUEEN4 1 3) means put

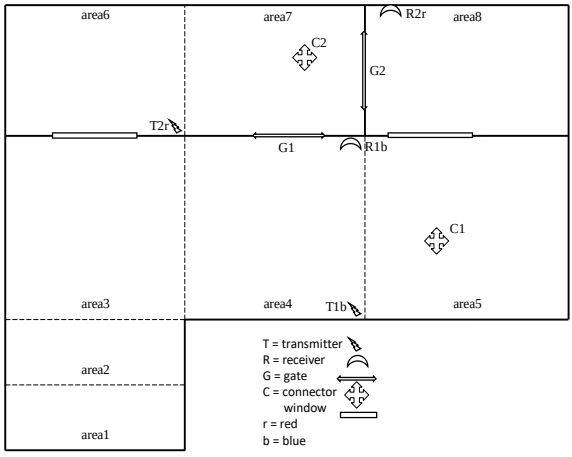
queen4 in the 1st row of the 3rd column. For larger versions of this problem (such as with eight queens), making sure *tree-or-graph* = 'tree, should provide the only symmetrical 8-queens solution in about 1 second.

```
Proceed with testing problem-4queens.lisp?: (y or n) y

working...

New path to goal found at depth = 4
New path to goal found at depth = 4
New path to goal found at depth = 4
New path to goal found at depth = 4
New path to goal found at depth = 4
New path to goal found at depth = 4
New path to goal found at depth = 4
New path to goal found at depth = 4
New path to goal found at depth = 4
New path to goal found at depth = 4
New path to goal found at depth = 4
New path to goal found at depth = 4
New path to goal found at depth = 4
New path to goal found at depth = 4
New path to goal found at depth = 4
New path to goal found at depth = 4
New path to goal found at depth = 4
New path to goal found at depth = 4
New path to goal found at depth = 4
New path to goal found at depth = 4
New path to goal found at depth = 4
New path to goal found at depth = 4
New path to goal found at depth = 4
New path to goal found at depth = 4
New path to goal found at depth = 4
New path to goal found at depth = 4
New path to goal found at depth = 4
New path to goal found at depth = 4
New path to goal found at depth = 4
New path to goal found at depth = 4
New path to goal found at depth = 4
New path to goal found at depth = 4
New path to goal found at depth = 4
New path to goal found at depth = 4
New path to goal found at depth = 4
New path to goal found at depth = 4
```

```
New path to goal found at depth = 4
New path to goal found at depth = 4
New path to goal found at depth = 4
New path to goal found at depth = 4
New path to goal found at depth = 4
New path to goal found at depth = 4
New path to goal found at depth = 4
New path to goal found at depth = 4
New path to goal found at depth = 4
New path to goal found at depth = 4
New path to goal found at depth = 4
New path to goal found at depth = 4


In problem QUEENS4, performed TREE search for EVERY
solution.

Search process completed normally.

Examined every state up to the depth cutoff.

Depth cutoff = 4

Maximum depth explored = 4

Total states processed = 185

Program cycles (state expansions) = 114

Average branching factor = 15.177387

Start state:
((NEXT-ROW 1))

Goal:
(NEXT-ROW 5)


Total solutions recorded = 48
(Check *solutions* for list of all solution records.)

Number of steps in a minimum path length solution = 4
```

```
A minimum length solution path from start state to goal
state:
(1.0 (PUT QUEEN4 1 3))
(2.0 (PUT QUEEN3 2 1))
(3.0 (PUT QUEEN2 3 4))
(4.0 (PUT QUEEN1 4 2))

Final state:
((LOC QUEEN1 4 2) (LOC QUEEN2 3 4) (LOC QUEEN3 2 1) (LOC
QUEEN4 1 3) (NEXT-ROW 5) (PLACED QUEEN1) (PLACED QUEEN2)
(PLACED QUEEN3)
 (PLACED QUEEN4))


A shortest path solution is also a minimum duration
solution.

Evaluation took:
  0.071 seconds of real time
  0.000000 seconds of total run time (0.000000 user,
0.000000 system)
  0.00% CPU
  255,366,108 processor cycles
  785,264 bytes consed
```

6. Smallspace Problem



This is an example of a rather complex and lengthy specification that illustrates the integration of many Wouldwork planner features. It has served as a useful testbed for new features. It is a partial solution to a problem situation given in The Road to Gehenna, an add-on module for the Talos Principle game. The objective is to position a set of connectors such that they relay a laser beam from a transmitter source to a receiver of the same color which controls a gate. Once the receiver detects a beam of the proper color, it opens a gate. The goal is to move from area5 to area8.

Smallspace Problem Specification:

```
;;;; Filename: problem-smallspace.lisp


;;; Problem specification (in Talos Principle)
;;; for the small space problem in Road to Gehenna sigil
;;; dome. First leg to area8.


(in-package :ww)  ;required
```

```
(ww-set *problem* smallspace)

(ww-set *solution-type* min-time)

(ww-set *depth-cutoff* 19)


(define-types
  myself       (me)
  gate         (gate1 gate2)
  barrier      (nil)
  jammer       (nil)
  connector    (connector1 connector2)
  box          (nil)
  fan          (nil)
  gears        (nil)
  ladder       (nil)
  rostrum      (nil)
  hue          (blue red)
  transmitter (transmitter1 transmitter2)
  receiver     (receiver1 receiver2)
  area         (area1 area2 area3 area4 area5 area6 area7
                area8)
  cargo        (either connector)
  target       (either gate gears)
  divider      (either gate barrier)
  terminus     (either transmitter receiver connector)
  fixture      (either transmitter receiver)
  station      (either fixture gate)
  support      (either box rostrum))


(define-dynamic-relations
  (holding myself $cargo)
  (free myself)
  (loc (either myself cargo) $area)
  (on (either myself cargo) $support)
  (attached fan gears)
  (jamming jammer $target)
  (connecting connector terminus)
  (active (either connector receiver gate))
  (color terminus $hue))
```

```
(define-static-relations
  ;agent can always move unimpeded between adjacent areas
  (adjacent area area)
  (locale fixture area)
  (separates divider area area)
  (climbable> ladder area area)
  ;(height support $real)
  (controls receiver $gate)
  ;clear los from an area to a gate/fixture
  (los0 area (either gate fixture))
  (los1 area divider (either gate fixture))
  (los2 area divider divider (either gate fixture))
  ;could see a mobile object in an area from a given area
  (visible0 area area)
  (visible1 area divider area)
  (visible2 area divider divider area))


(define-complementary-relations
  (holding myself $cargo) -> (not (free myself)))


;;;; QUERY FUNCTIONS ;;;;


(define-query source? (?terminus)
  (or (transmitter ?terminus)
      (and (connector ?terminus)
           (active ?terminus))))


(define-query los-thru-2-dividers? (?area ?station)
  (exists ((?d1 ?d2) divider)
    (and (los2 ?area ?d1 ?d2 ?station)
         (or (and (barrier ?d1)
                  (barrier ?d2))
             (and (barrier ?d1)
                  (gate ?d2)
                  (not (active ?d2)))
             (and (barrier ?d2)
                  (gate ?d1)
                  (not (active ?d1)))
             (and (gate ?d1)
                  (not (active ?d1))
                  (gate ?d2)
                  (not (active ?d2)))))))
```

```
(define-query los-thru-1-divider? (?area ?station)
  (exists (?d divider)
    (and (los1 ?area ?d ?station)
         (or (barrier ?d)
             (and (gate ?d)
                  (not (active ?d)))))))


(define-query los? (?area ?station)
  (or (los0 ?area ?station)
      (los-thru-1-divider? ?area ?station)
      (los-thru-2-dividers? ?area ?station)))


(define-query visible-thru-2-dividers? (?area1 ?area2)
  (exists ((?d1 ?d2) divider)
    (and (visible2 ?area1 ?d1 ?d2 ?area2)
         (or (and (barrier ?d1)
                  (barrier ?d2))
             (and (barrier ?d1)
                  (gate ?d2)
                  (not (active ?d2)))
             (and (barrier ?d2)
                  (gate ?d1)
                  (not (active ?d1)))
             (and (gate ?d1)
                  (not (active ?d1))
                  (gate ?d2)
                  (not (active ?d2)))))))


(define-query visible-thru-1-divider? (?area1 ?area2)
  (exists (?d divider)
    (and (visible1 ?area1 ?d ?area2)
         (or (barrier ?d)
             (and (gate ?d)
                  (not (active ?d)))))))


(define-query visible? (?area1 ?area2)
  (or (visible0 ?area1 ?area2)
      (visible-thru-1-divider? ?area1 ?area2)
      (visible-thru-2-dividers? ?area1 ?area2)))
```

```
(define-query connectable? (?area ?terminus)
  (or (los? ?area ?terminus)  ;from connector in area to
terminus
      (and (connector ?terminus)
           (exists (?a area)
             (and (loc ?terminus ?a)
                  (visible? ?area ?a))))))


(define-query passable? (?area1 ?area2)
  (or (adjacent ?area1 ?area2)
      (exists (?b (either barrier ladder))
        (and (separates ?b ?area1 ?area2)
             (free me)))  ;must drop cargo first
      (exists (?g gate)
        (and (separates ?g ?area1 ?area2)
             (not (active ?g))))))


;;;; UPDATE FUNCTIONS ;;;;


(define-update activate-connector! (?connector ?hue)
  (do (active ?connector)
      (color ?connector ?hue)))


(define-update deactivate-connector! (?connector ?hue)
  (do (not (active ?connector))
      (not (color ?connector ?hue))))


(define-update activate-receiver! (?receiver)
  (do (active ?receiver)
      (doall (?g gate)
        (if (and (controls ?receiver ?g)
                 (active ?g))
          (not (active ?g))))))


(define-update deactivate-receiver! (?receiver)
  (do (not (active ?receiver))
      (doall (?g gate)
        (if (controls ?receiver ?g)
          (active ?g)))))
```

```
(define-update chain-activate! (?connector ?hue)
  (do (activate-connector! ?connector ?hue)
      (doall (?r receiver)
        (if (and (connecting ?connector ?r)
                 (not (active ?r))
                 (bind (color ?r $rhue))
                 (eql $rhue ?hue))
          (activate-receiver! ?r)))
      (doall (?c connector)
        (if (and (different ?c ?connector)
                 (connecting ?connector ?c)
                 (not (active ?c)))
          (chain-activate! ?c ?hue)))))


(define-update chain-deactivate! (?connector ?hue)
  (do (deactivate-connector! ?connector ?hue)
      (doall (?r receiver)
        (if (and (connecting ?connector ?r)
                 (not (exists (?c connector)
                         (and (connecting ?c ?r)
                              (bind (color ?c $hue1)
                              (eql $hue1 ?hue))))))
          (deactivate-receiver! ?r)))
      (doall (?c connector)
        (if (and (different ?c ?connector)
                 (connecting ?connector ?c))
          (do (not (connecting ?connector ?c))
              (chain-deactivate! ?c ?hue))))
      (doall (?t transmitter ?c connector)  ;reactivate
connectors with a transmitter source
        (if (and (not (eql ?c ?connector))
                 (connecting ?c ?t)
                 (not (active ?c))
                 (bind (color ?t $thue)))
          (chain-activate! ?c $thue)))))
```

```
;;;; ACTIONS ;;;;


(define-action connect-to-1-terminus
    2
  (?terminus terminus)
  (and (bind (holding me $cargo))
       (connector $cargo)
       (bind (loc me $area))
       (connectable? $area ?terminus))
  ($cargo fluent ?terminus terminus ($area $hue) fluent)
  (assert (not (holding me $cargo))
          (loc $cargo $area)
          (connecting $cargo ?terminus)
          (if (and (source? ?terminus)
                   (bind (color ?terminus $hue)))
            (activate-connector! $cargo $hue))))


(define-action connect-to-2-terminus
    3
  (combinations (?terminus1 ?terminus2) terminus)
  (and (bind (holding me $cargo))
       (connector $cargo)
       (bind (loc me $area))
       (connectable? $area ?terminus1)
       (connectable? $area ?terminus2))
  ($cargo fluent (?terminus1 ?terminus2) terminus $area
fluent)
  (assert (not (holding me $cargo))
          (loc $cargo $area)
          (connecting $cargo ?terminus1)
          (connecting $cargo ?terminus2)
          (bind (color ?terminus1 $hue1))
          (bind (color ?terminus2 $hue2))
          (if (or $hue1 $hue2)    ;at least one active
            (if (eql $hue1 $hue2)  ;both active and the
same color
              (setq $hue $hue1)
              (if (not (and $hue1 $hue2))  ;both are not
active (with different colors)
                (setq $hue (or $hue1 $hue2)))))
          (if $hue
                (chain-activate! $cargo $hue))))
```

```
(define-action connect-to-3-terminus
    4
  (combinations (?terminus1 ?terminus2 ?terminus3)
terminus)
  (and (bind (holding me $cargo))
       (connector $cargo)
       (bind (loc me $area))
       (connectable? $area ?terminus1)
       (connectable? $area ?terminus2)
       (connectable? $area ?terminus3))
  ($cargo fluent (?terminus1 ?terminus2 ?terminus3)
terminus $area fluent)
  (assert (not (holding me $cargo))
          (loc $cargo $area)
          (connecting $cargo ?terminus1)
          (connecting $cargo ?terminus2)
          (connecting $cargo ?terminus3)
          (bind (color ?terminus1 $hue1))
          (bind (color ?terminus2 $hue2))
          (bind (color ?terminus3 $hue3))
          (if (or $hue1 $hue2 $hue3)    ;at least one
active
            (if (eql* $hue1 $hue2 $hue3)  ;exactly three
active and the same color
              (setq $hue $hue1)
              (if (or (eql $hue1 $hue2)  ;exactly two
active and the same color
                      (eql $hue1 $hue3))
                (setq $hue $hue1)
                (if (eql $hue2 $hue3)
                  (setq $hue $hue2)
                  (if (not (and $hue1 $hue2 $hue3))  ;all
are not active (with different colors)
                    (setq $hue (or $hue1 $hue2
$hue3)))))))
          (if $hue
            (chain-activate! $cargo $hue))))
```

```
(define-action pickup-connector
    1
  (?connector connector)
  (and (free me)
       (bind (loc me $area))
       (loc ?connector $area))
  (?connector connector $area fluent)
  (assert (holding me ?connector)
          (not (loc ?connector $area))
          (if (bind (color ?connector $hue))
            (chain-deactivate! ?connector $hue))
          (doall (?t terminus)
            (if (connecting ?connector ?t)
              (not (connecting ?connector ?t))))))


(define-action drop-cargo
    1
  ()
  (and (bind (loc me $area))
       (bind (holding me $cargo)))
  ($cargo fluent $area fluent)
  (assert (not (holding me $cargo))
          (loc $cargo $area)))


(define-action move
    1
  (?area2 area)
  (and (bind (loc me $area1))
       (different $area1 ?area2)
       (passable? $area1 ?area2))
  ($area1 fluent ?area2 area)
  (assert (loc me ?area2)))


;;;; INITIALIZATION ;;;;


(define-init
  ;dynamic
  (loc me area5)
  (loc connector1 area5)
  (loc connector2 area7)
  (free me)
  (active gate1)
  (active gate2)
```

```
;static
(adjacent area1 area2)
(adjacent area2 area3)
(adjacent area3 area4)
(adjacent area4 area5)
(adjacent area6 area7)
(locale transmitter1 area4)
(locale transmitter2 area7)
(locale receiver1 area4)
(locale receiver2 area8)
(color transmitter1 blue)
(color transmitter2 red)
(color receiver1 blue)
(color receiver2 red)
(controls receiver1 gate1)
(controls receiver2 gate2)
(separates gate1 area4 area7)
(separates gate2 area7 area8)

;los is from an area to a fixed station
(los0 area2 transmitter1)
(los0 area3 transmitter1)
(los0 area3 receiver1)
(los0 area5 transmitter1)
(los0 area5 receiver1)
(los0 area5 receiver2)
(los0 area6 transmitter1)
(los0 area6 transmitter2)
(los0 area7 transmitter2)
(los0 area8 transmitter1)
(los1 area7 gate1 transmitter1)
(los1 area7 gate2 receiver2)
(los1 area8 gate2 transmitter2)
(los2 area3 gate1 gate2 receiver2)
(los2 area4 gate1 gate2 receiver2)

;visibility is from an area to an area
;potentially containing a movable target or terminus
(visible0 area1 area3)
(visible0 area1 area4)
(visible0 area1 area5)
(visible0 area2 area4)
(visible0 area2 area5)
(visible0 area2 area6)
(visible0 area3 area5)
(visible0 area3 area6)
(visible0 area3 area7)
```

```
  (visible0 area3 area8)
  (visible0 area4 area6)
  (visible0 area4 area8)
  (visible0 area5 area6)
  (visible0 area5 area8)
  (visible1 area1 gate1 area7)
  (visible1 area3 gate1 area7)
  (visible1 area2 gate1 area7)
  (visible1 area4 gate1 area7)
  (visible1 area4 gate1 area6)
  (visible1 area5 gate1 area7)
  (visible1 area6 gate2 area8)
  (visible1 area7 gate2 area8)


  (visible2 area2 gate1 gate2 area8)
  (visible2 area3 gate1 gate2 area8)
  (visible2 area4 gate1 gate2 area8)
)

;;;; INITIALIZATION ACTIONS ;;;;

;init-actions save listing systematic facts

 (define-init-action init-los0
   ;los exists to any station within its local area
    0
  (?station station (?area1 ?area2) area)
  (or (locale ?station ?area1)              ;for fixtures
      (separates ?station ?area1 ?area2))   ;for gates
  ()
  (assert (los0 ?area1 ?station)))


 (define-init-action init-visible0-locally
   ;any object is visible from its own local area
    0
  (?area area)
  (always-true)
  ()
  (assert (visible0 ?area ?area)))
```

```
 (define-init-action init-visible0-via-adjacency
   ;any object is visible from an adjacent area
    0
  ((?area1 ?area2) area)
  (adjacent ?area1 ?area2)
  ()
  (assert (visible0 ?area1 ?area2)))


 (define-init-action init-visible1-thru-divider
   ;any object is visible thru a divider
    0
  (?divider divider (?area1 ?area2) area)
  (separates ?divider ?area1 ?area2)
  ()
  (assert (visible1 ?area1 ?divider ?area2)))



;;;; GOAL ;;;;


(define-goal  ;always put this last
  (loc me area8))
```

Smallspace Problem Solution:

```
Proceed with testing problem-smallspace.lisp?: (y or n) y

* (solve)

working...

New path to goal found at depth = 19

New path to goal found at depth = 17


In problem SMALLSPACE, performed GRAPH search for MIN-
TIME solution.

Search process completed normally.

Examined only worthwhile states up to the depth cutoff.

Depth cutoff = 19
```

```
Maximum depth explored = 19

Total states processed = 28,917

Repeated states = 6,681, ie, 23.1 percent

Program cycles (state expansions) = 9,856

Average branching factor = 14.7

Start state:
((ACTIVE GATE1) (ACTIVE GATE2) (COLOR RECEIVER1 BLUE)
(COLOR RECEIVER2 RED) (COLOR TRANSMITTER1 BLUE) (COLOR
TRANSMITTER2 RED) (FREE ME)
 (LOC CONNECTOR1 AREA5) (LOC CONNECTOR2 AREA7) (LOC ME
AREA5))

Goal:
(LOC ME AREA8)



Total solution paths recorded = 2, of which 2 is/are
unique solution paths
Check *solutions* and *unique-solutions* for solution
records.

Duration of a minimum time solution = 26.0

A minimum time solution path from start state to goal
state:
(1.0 (PICKUP-CONNECTOR CONNECTOR1 AREA5))
(4.0 (CONNECT-TO-2-TERMINUS CONNECTOR1 RECEIVER1
TRANSMITTER1 AREA5))
(5.0 (MOVE AREA5 AREA4))
(6.0 (MOVE AREA4 AREA7))
(7.0 (PICKUP-CONNECTOR CONNECTOR2 AREA7))
(10.0 (CONNECT-TO-2-TERMINUS CONNECTOR2 CONNECTOR1
TRANSMITTER1 AREA7))
(11.0 (PICKUP-CONNECTOR CONNECTOR2 AREA7))
(14.0 (CONNECT-TO-2-TERMINUS CONNECTOR2 CONNECTOR1
TRANSMITTER2 AREA7))
(15.0 (PICKUP-CONNECTOR CONNECTOR2 AREA7))
(16.0 (MOVE AREA7 AREA4))
(18.0 (CONNECT-TO-1-TERMINUS CONNECTOR2 RECEIVER1 AREA4
NIL))
(19.0 (MOVE AREA4 AREA5))
```

```
(20.0 (PICKUP-CONNECTOR CONNECTOR1 AREA5))
(23.0 (CONNECT-TO-2-TERMINUS CONNECTOR1 CONNECTOR2
RECEIVER2 AREA5))
(24.0 (MOVE AREA5 AREA4))
(25.0 (MOVE AREA4 AREA7))
(26.0 (MOVE AREA7 AREA8))

Final state:
((ACTIVE CONNECTOR1) (ACTIVE CONNECTOR2) (ACTIVE
RECEIVER1) (ACTIVE RECEIVER2) (COLOR CONNECTOR1 RED)
(COLOR CONNECTOR2 RED)
 (COLOR RECEIVER1 BLUE) (COLOR RECEIVER2 RED) (COLOR
TRANSMITTER1 BLUE) (COLOR TRANSMITTER2 RED) (CONNECTING
CONNECTOR1 CONNECTOR2)
 (CONNECTING CONNECTOR1 RECEIVER2) (CONNECTING CONNECTOR2
RECEIVER1) (FREE ME) (LOC CONNECTOR1 AREA5) (LOC
CONNECTOR2 AREA4) (LOC ME AREA8))

Evaluation took:
  0.091 seconds of real time
  0.062500 seconds of total run time (0.062500 user,
0.000000 system)
  68.13% CPU
  273,495,088 processor cycles
  66,100,240 bytes consed
```

7. Capt John's Journey Problem

The following is a logic problem from braingle.com called Captain John's Journey (Part 1), submitted by cdrock. It illustrates how to solve a Constraint Satisfaction Problem (CSP) with the Wouldwork planner, since a CSP is not normally regarded as a planning problem. The basic approach is to write a single action specification that progressively generates values for each constrained variable, and then checks those values against a goal detailing the given constraints. The problem is solved when the set of variable values satisfies all of the goal constraints. Note that the 4-queens problem presented earlier is a simple example of a CSP.

Captain John is the captain of a pirate ship called the Wasp. He just heard about a lost treasure on a far-away island. He needs to get his two crew mates, and lead them to a ship, but there are guards around and he needs to do this without passing them, or they will throw him in the brig. Can you help him get his two crew mates to the ship without getting sent to the brig?

The positions of everything are in a 3-by-3 grid (1 John, 1 ship, 2 crew mates, 2 guards, and 3 grass areas). John may only move 1 space at a time, either vertically, horizontally, or diagonally. He can only go to each space once.

But before he can figure out the right way to go, he must figure out where everything is to start with. This is what he knew:

1. The Wasp is not in the same row or column as John.
2. John is not in the same row or column as either guard.
3. Neither guard is in the third column.
4. Both guards are vertically next to grass.
5. The ship is in the same row as one guard, and the same column as the other guard.
6. One of the grass spaces is diagonally next to both crew mates.

7. One of the grass spaces is in the 2nd column, in the first row.
8. The two guards are not in the same row or column.

These statements are enough for Capt John to figure out where all nine tokens are initially located on the 3-by-3 grid (part 1).

Capt John Problem Specification

```
;;; Filename: problem-captjohn.lisp


;;; Brain Teaser logic problem,
;;; Capt John's Journey (part 1)


(in-package :ww)

(ww-set *problem* captjohn)

(ww-set *solution-type* first)

(ww-set *tree-or-graph* tree)


(define-types
    captain (john)
    ship    (wasp)
    crew    (crew1 crew2)
    guard   (guard1 guard2)
    grass   (grass1 grass2 grass3)
    object  (either captain ship crew guard grass)
    row     (1 2 3)
    column  (1 2 3))


(define-dynamic-relations
    (loc object $row $column)
    (next-row $row)
    (next-col $column))


(define-query already-placed? (?object)
  (bind (loc ?object $r $c)))
```

```
(define-query in-same-row? (?object1 ?object2)
  (and (bind (loc ?object1 $r1 $c1))
       (bind (loc ?object2 $r2 $c2))
       (= $r1 $r2)))


(define-query in-same-col? (?object1 ?object2)
  (and (bind (loc ?object1 $r1 $c1))
       (bind (loc ?object2 $r2 $c2))
       (= $c1 $c2)))


(define-query in-col? (?object ?column)
  (and (bind (loc ?object $r $c))
       (= $c ?column)))


(define-query vert-next-to? (?object1 ?object2)
  (and (bind (loc ?object1 $r1 $c1))
       (bind (loc ?object2 $r2 $c2))
       (and (= $c1 $c2)
       (or (= $r1 (1+ $r2))
           (= $r1 (1- $r2))))))


(define-query diag-next-to? (?object1 ?object2)
  (and (bind (loc ?object1 $r1 $c1))
       (bind (loc ?object2 $r2 $c2))
       (or (and (= (1+ $r1) $r2)
                (= (1+ $c1) $c2))
           (and (= (1+ $r1) $r2)
                (= (1- $c1) $c2))
           (and (= (1- $r1) $r2)
                (= (1+ $c1) $c2))
           (and (= (1- $r1) $r2)
                (= (1- $c1) $c2)))))
```

```
(define-action put
    1
  (?object object)
  (and (not (already-placed? ?object))
       (bind (next-row $row))
       (bind (next-col $col)))
  (?object object ($row $col) fluent)
  (assert (loc ?object $row $col)
          (if (= $col 3)
            (do (next-col 1)
                (next-row (1+ $row)))
            (next-col (1+ $col)))))


(define-init
  (next-row 1)
  (next-col 1))


(define-goal
  (and (next-row 4)  ;only check if on last row
       (and (not (in-same-row? wasp john))
            (not (in-same-col? wasp john)))
       (forall (?guard guard)
         (and (not (in-same-row? john ?guard))
              (not (in-same-col? john ?guard))))
       (forall (?guard guard)
         (not (in-col? ?guard 3)))
       (forall (?guard guard)
         (exists (?grass grass)
           (vert-next-to? ?guard ?grass)))
       (exists ((?guard1 ?guard2) guard)
         (and (in-same-row? wasp ?guard1)
              (in-same-col? wasp ?guard2)))
       (exists (?grass grass)
         (forall (?crew crew)
           (diag-next-to? ?grass ?crew)))
       (exists (?grass grass)
         (loc ?grass 1 2))
       (exists ((?guard1 ?guard2) guard)
         (and (not (in-same-row? ?guard1 ?guard2))
              (not (in-same-col? ?guard1 ?guard2))))))
```

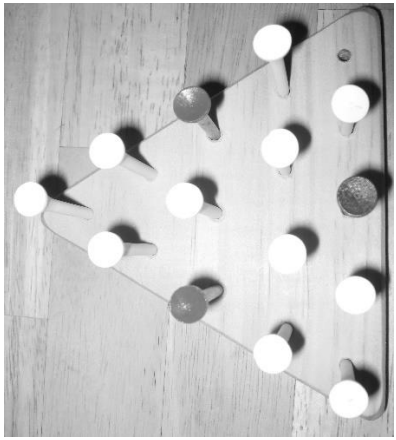
Capt John Problem Solution

```
Proceed with testing problem-captjohn.lisp?: (y or n) y

working...

total states processed so far = 100,000
average branching factor = 25.818073

total states processed so far = 200,000
average branching factor = 26.914892

total states processed so far = 300,000
average branching factor = 27.563103

total states processed so far = 400,000
average branching factor = 28.014376

total states processed so far = 500,000
average branching factor = 28.359274

New path to goal found at depth = 9


In problem CAPTJOHN, performed TREE search for FIRST
solution.

Search ended with first solution found.

Depth cutoff = 0

Maximum depth explored = 10

Total states processed = 512,409

Program cycles (state expansions) = 323,896

Average branching factor = 28.392736

Start state:
((NEXT-COL 1) (NEXT-ROW 1))
```

```
Goal:
(AND (NEXT-ROW 4)
     (AND (NOT (IN-SAME-ROW? WASP JOHN))
          (NOT (IN-SAME-COL? WASP JOHN)))
     (FORALL (?GUARD GUARD)
       (AND (NOT (IN-SAME-ROW? JOHN ?GUARD))
            (NOT (IN-SAME-COL? JOHN ?GUARD))))
     (FORALL (?GUARD GUARD)
       (NOT (IN-COL? ?GUARD 3)))
     (FORALL (?GUARD GUARD)
       (EXISTS (?GRASS GRASS)
         (VERT-NEXT-TO? ?GUARD ?GRASS)))
     (EXISTS ((?GUARD1 ?GUARD2) GUARD)
       (AND (IN-SAME-ROW? WASP ?GUARD1)
            (IN-SAME-COL? WASP ?GUARD2)))
     (EXISTS (?GRASS GRASS)
       (FORALL (?CREW CREW)
         (DIAG-NEXT-TO? ?GRASS ?CREW)))
     (EXISTS (?GRASS GRASS)
       (LOC ?GRASS 1 2))
     (EXISTS ((?GUARD1 ?GUARD2) GUARD)
       (AND (NOT (IN-SAME-ROW? ?GUARD1 ?GUARD2))
            (NOT (IN-SAME-COL? ?GUARD1 ?GUARD2)))))



Total solutions recorded = 1
(Check *solutions* for list of all solution records.)

Number of steps in first solution found: = 9

Duration of first solution found = 9.0

Solution path of first solution found from start state to
goal state:
(1.0 (PUT GUARD1 1 1))
(2.0 (PUT GRASS1 1 2))
(3.0 (PUT CREW1 1 3))
(4.0 (PUT GRASS2 2 1))
(5.0 (PUT GRASS3 2 2))
(6.0 (PUT JOHN 2 3))
(7.0 (PUT WASP 3 1))
(8.0 (PUT GUARD2 3 2))
(9.0 (PUT CREW2 3 3))
```

```
Final state:
((LOC CREW1 1 3) (LOC CREW2 3 3) (LOC GRASS1 1 2)
 (LOC GRASS2 2 1) (LOC GRASS3 2 2) (LOC GUARD1 1 1)
 (LOC GUARD2 3 2) (LOC JOHN 2 3) (LOC WASP 3 1)
 (NEXT-COL 1) (NEXT-ROW 4))

Evaluation took:
  2.245 seconds of real time
  2.218750 seconds of total run time (2.218750 user,
0.000000 system)
  [ Run times consist of 0.031 seconds GC time, and 2.188
seconds non-GC time. ]
  98.84% CPU
  8,081,366,254 processor cycles
  1,341,335,376 bytes consed
```

8. Triangle Peg Puzzle



The Triangle Peg Puzzle (aka Cracker Barrel Puzzle, Conqueror Puzzle) consists of a triangular peg board with 15 holes and 14 pegs, initially filling all holes except one. The objective is to jump over a peg on each move, as in checkers, continuing to jump with different pegs on each move, until there is only one peg left. Any solution therefore will require 13 jumps total.

This puzzle illustrates how Common Lisp code can be added to a problem specification, in this case to setup the initial board configuration, when using a standard planning initialization action would be awkward and inefficient.

Triangle Peg Problem Specification

```
;;;; Filename: problem-triangle-xy.lisp


;;; Basic problem specification for triangle peg problem
with peg-count.

;;; The peg board positions have coordinates measured
;;; from the triangle's right diagonal (/) and left
diagonal (\)
;;; side lengths from 1 to *N* with pegs in all positions
except 11.
```

```
;;; Jumps are by / up or down, \ up or down, - (horiz)
right or left.
;;;          11
;;;        12  21
;;;      13  22  31
;;;    14  23  32  41
;;; 15  24  33  42  51


(in-package :ww)  ;required

(ww-set *problem* triangle-xy)

(ww-set *tree-or-graph* tree)


(defparameter *N* 5)  ;the number of pegs on a side

(defparameter *holes* '((1 1)))  ;coordinates of the
initial holes


(define-types
  peg (compute (loop for i from 1  ;peg1, peg2, ...
                     to (- (/ (* *N* (1+ *N*)) 2)
                           (length *holes*))
                  collect (intern (format nil
                                          "PEG~D" i))))
  row (compute (loop for i from 1 to *N*
                  collect i))
  col (compute (loop for j from 1 to *N*
                  collect j))
  current-peg (get-current-pegs?))


(define-dynamic-relations
    (loc peg $row $col)       ;location of a peg
    (contents row col $peg)   ;peg contents at a location
    (remaining-pegs $list)    ;list of remaining pegs
    (peg-count $integer))     ;pegs remaining on the board


(define-query get-current-pegs? ()
  (do (bind (remaining-pegs $pegs))
      $pegs))
```

```
(define-action jump-left-down  ;jump downward in the /
                                diagonal direction
    1
  (?peg current-peg)
  (and (bind (loc ?peg $r $c))
       (<= (+ $r $c) (- *N* 1))
       (setq $c+1 (1+ $c))
       (bind (contents $r $c+1 $adj-peg))
       (setq $c+2 (+ $c 2))
       (not (bind (contents $r $c+2 $any-peg)))
       (bind (peg-count $peg-count))
       (bind (remaining-pegs $pegs)))
  (($r $c) fluent)
  (assert (not (contents $r $c ?peg))  ;from
          (loc ?peg $r $c+2)           ;to
          (contents $r $c+2 ?peg)      ;update to
          (not (loc $adj-peg $r $c+1))    ;remove adj peg
          (not (contents $r $c+1 $adj-peg)) ;remove adj
          (peg-count (1- $peg-count))
          (remaining-pegs (remove $adj-peg $pegs))))


(define-action jump-right-up  ;jump upward in the /
                               diagonal direction
    1
  (?peg current-peg)
  (and (bind (loc ?peg $r $c))
       (>= $c 3)
       (setq $c-1 (1- $c))
       (bind (contents $r $c-1 $adj-peg))
       (setq $c-2 (- $c 2))
       (not (bind (contents $r $c-2 $any-peg)))
       (bind (peg-count $peg-count))
       (bind (remaining-pegs $pegs)))
  (($r $c) fluent)
  (assert (not (contents $r $c ?peg))
          (loc ?peg $r $c-2)
          (contents $r $c-2 ?peg)
          (not (loc $adj-peg $r $c-1))
          (not (contents $r $c-1 $adj-peg))
          (peg-count (1- $peg-count))
          (remaining-pegs (remove $adj-peg $pegs))))
```

```
(define-action jump-right-down  ;jump downward in the \
                                 diagonal direction
    1
  (?peg current-peg)
  (and (bind (loc ?peg $r $c))
       (<= (+ $r $c) (- *N* 1))
       (setq $r+1 (+ $r 1))
       (bind (contents $r+1 $c $adj-peg))
       (setq $r+2 (+ $r 2))
       (not (bind (contents $r+2 $c $any-peg)))
       (bind (peg-count $peg-count))
       (bind (remaining-pegs $pegs)))
  (($r $c) fluent)
  (assert (not (contents $r $c ?peg))
          (loc ?peg $r+2 $c)
          (contents $r+2 $c ?peg)
          (not (loc $adj-peg $r+1 $c))
          (not (contents $r+1 $c $adj-peg))
          (peg-count (1- $peg-count))
          (remaining-pegs (remove $adj-peg $pegs))))


(define-action jump-left-up  ;jump upward in the \
                              diagonal direction
    1
  (?peg current-peg)
  (and (bind (loc ?peg $r $c))
       (>= $r 3)
       (setq $r-1 (- $r 1))
       (bind (contents $r-1 $c $adj-peg))
       (setq $r-2 (- $r 2))
       (not (bind (contents $r-2 $c $any-peg)))
       (bind (peg-count $peg-count))
       (bind (remaining-pegs $pegs)))
  (($r $c) fluent)
  (assert (not (contents $r $c ?peg))
          (loc ?peg $r-2 $c)
          (contents $r-2 $c ?peg)
          (not (loc $adj-peg $r-1 $c))
          (not (contents $r-1 $c $adj-peg))
          (peg-count (1- $peg-count))
          (remaining-pegs (remove $adj-peg $pegs))))
```

```
(define-action jump-right-horiz  ;jump rightward in the
                                  horizontal direction
    1
  (?peg current-peg)
  (and (bind (loc ?peg $r $c))
       (>= $c 3)
       (setq $r+1 (+ $r 1))
       (setq $c-1 (- $c 1))
       (bind (contents $r+1 $c-1 $adj-peg))
       (setq $r+2 (+ $r 2))
       (setq $c-2 (- $c 2))
       (not (bind (contents $r+2 $c-2 $any-peg)))
       (bind (peg-count $peg-count))
       (bind (remaining-pegs $pegs)))
  (($r $c) fluent)
  (assert (not (contents $r $c ?peg))
          (loc ?peg $r+2 $c-2)
          (contents $r+2 $c-2 ?peg)
          (not (loc $adj-peg $r+1 $c-1))
          (not (contents $r+1 $c-1 $adj-peg))
          (peg-count (1- $peg-count))
          (remaining-pegs (remove $adj-peg $pegs))))
```

```
(define-action jump-left-horiz  ;jump leftward in the
                                 horizontal direction
    1
  (?peg current-peg)
  (and (bind (loc ?peg $r $c))
       (>= $r 3)
       (setq $r-1 (- $r 1))
       (setq $c+1 (+ $c 1))
       (bind (contents $r-1 $c+1 $adj-peg))
       (setq $r-2 (- $r 2))
       (setq $c+2 (+ $c 2))
       (not (bind (contents $r-2 $c+2 $any-peg)))
       (bind (peg-count $peg-count))
       (bind (remaining-pegs $pegs)))
  (($r $c) fluent)
  (assert (not (contents $r $c ?peg))
          (loc ?peg $r-2 $c+2)
          (contents $r-2 $c+2 ?peg)
          (not (loc $adj-peg $r-1 $c+1))
          (not (contents $r-1 $c+1 $adj-peg))
          (peg-count (1- $peg-count))
          (remaining-pegs (remove $adj-peg $pegs))))


(progn (format t "~&Initializing database...~%")
 (loop with pegs = (gethash 'peg *types*)
   ;*db* is the name of the initial database
   ;update is the function that asserts a proposition
   ;into the database
   initially (update *db* `(peg-count ,(length pegs)))
             (update *db* `(remaining-pegs ,pegs))
   for row from 1 to *N*
    do (loop for col from 1 to (- (1+ *N*) row)
         unless (member (list row col) *holes*
                        :test #'equal)
         do (let ((peg (pop pegs)))
              (update *db* `(loc ,peg ,row ,col))
              (update *db* `(contents ,row ,col ,peg))))))


(define-goal  ;only one peg left
  (peg-count 1))
```

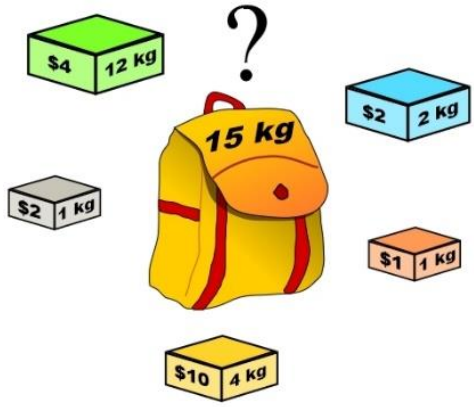
Triangle Peg Problem Solution:

```
Proceed with testing problem-triangle.lisp?: (y or n) y
```

```
working...

New path to goal found at depth = 13


In problem TRIANGLE-XY, performed TREE search for FIRST
solution.

Search ended with first solution found.

Depth cutoff = 0

Maximum depth explored = 13

Total states processed = 136

Program cycles (state expansions) = 66

Average branching factor = 9.317418


Start state:
((CONTENTS 1 2 PEG1) (CONTENTS 1 3 PEG2)
 (CONTENTS 1 4 PEG3) (CONTENTS 1 5 PEG4)
 (CONTENTS 2 1 PEG5) (CONTENTS 2 2 PEG6)
 (CONTENTS 2 3 PEG7) (CONTENTS 2 4 PEG8)
 (CONTENTS 3 1 PEG9) (CONTENTS 3 2 PEG10)
 (CONTENTS 3 3 PEG11) (CONTENTS 4 1 PEG12)
 (CONTENTS 4 2 PEG13) (CONTENTS 5 1 PEG14) (LOC PEG1 1 2)
 (LOC PEG10 3 2) (LOC PEG11 3 3) (LOC PEG12 4 1)
 (LOC PEG13 4 2) (LOC PEG14 5 1) (LOC PEG2 1 3)
 (LOC PEG3 1 4) (LOC PEG4 1 5) (LOC PEG5 2 1)
 (LOC PEG6 2 2) (LOC PEG7 2 3) (LOC PEG8 2 4)
 (LOC PEG9 3 1) (PEG-COUNT 14)
 (REMAINING-PEGS (PEG1 PEG2 PEG3 PEG4 PEG5 PEG6 PEG7 PEG8
                  PEG9 PEG10 PEG11 PEG12 PEG13 PEG14)))

Goal:
(PEG-COUNT 1)



Total solutions recorded = 1
(Check *solutions* for list of all solution records.)

Number of steps in first solution found: = 13
```

```
Duration of first solution found = 13.0

Solution path of first solution found from start state to
goal state:
(1.0 (JUMP-RIGHT-UP 1 3))
(2.0 (JUMP-RIGHT-UP 1 5))
(3.0 (JUMP-LEFT-UP 3 2))
(4.0 (JUMP-LEFT-DOWN 1 2))
(5.0 (JUMP-RIGHT-UP 2 4))
(6.0 (JUMP-LEFT-DOWN 2 1))
(7.0 (JUMP-LEFT-UP 4 1))
(8.0 (JUMP-RIGHT-DOWN 1 1))
(9.0 (JUMP-RIGHT-HORIZ 1 4))
(10.0 (JUMP-LEFT-HORIZ 4 2))
(11.0 (JUMP-LEFT-DOWN 3 1))
(12.0 (JUMP-RIGHT-HORIZ 2 4))
(13.0 (JUMP-LEFT-HORIZ 5 1))


Final state:
((CONTENTS 3 3 PEG14) (LOC PEG14 3 3) (PEG-COUNT 1)
 (REMAINING-PEGS (PEG14)))

Evaluation took:
  0.010 seconds of real time
  0.000000 seconds of total run time (0.000000 user,
0.000000 system)
  0.00% CPU
  37,162,582 processor cycles
  1,505,440 bytes consed
```

9. Knapsack Problem



Given a knapsack which can carry a maximum weight W, and a number of items with weights w_i and values v_i, how can you pack the knapsack so as to fit in items with the maximum total value V, without exceeding the knapsack's capacity W?

This is an optimization problem, in this case find a goal (ie, a knapsack filling) that has the maximum total value. You must specify what is to be optimized, namely the value in the knapsack ($objective-value), computed in the *put* rule below.

The query function *get-best-relaxed-value?* computes a maximum possible value to fill the remaining capacity in the knapsack from any given state, allowing a fractional object as the last object (with its fractional weight and fractional value). Since the items have been initially ordered by their value/weight ratios, the last fractional value added is the best total value that can be achieved in that state. This relaxed value is used by the planner to prune any subsequent states whose value is worse than the best value so far.

Knapsack Problem Specification

```
;;;; Filename: problem-knap19.lisp

;;; Problem specification for a 19-item knapsack problem.


(in-package :ww)  ;required


(ww-set *problem* knap19)

(ww-set *solution-type* max-value)


(defstruct item  ;an item template
  (name nil :type symbol)  ;item name--eg, ITEM14
  (value 0 :type fixnum)  ;value of that item
  (weight 0 :type fixnum)  ;weight of that item
  (value/weight 0 :type ratio))  ;the ratio of the item's
                                   value/weight

(defparameter *items* nil)  ;the list of available items
(defparameter *num-items* 0)  ;total number of items
(defparameter *max-knapsack-weight* 0)  ;knapsack max
                                          weight

(defun readin (knapsack-data-file)
  ;Read in knapsack data from a file.
  (with-open-file (ifile knapsack-data-file
    :direction :input :if-does-not-exist nil)
    (when (not (streamp ifile))
      (error "File does not exist!"))
    (setq *num-items* (read ifile))
    (setq *max-knapsack-weight* (read ifile))
    (dotimes (i *num-items*)
      (let ((value (read ifile))
            (weight (read ifile)))
        (push (make-item
                :name (intern (concatenate 'string "ITEM"
                                (write-to-string (1+ i))))
                          :value value
                          :weight weight
                          :value/weight (/ value weight))
              *items*)))
    (setq *items* (sort *items* #'>
                        :key #'item-value/weight))))
```

```
(setq *default-pathname-defaults*  ;point to where
                                knapsack data file is stored
   #P"D:\\Users Data\\Dave\\SW Library\\AI\\
          Planning\\Wouldwork Planner\\")

(readin "data-knap19.lisp")  ;name of the knapsack data
                              file


(define-types
  knapsack (knapsack1)
  item (compute (loop for item in *items*
                   collect (item-name item))))


(define-dynamic-relations
  (in item knapsack)
  (load knapsack $fixnum)
  (worth knapsack $fixnum))


(define-static-relations
  (value item $fixnum)
  (weight item $fixnum))
```

```
(define-query get-best-relaxed-value? ()  ;for a succ
                                           state
  (do (bind (load knapsack1 $load))
      (bind (worth knapsack1 $worth))
      (doall (?item item)  ;items previously ordered by
                           value/weight ratio
        (and (not (in ?item knapsack1))
             (bind (weight ?item $weight))
             (bind (value ?item $value))
             (if (< (+ $load $weight)
                    *max-knapsack-weight*)
               (setq $load (+ $load $weight)
                     $worth (+ $worth $value))
               (do (setq $fraction
                     (/ (- *max-knapsack-weight* $load)
                        $weight))
                   (setq $worth
                     (+ $worth (* $fraction $value)))))))
      $worth))


(define-action put
    1
  (?knapsack knapsack ?item item)
  (and (not (in ?item ?knapsack))
       (bind (weight ?item $item-weight))
       (bind (load ?knapsack $knapsack-load))
       (setq $new-knapsack-load
         (+ $knapsack-load $item-weight))
       (<= $new-knapsack-load *max-knapsack-weight*)
       (bind (worth ?knapsack $knapsack-worth))
       (bind (value ?item $item-value))
       (setq $new-knapsack-worth
         (+ $knapsack-worth $item-value))
       (setq $objective-value $new-knapsack-worth))
  (?item item ?knapsack knapsack)
  (assert (in ?item ?knapsack)
          (load ?knapsack $new-knapsack-load)
          (worth ?knapsack $new-knapsack-worth)))


(define-init
  `(capacity knapsack1 ,*max-knapsack-weight*)
  (load knapsack1 0)
  (worth knapsack1 0))
```

```
(define-init-action init-item-weights&values
    0
  (?item item)
  (setq $item-structure (find ?item *items*
                              :key #'item-name))
  ()
  (assert (weight ?item (item-weight $item-structure))
          (value ?item (item-value $item-structure))))


(define-goal  ;can't put any further item into the
               knapsack
  (not (exists (?knapsack knapsack ?item item)
         (and (not (in ?item ?knapsack))
              (bind (weight ?item $item-weight))
              (bind (load ?knapsack $knapsack-load))
              (<= (+ $knapsack-load $item-weight)
                  *max-knapsack-weight*)
              (bind (worth ?knapsack
                           $knapsack-worth))))))
```

Knapsack Problem Solution:

```
Proceed with testing problem-knap19.lisp?: (y or n) y

working...

New path to goal found at depth = 4
Objective value = 12019

New path to goal found at depth = 4
Objective value = 12060

New path to goal found at depth = 4
Objective value = 12066

New path to goal found at depth = 5
Objective value = 12085

New path to goal found at depth = 5
Objective value = 12187

New path to goal found at depth = 5
Objective value = 12197

New path to goal found at depth = 5
Objective value = 12248
```

```
total states processed so far = 100,000
ht count: 13,859    ht size: 100,000
average branching factor = 24.570202

total states processed so far = 200,000
ht count: 24,773    ht size: 100,000
average branching factor = 24.997911


In problem KNAP19, performed GRAPH search for MAX-VALUE
solution.

Search process completed normally.

Examined only worthwhile states up to the depth cutoff.

Depth cutoff = 0

Maximum depth explored = 11

Total states processed = 263,715

Repeated states = 233,503, ie, 88.54369 percent

Program cycles (state expansions) = 40,539

Average branching factor = 25.144413

Start state:
((LOAD KNAPSACK1 0) (WORTH KNAPSACK1 0))

Goal:
(NOT
 (EXISTS (?KNAPSACK KNAPSACK ?ITEM ITEM)
  (AND (NOT (IN ?ITEM ?KNAPSACK))
       (BIND (WEIGHT ?ITEM $ITEM-WEIGHT))
       (BIND (LOAD ?KNAPSACK $KNAPSACK-LOAD))
       (<= (+ $KNAPSACK-LOAD $ITEM-WEIGHT)
           *MAX-KNAPSACK-WEIGHT*)
       (BIND (WORTH ?KNAPSACK $KNAPSACK-WORTH)))))



Total solutions recorded = 7
(Check *solutions* for list of all solution records.)
```

```
Value of a maximum value solution = 12,248


A maximum value solution path from start state to goal
state:
(1.0 (PUT ITEM14 KNAPSACK1))
(2.0 (PUT ITEM3 KNAPSACK1))
(3.0 (PUT ITEM8 KNAPSACK1))
(4.0 (PUT ITEM13 KNAPSACK1))
(5.0 (PUT ITEM6 KNAPSACK1))


Final state:
((IN ITEM13 KNAPSACK1) (IN ITEM14 KNAPSACK1)
 (IN ITEM3 KNAPSACK1) (IN ITEM6 KNAPSACK1)
 (IN ITEM8 KNAPSACK1) (LOAD KNAPSACK1 30996)
 (WORTH KNAPSACK1 12248))

Evaluation took:
  39.871 seconds of real time
  39.796875 seconds of total run time (39.671875 user,
0.125000 system)
  [ Run times consist of 0.173 seconds GC time, and
39.624 seconds non-GC time. ]
  99.81% CPU
  143,534,420,434 processor cycles
  752,642,720 bytes consed
```