CLASSICAL PLANNING WITH THE WOULDWORK PLANNER

USER MANUAL

David Alan Brown

ISBN-13: 978-1546752059

ISBN-10: 1546752056

Table of Contents(9)

CLASSICAL PLANNING WITH THE WOULDWORK PLANNER (9)

Introduction

This is a user manual for the Wouldwork Planner (a.k.a. The I'd Be Pleased If You Would Work Planner). 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 specifed 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 problem-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 Forward, LPG, MIPS-XXL, SATPLAN, SGPLAN, Metric-FF Planner, Optop, 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 by one individual, 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. 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
- 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 Lisp code in action rules, constraints, and functions
- User control over search depth
- Identification of shortest length and shortest time plans found
- Output 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 rigorous testing. Run-time error checking is piecemeal, and latent software bugs surely remain. Users may send bug reports, suggestions, or other useful comments to Dave Brown at davypough@gmail.com.

Quickstart for MS Windows

1) Go to https://github.com/davypough/wouldwork and click on the releases tab. Download blocks.exe.
2) Open a Windows command prompt at the download directory, and enter blocks.exe. (Alternately, open Windows Powershell at the download directory, and enter .\blocks.exe.) This runs the planner on the blocks world problem specified in the file problem-blocks.lisp in the Wouldwork\src\ directory.
3) Review the planning results in the command window.

If you have a 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. The ASDF registry should be set in your init file to point to the directory where the Wouldwork source files are located, for example:

```
(setq asdf:*central-registry*
 (list #P"D:\\Users Data\\Dave\\Wouldwork\\src\\"))
```

(See the file 'wouldwork planner.asd' in the Wouldwork\src\ directory for what Wouldwork loads. Also note that SBCL loads with 1000MB = 1GB of memory by default. If a planning problem exits with an out-of-memory condition, you will need to increase the default by loading SBCL at the command prompt with additional memory, for example:

```
> sbcl.exe --dynamic-space-size 2000
```

which doubles the default.)

First, create a new problem specification as described below, and name the file problem.lisp, since the planner always looks for a problem specification file with this name. Then to recompile and load the planner from the source files, at the Lisp prompt execute (asdf:load-system "wouldwork planner" :force t), which will reinstall everything based on the instructions in the file 'wouldwork planner.asd'. Normally, planner parameters (eg, depth cutoff) are set in the problem specification, or simply left at their default values. But if you wish to override the default parameters, you can add them to the problem specification file. Alternately, you can reset them at the Lisp prompt, after compiling and loading, by first switching to the planner package with (in-package :ww). Finally, executing (bnb::solve) 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 more 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 above 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))
```

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 primary 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 primary 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 (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 (?block) on a support (?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. 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)
            (on ?b ?block)))
     (on ?block ?block-support)
     (not (exists (?b block)
            (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 support 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 target support place, labeled ?support, itself has a clear top. It is worth noting that the second condition can also be eliminated if a fluent is used in place of the variable ?block-support, resulting in a more natural expression of the precondition as simply (and (cleartop! ?block) (cleartop! ?support)).

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.

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), fourth is the precondition, fifth is the free parameters appearing in the effect, and sixth is the effect:

```
(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 (?block-support ?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 combinations 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 (?block ?block-support ?support) combination triples ((A B C) (A C B) (A B T) (A T B) (A T C) (A C T) (B A C) (B C A) (B A T) (B T A) (B T C) (B C T) (C B A) (C A B) (C B T) (C T B) (C T A) (C A T)), each of which are tested against the action preconditions. 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. Each instantiation represents a possible next step in the planner's path to the goal. 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. 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 somewhat for each rule, but at the 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 parameters, thereby restricting the effect parameter instantiations. Thus, the effect parameters become independently instantiated for all effect combinations, but limited to those successfully inherited from the precondition. The body of the effect then runs for each effect instantiation, registering the specific effects of the action for that instantiation in the database. 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 in the action rules—for example, (not (on B T)) expresses the condition that B is not on the table.

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)))
```

This completes the basic specification for the blocks world problem, except for a final operational statement telling 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 can 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. 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 unspecified, and stipulate a graph (rather than tree) search. A graph search avoids reanalyzing previously visited states, thereby potentially shortening the search, but incurs some additional overhead that a tree search avoids.

Program Control Settings

There are a number of program settings (like depth cutoff mentioned in the prior paragraph) that the user has control over. All of these settings have default values, but the user can change them in the problem specification, if needed. Simply add a statement like (setq <setting> <value>). The setting names and their initial default values (in the package :ww) are as follows:

depth-cutoff 0
Specifies the max search depth; negative or 0 means no cutoff, which may potentially search forever, especially for a tree search.

tree-or-graph '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); graph search is the default, but if there are no repeated states, tree search will be significantly faster

first-solution-sufficient nil
Specifies whether only one solution, the first found, is required, ending the search; setting to t (true) will stop with the first solution, while nil (false) will continue searching for the shortest solution

debug 0
Specifies the current amount of debugging information to be displayed during the planning process (discussed later at the end of Part 2)

progress-reporting-interval 100000
Specifies how often to report progress to the terminal during search; reports after each multiple n of states examined; useful for long searches

max-states 1000000
Specifies the estimated total maximum number of *unique* states to be explored during search; if this number is exceeded, hash table resizing may slow search significantly

Program Output

*** (bnb::solve)**

working ...

New path to goal found at depth = 2

Graph search process completed normally,
examining every state up to the depth cutoff.

Depth cutoff = 0

Maximum depth explored = 4

Total states processed = 60

Unique states encountered = 25

Program cycles (state expansions) = 11

Average branching factor = 3.5454545

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

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

Total solutions found = 1
(Check ww::*solutions* for list of all solutions.)

Number of steps in minimum path length solution = 2

Solution path from start state to goal state:
(1 (PUT B T C)) ;means put B, which is on T, on C
(2 (PUT A T B))

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

Shortest path solution is also a minimum duration solution

Evaluation took:
0.033 seconds of real time
0.031250 seconds of total run time (0.000000 user, 0.031250 system)
93.94% CPU
110,205,816 processor cycles
44,833,920 bytes consed

The program first outputs progressive improvements to solutions as planning proceeds, assuming the user has requested a full search for the shortest solution. In this case there is only one solution. Otherwise, planning exits with the first plan that meets the goal condition. Next is a statement about the kind of search conducted (tree or graph), 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, of which only a smaller number of unique states were actually analyzed (due to graph search not reanalyzing previously analyzed states). 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 as a result of considering all possible actions. The number of solution plans found is then reported, with reference to where all the successful plans are stored, if needed.

The best (shortest) plan found has the shallowest solution depth, and the sequence of actions in the plan is displayed 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 parameters 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 A, which is on T, onto B”. Next, the shortest duration plan is displayed, if the actions take time to complete, and the shortest duration plan is different from the shortest path length plan. Lastly, 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. 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*. 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. Instead of listing a large number of objects for a type, you can also use the operator *compute*, followed by Lisp code, to automatically compute the object list (see Problem 8: Triangle Peg Puzzle in the Appendix for an example).

Object Relations

A fundamental description of any object necessarily includes the properties it has and the relations it participates in. 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 global 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 table), says that tables 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.

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.

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 for a matching proposition. In the previous example, the relevant database is queried for a proposition matching the pattern (height block $real). If it finds a proposition like (height A 3.2), it instantiates the variable $real with the value 3.2. This instantiation is then available for further evaluation in the action as the value of the variable $real. 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. If there is no matching proposition, then the statement in the action rule is automatically false.

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 $hue) 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.

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 block1) implies (not (free)), and (not (holding block1) 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 return a true or false value when a condition holds, but others may return an arbitrary value (like function calls), or add and delete propositions from a state depending on an agent's actions (like *assert* statements) . Simple examples of logical statements appear in Part 1. The complete list of admissible logical statement forms is included in the later section: *The Variety of Logical Statements*.

Of particular note are the quantifier statements *exists* (or *forsome*) and *every* (or *forall*). 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 Pameter 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. 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) area)
    (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 one usually gives different names to different objects in a given context to avoid confusion.

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 value for ?block is first generated (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 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. Thus, while generated variables can be used anywhere, when there is a choice between using a generated or fluent variable, the fluent representation is 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 ?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.

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-base-relations
     (contents jug $integer)
     (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 jug $cap fluent)
    (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 Appendix. The *bind* operator basically tells the planner to lookup the values of all fluent variables in a statement. Those values are then available for use in subsequent statements.

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 structures. In the jugs example above, the fill action has a parameter list for the precondition, namely (?jug jug), and for the effect (?jug jug $cap fluent). The presentation format in a parameter list is always one or more variables followed by their type. The order of variable-type presentations in a precondition or quantified statement is arbitrary, but the order 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. Complex types expressed via the *either* construct are also allowed in 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 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, since their binding is always established through the use of the *bind* operator. However, fluent variables can appear in action effects 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 and all blocks are supports. For this situation all possible combinations will be generated (including ?block = A and ?support = A), since the types are different.

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 format solution steps. However, any new effect variable (not appearing in the precondition) must be included in the effect parameter list.

Goals

A planning goal is a logical statement that evaluates to true or false 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 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 the best solution (specified as *first-solution-sufficient* = t or nil), planning will either exit or continue the search for more solutions, respectively.

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 find 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, as are allowed in functions). 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.

Functions

One very flexible kind of statement that can appear in an action rule is a function call, used to query or update the current planning state. In the simplest cases, arbitrary built-in Lisp function calls may appear along with other 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. In standard programming style, function calls pass the current values of their arguments on to built-in Lisp or user-defined functions, which perform some computation before passing the result back to the calling statement. This result then is optionally available for subsequent use in the action rule.

A function provides a convenient way to encapsulate and hide the details of a complex analysis, thus making action rules more readable. And since functions are called from action rules 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.

Function calls can also be used to assign computed values to fluent arguments like $height1 and $height2. The following blocks world function takes a support and a fluent variable, 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 (setq $elev (elevation! ?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 *setq* 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. The exclamation (!) postfix is simply an optional designator distinguishing user-defined functions from other relations.

```
(define-query elevation! (?support)
  (do (bind (height ?support $h))
      (bind (on ?support $s))
      (if (not (support $s))
        (return-from elevation!
          $h)
        (return-from elevation!
          (+ $h (elevation! state $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, 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 function must consist of only one statement.

The syntax for a (query or update) function definition must include a name for the function (eg, elevation!), an argument list of variables which are passed into the function (eg, ?support), and a body consisting of a sequence of logical statements that return a value when the function is executed (using the formal *return-from* operator, or the value of the last function statement executed). Note that the elevation*!* function call within the Lisp *return-from* operator has an extra required 'state' initial argument. This is technically because the planner currently does not analyze the code inside of Lisp expressions, and the internal representation of all user-defined functions requires the current state as an initial argument.

There are two basic kinds of function specification, 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 in the calling statement. The *setq* 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! ?support)
  (assert (on ?block ?support)))
```

uses the query function stable! to check if a ?support is stable before putting a block on it.

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! ?transmitter)
```

might consolidate a large number of database conditions and updates associated with deactivation.

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 followup 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 preceeding effect statements are completed.

Any number of followup actions are allowed, each of which will be executed in strict sequence. The convention for including multiple followup 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 a function call, not a generic logic statement.

Assert vs Commit

Normally, effect statements are asserted (using the *assert* operator), which allows the planner to manage the order of database changes and avoid possible database inconsistencies. However, the *commit* operator can override this default behavior for special cases requiring an immediate database update. Such special cases may include recursive function calls, where later database changes may depend on earlier changes. Using *assert* in these cases is incorrect, since the database changes then would not be made until the top-level recursive call returns.

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 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) 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)))
```

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 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)
  (?sentry sentry)
  (assert (active ?sentry)))
```

The duration in an initialization action is always set to 0, but otherwise it looks and functions like a normal action.

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.

`(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 an assertion into the current state database.

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

```
(commit (loc ?sentry ?area)), or
(commit (not (loc ?sentry ?area)))
```

Commit statements immediately add or retract propositions, in sequence, from the current database when they are executed. By comparison, assert statements are consolidated before updating the database.

```
(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 in a precondition containing fluents (eg, $area) checks whether the proposition with the current variable bindings is in the database. attempts to instantiate the fluents on their first appearance. The value of the fluent is then available for subsequent use in the precondition, unless the instantiated proposition was not found in the database. If not found in the database, the statement is false, otherwise true. In an action effect the proposition containing a fluent previously instantiated in the precondition is simply asserted, as before. If not previously instantiated (ie, is new in the effect), then it must first be bound before being used. 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))
```

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.

```
(if (active ?sentry)
  (assert (dangerous ?sentry))
  (assert (benign ?sentry)))
```

A conditional *if* statement performs a test and depending on the true/false result selects one or more statements to assert (or commit). Its full format consists of three clauses (if <test> <then> <else>), although the <else> clause is optional. It makes no sense to use an *if* statement in a precondition, since it necessarily includes database updates. In an effect the test is performed as if it were included in the precondition. 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 function, indicated by an optional postfix exclamation (!) on a predicate or other computation. The sequence of argment variables must correspond to the order of those in the function's definition. The statement's value is the value returned by the function.

```
(setq $elevation (elevation! ?support))
```

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

```
(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.

```
(next (activate-transmitter! ?transmitter)) and/or
(finally (activate-receiver! ?receiver))
```
The *next* and *finally* operators are used in action effects for sequentially updating the current database (eg, for triggering follow-on updates after some initial updates are complete).

```
(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)).

```
(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 & 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 variable named *debug* (in the :ww package) controls the level of information output to the terminal during planning, and 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.

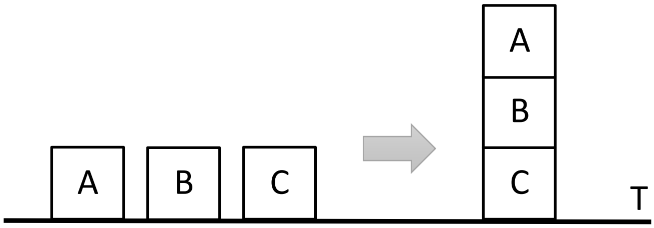
First load the program into a Lisp environment with (asdf:make “wouldwork” :force t). This should compile and load everything itemized in the wouldwork planner.asd file. To begin debugging or monitoring after loading, switch to the planner package by executing (in-package :ww) at the Lisp prompt. Then execute (setq *debug* 1), or some other level, to set the debugging level. Finally, begin running the program by executing (bnb::solve). Debugging output is displayed in the Lisp REPL window, along with normal output.

To assist with debugging individual actions, constraints, goals, functions, etc, you can also insert arbitrary 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)
 (do (assert (on ?block ?support))
     (if (bind (on ?block $s))
       (assert (not (on ?block $s))))))
```

APPENDIX: SAMPLE PROBLEMS

1. Blocks World Problem



Develop a plan to stack three blocks on a table.

Blocks Problem Specification:

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


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


(in-package :ww)  ;required


(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 ?support support)
  (and (cleartop! ?block)
       (cleartop! ?support))
  (?block block ?support support)
  (do (assert (on ?block ?support))
      (if (bind (on ?block $s))
        (assert (not (on ?block $s))))))


(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:

```
* (bnb::solve)

working ...

New path to goal found at depth = 2

Graph search process completed normally,
examining every state up to the depth cutoff.

Depth cutoff = 0

Maximum depth explored = 3

Total states processed = 68

Unique states encountered = 48
```

```
Program cycles (state expansions) = 25

Average branching factor = 2.24

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

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

Total solutions found = 1
(Check ww::*solutions* for list of all solutions.)

Number of steps in minimum path length solution = 2

Solution path from start state to goal state:
(1 (PUT B C))
(2 (PUT A B))

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

Shortest path solution is also a minimum duration
solution

Evaluation took:
  0.050 seconds of real time
  0.031250 seconds of total run time (0.015625 user,
0.015625 system)
  62.00% CPU
  166,262,634 processor cycles
  44,833,792 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


(setq *depth-cutoff* 10)
```

```
(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:

```
* (bnb::solve)

working ...

New path to goal found at depth = 10


Graph search process completed normally,
examining every state up to the depth cutoff.

Depth cutoff = 10

Maximum depth explored = 10

Total states processed = 32

Unique states encountered = 25

Program cycles (state expansions) = 16

Average branching factor = 1.5625
```

```
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 found = 1
(Check ww::*solutions* for list of all solutions.)

Number of steps in minimum path length solution = 10

Solution path from start state to goal state:
(1 (PICKUP BOX1 AREA1))
(2 (PUT BOX1 PLATE1 AREA1))
(3 (MOVE AREA1 AREA2))
(4 (PICKUP BOX2 AREA2))
(5 (MOVE AREA2 AREA1))
(6 (PUT BOX2 PLATE2 AREA1))
(7 (PICKUP BOX1 AREA1))
(8 (MOVE AREA1 AREA3))
(9 (PUT BOX1 PLATE3 AREA3))
(10 (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))

Shortest path solution is also a minimum duration
solution

Evaluation took:
  0.066 seconds of real time
  0.031250 seconds of total run time (0.015625 user,
0.015625 system)
  46.97% CPU
  224,305,194 processor cycles
  44,898,672 bytes consed
```

3. Jugs Problem

2-gallon jug 5-gallon jug reservoir

Fill/empty jugs at reservoir, pour water between jugs until exactly 1 gallon remains.

Jugs Problem Specification:

```
;;;; Filename: problem-2jugs.lisp


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


(in-package :ww)  ;required


(setq *depth-cutoff* 6)  ;set to expected # steps to goal


(define-types
    jug (jug1 jug2))


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


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

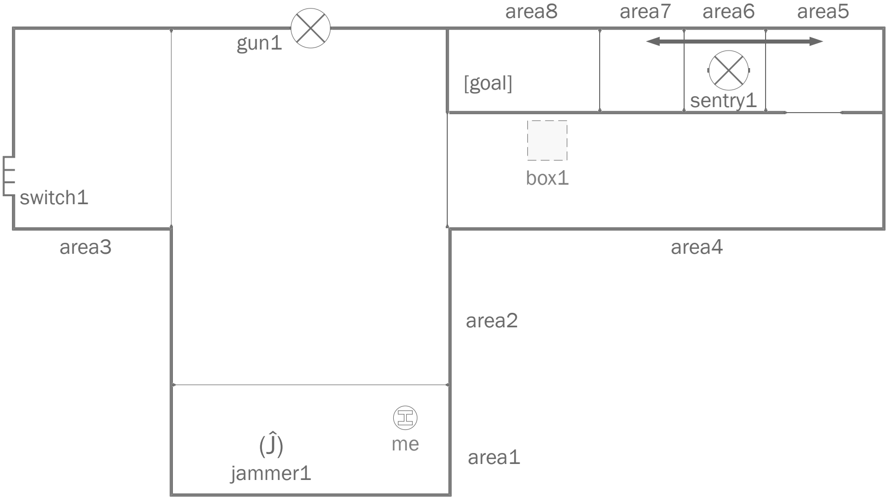
```
(define-action fill
    1
  (?jug jug)
  (and (bind (contents ?jug $amt))
       (bind (capacity ?jug $cap))
       (< $amt $cap))
  (?jug jug $cap fluent)
  (assert (contents ?jug $cap)))


(define-action empty
    1
  (?jug jug)
  (and (bind (contents ?jug $amt))
       (> $amt 0))
  (?jug jug)
  (assert (contents ?jug 0)))


(define-action pour  ;A into B
    1
  ((?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 fluent
   $capB fluent)
  (assert (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)))
```

Jugs Problem Solution:

```
* (bnb::solve)

working ...

New path to goal found at depth = 6

New path to goal found at depth = 4


Graph search process completed normally,
examining every state up to the depth cutoff.

Depth cutoff = 6

Maximum depth explored = 6

Total states processed = 24

Unique states encountered = 11

Program cycles (state expansions) = 8

Average branching factor = 2.25

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

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

Total solutions found = 2
(Check ww::*solutions* for list of all solutions.)

Number of steps in minimum path length solution = 4

Solution path from start state to goal state:
(1 (FILL JUG2 5))
(2 (POUR JUG2 5 JUG1 0 2))
(3 (EMPTY JUG1))
(4 (POUR JUG2 3 JUG1 0 2))

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

```
Shortest path solution is also a minimum duration
solution

Evaluation took:
  0.063 seconds of real time
  0.031250 seconds of total run time (0.000000 user,
0.031250 system)
  49.21% CPU
  217,240,842 processor cycles
  44,702,896 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. 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


(setq *tree-or-graph* 'graph)


(setq *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
  :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)
            (assert (not (red ?switch))
                    (green ?switch))
            (assert (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 sentry1 area6)
  (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:

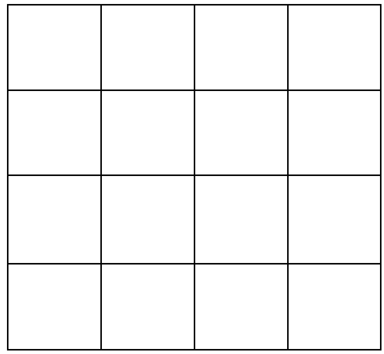
```
* (bnb::solve)

working ...

New path to goal found at depth = 16

New path to goal found at depth = 16


Graph search process completed normally,
examining every state up to the depth cutoff.

Depth cutoff = 16

Maximum depth explored = 16

Total states processed = 2,362

Unique states encountered = 905

Program cycles (state expansions) = 381

Average branching factor = 2.929134

Start state:
((LOC BOX1 AREA4) (LOC JAMMER1 AREA1) (LOC ME AREA1)
 (LOC SENTRY1 AREA6) (LOC SWITCH1 AREA3) (RED SWITCH1))

Goal:
(LOC ME AREA8)

Total solutions found = 2
(Check ww::*solutions* for list of all solutions.)

Number of steps in minimum path length solution = 16
```

```
Solution path from start state to goal state:
(1 (PICKUP JAMMER1 AREA1))
(2 (JAM GUN1 JAMMER1 AREA1))
(3 (MOVE AREA1 AREA2))
(4 (MOVE AREA2 AREA3))
(5 (THROW SWITCH1))
(6 (MOVE AREA3 AREA2))
(7 (MOVE AREA2 AREA1))
(8 (PICKUP JAMMER1 AREA1))
(9 (MOVE AREA1 AREA2))
(10 (MOVE AREA2 AREA4))
(11 (WAIT 1))
(12 (MOVE AREA4 AREA5))
(13 (JAM SENTRY1 JAMMER1 AREA5))
(14 (MOVE AREA5 AREA6))
(15 (MOVE AREA6 AREA7))
(16 (MOVE AREA7 AREA8))

Final state:
((GREEN SWITCH1) (JAMMING JAMMER1 SENTRY1)
 (LOC BOX1 AREA4) (LOC JAMMER1 AREA5) (LOC ME AREA8)
 (LOC SENTRY1 AREA7) (LOC SWITCH1 AREA3))

Shortest path solution is also a minimum duration
solution

Evaluation took:
  0.134 seconds of real time
  0.125000 seconds of total run time (0.093750 user,
0.031250 system)
  [ Run times consist of 0.016 seconds GC time, and 0.109
seconds non-GC time. ]
  93.28% CPU
  451,428,093 processor cycles
  61,200,000 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-4queens.lisp


;;; Problem specification for 4-queens.


(in-package :ww)  ;required


(setq *depth-cutoff* 4)


(setq *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)
          (not (next-row $row))
          (next-row (1+ $row))))


(define-init
    (next-row 1))


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

4-Queens Problem Solution:

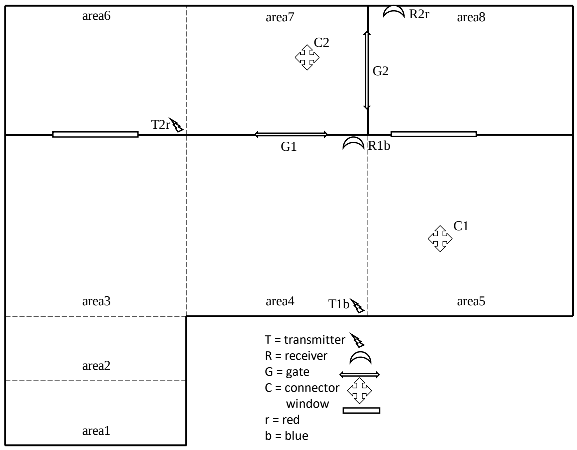
There are exactly 48 unique solutions to the 4-queens problem taking into account all possible successful arrangements of four distinct queens labeled queen1, queen2, queen3, queen4. Considering only the arrangements of queens on the board, however, there are only two distinct successful arrangements. 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 under 1 second.

```
* (bnb::solve)
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

Tree search process completed normally,
examining every state up to the depth cutoff.

Depth cutoff = 4

Maximum depth explored = 4

Total states processed = 185

Program cycles (states expanded) = 66

Average branching factor = 2.7878785

Start state:
((NEXT-ROW 1))

Goal:
(NEXT-ROW 5)

Total solutions found = 48
(Check ww::*solutions* for list of all solutions.)

Number of steps in minimum path length solution = 4

Solution path from start state to goal state:
(1 (PUT QUEEN1 1 2))
(2 (PUT QUEEN2 2 4))
(3 (PUT QUEEN3 3 1))
(4 (PUT QUEEN4 4 3))

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

Shortest path solution is also a minimum duration
solution
```

```
Evaluation took:
  0.112 seconds of real time
  0.000000 seconds of total run time (0.000000 user,
0.000000 system)
  0.00% CPU
  377,798,808 processor cycles
  4,969,024 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
```

```
(setq *depth-cutoff* 19)


(setq *first-solution-sufficient* nil)


(define-types
  myself      (me)
  gate        (gate1 gate2)
  barrier     ()
  jammer      ()
  connector   (connector1 connector2)
  box         ()
  fan         ()
  gears       ()
  ladder      ()
  rostrum     ()
  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 terminus 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 same-color! (?terminus1 ?terminus2)
  (and (bind (color ?terminus1 $hue1))
       (bind (color ?terminus2 $hue2))
       (eql $hue1 $hue2)))


(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))))))


(define-query sourced! (?conn-or-rcvr $hue $visits)
  (do (push ?conn-or-rcvr $visits)
      (or (exists (?t transmitter)
            (and (connecting ?t ?conn-or-rcvr)
                 (bind (color ?t $hue1))
                 (eql $hue1 $hue)))
          (exists (?c connector)
            (and (connecting ?c ?conn-or-rcvr)
                 (active ?c)
                 (bind (color ?c $hue1))
                 (eql $hue1 $hue)
                 (not (member ?c $visits))
                 (sourced! ?c $hue $visits))))))
```

```
;;;; UPDATE FUNCTIONS ;;;;


(define-update activate-connector! (?connector ?hue)
  (if (not (active ?connector))
    (commit (active ?connector)
          (color ?connector ?hue))))


(define-update deactivate-connector! (?connector)
  (if (and (active ?connector)
           (bind (color ?connector $hue)))
    (commit (not (active ?connector))
            (not (color ?connector $hue)))))


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


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


(define-update disconnect-connector! (?connector)
  (doall (?t terminus)
    (if (connecting ?connector ?t)
      (commit (not (connecting ?connector ?t))))))


(define-update disengage-jammer! (?jammer ?target)
  (assert (not (jamming ?jammer ?target))
          (if (not (exists (?j jammer)
                     (and (different ?j ?jammer)
                          (jamming ?j ?target))))
            (assert (active ?target)))))
```

```
(define-update chain-activate! (?connector)
  (if (and (active ?connector)
           (bind (color ?connector $hue)))
    (doall (?cr (either connector receiver))
      (if (connecting ?connector ?cr)
        (if (connector ?cr)
          (if (not (active ?cr))
            (do (activate-connector! ?cr $hue)
                (chain-activate! ?cr)))
          (if (receiver ?cr)
            (if (and (not (active ?cr))
                     (same-color! ?cr ?connector))
              (activate-receiver! ?cr))))))))


(define-update activate-connector-if! (?connector)
  (if (exists (?t transmitter)
        (and (connecting ?t ?connector)
             (bind (color ?t $hue))))
    (if (not (exists ((?t1 ?t2) transmitter)
               (and (connecting ?t1 ?connector)
                    (connecting ?t2 ?connector)
                    (bind (color ?t1 $hue1))
                    (bind (color ?t2 $hue2))
                    (not (eql $hue1 $hue2)))))
      (activate-connector! ?connector $hue))
    (if (exists (?c connector)
          (and (connecting ?c ?connector)
               (active ?c)
               (bind (color ?c $hue))))
      (if (not (exists ((?c1 ?c2) connector)
                 (and (connecting ?c1 ?connector)
                      (connecting ?c2 ?connector)
                      (active ?c1)
                      (active ?c2)
                      (bind (color ?c1 $hue1))
                      (bind (color ?c2 $hue2))
                      (not (eql $hue1 $hue2)))))
        (activate-connector! ?connector $hue)))))
```

```
(define-update deactivate-any-orphans! ()
  (do (doall (?c connector)
        (if (and (active ?c)
                 (bind (color ?c $hue))
                 (not (sourced! ?c $hue nil)))
          (deactivate-connector! ?c)))
      (doall (?r receiver)
        (if (and (active ?r)
                 (bind (color ?r $hue))
                 (not (sourced! ?r $hue nil)))
          (deactivate-receiver! ?r)))))


;;;; ACTIONS ;;;;


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

```
(define-action connect-to-2-terminus
    1
  ((?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)
  (do (assert (not (holding me $cargo))
              (loc $cargo $area)
              (connecting $cargo ?terminus1)
              (connecting $cargo ?terminus2))
      (next (activate-connector-if! $cargo))
      (finally (chain-activate! $cargo))))


(define-action connect-to-3-terminus
    1
  ((?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)
  (do (assert (not (holding me $cargo))
              (loc $cargo $area)
              (connecting $cargo ?terminus1)
              (connecting $cargo ?terminus2)
              (connecting $cargo ?terminus3))
      (next (activate-connector-if! $cargo))
      (finally (chain-activate! $cargo))))
```

```
(define-action jam
    1
  (?target target)
  (and (bind (holding me $cargo))
       (jammer $cargo)
       (bind (loc me $area))
       (los! $area ?target))
  (?target target $cargo fluent $area fluent)
  (assert (not (holding me $cargo))
          (loc $cargo $area)
          (jamming $cargo ?target)
          (not (active ?target))))


(define-action pickup-jammer
    1
  (?jammer jammer)
  (and (free me)
       (bind (loc me $area))
       (loc ?jammer $area))
  (?jammer jammer ($area $target) fluent)
  (do (assert (holding me ?jammer)
              (not (loc ?jammer $area)))
      (if (bind (jamming ?jammer $target))
        (disengage-jammer! ?jammer $target))))


(define-action pickup-connector
    1
  (?connector connector)
  (and (free me)
       (bind (loc me $area))
       (loc ?connector $area))
  (?connector connector $area fluent)
  (do (assert (holding me ?connector)
              (not (loc ?connector $area)))
      (next (deactivate-connector! ?connector))
      (next (disconnect-connector! ?connector))
      (finally (deactivate-any-orphans!))))
```

```
(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 (not (loc me $area1))
          (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
  (?station station ?area1 area)
  (assert (los0 ?area1 ?station)))


 (define-init-action init-visible0-locally
   ;any object is visible from its own local area
    0
  (?area area)
  (always-true)
  (?area area)
  (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)
  ((?area1 ?area2) area)
  (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)
  (?divider divider (?area1 ?area2) area)
  (assert (visible1 ?area1 ?divider ?area2)))
```

```
;;;; GOAL ;;;;


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

Smallspace Problem Solution:

```
* (bnb::solve)

working ...

New path to goal found at depth = 19


Graph search process completed normally,
examining every state up to the depth cutoff.

Depth cutoff = 19

Maximum depth explored = 19

Total states processed = 20,073

Unique states encountered = 6,999

Program cycles (state expansions) = 9,185

Average branching factor = 1.8589016

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 solutions found = 1
(Check ww::*solutions* for list of all solutions.)

Number of steps in minimum path length solution = 19
```

```
Solution path from start state to goal state:
(1 (PICKUP-CONNECTOR CONNECTOR1 AREA5))
(2 (CONNECT-TO-2-TERMINUS CONNECTOR1 TRANSMITTER1
                          RECEIVER1 AREA5))
(3 (MOVE AREA5 AREA4))
(4 (MOVE AREA4 AREA7))
(5 (PICKUP-CONNECTOR CONNECTOR2 AREA7))
(6 (MOVE AREA7 AREA6))
(7 (CONNECT-TO-1-TERMINUS CONNECTOR2 TRANSMITTER1 AREA6))
(8 (MOVE AREA6 AREA7))
(9 (MOVE AREA7 AREA4))
(10 (MOVE AREA4 AREA5))
(11 (PICKUP-CONNECTOR CONNECTOR1 AREA5))
(12 (CONNECT-TO-3-TERMINUS CONNECTOR1 RECEIVER1 RECEIVER2
                           CONNECTOR2 AREA5))
(13 (MOVE AREA5 AREA4))
(14 (MOVE AREA4 AREA7))
(15 (MOVE AREA7 AREA6))
(16 (PICKUP-CONNECTOR CONNECTOR2 AREA6))
(17 (CONNECT-TO-2-TERMINUS CONNECTOR2 TRANSMITTER2
                           CONNECTOR1 AREA6))
(18 (MOVE AREA6 AREA7))
(19 (MOVE AREA7 AREA8))

Final state:
((ACTIVE CONNECTOR1) (ACTIVE CONNECTOR2) (ACTIVE GATE1)
(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 RECEIVER1)
(CONNECTING CONNECTOR1 RECEIVER2)
(CONNECTING CONNECTOR2 CONNECTOR1)
(CONNECTING CONNECTOR2 TRANSMITTER2)
(CONNECTING RECEIVER1 CONNECTOR1)
(CONNECTING RECEIVER2 CONNECTOR1)
(CONNECTING TRANSMITTER2 CONNECTOR2) (FREE ME)
(LOC CONNECTOR1 AREA5) (LOC CONNECTOR2 AREA6)
(LOC ME AREA8))

Shortest path solution is also a minimum duration
solution
```

```
Evaluation took:
  3.505 seconds of real time
  3.437500 seconds of total run time (3.328125 user,
0.109375 system)
  [ Run times consist of 0.092 seconds GC time, and 3.346
seconds non-GC time. ]
  98.09% CPU
  11,800,854,039 processor cycles
  208,246,928 bytes consed  11,412,855,090 processor
cycles
  208,217,648 bytes consed
```

7. Constraint Satisfaction Problem (CSP)

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 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. 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.

Capt John Problem Specification

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


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


(in-package :ww)


(setq *tree-or-graph* 'tree)


(setq *first-solution-sufficient* t)


(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)
            (assert (next-col 1)
                    (next-row (1+ $row)))
            (assert (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

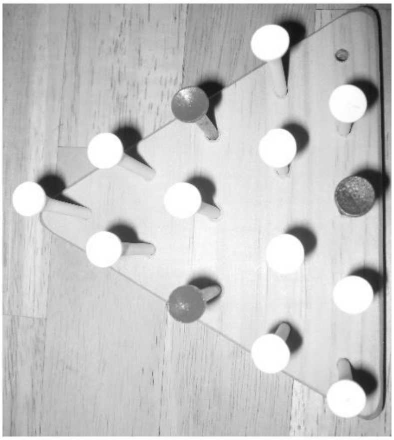
```
* (bnb::solve)
```

```
working ...

total states processed so far = 100,000
average branching factor = 1.5822474

total states processed so far = 200,000
average branching factor = 1.5821092

total states processed so far = 300,000
average branching factor = 1.5820583

total states processed so far = 400,000
average branching factor = 1.5820508

total states processed so far = 500,000
average branching factor = 1.582012

New path to goal found at depth = 9


Tree search ended with first solution found.

Depth cutoff = 0

Maximum depth explored = 9

Total states processed = 512,409

Program cycles (states expanded) = 323,896

Average branching factor = 1.5820255

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 found = 1
(Check ww::*solutions* for list of all solutions.)

Number of steps in minimum path length solution = 9

Solution path from start state to goal state:
(1 (PUT GUARD1 1 1))
(2 (PUT GRASS1 1 2))
(3 (PUT CREW1 1 3))
(4 (PUT GRASS2 2 1))
(5 (PUT GRASS3 2 2))
(6 (PUT JOHN 2 3))
(7 (PUT WASP 3 1))
(8 (PUT GUARD2 3 2))
(9 (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))

Duration of minimum time solution = 9

Minimum time solution path from start state to goal
state:
(1 (PUT GUARD1 1 1))
(2 (PUT GRASS1 1 2))
(3 (PUT CREW1 1 3))
(4 (PUT GRASS2 2 1))
(5 (PUT GRASS3 2 2))
(6 (PUT JOHN 2 3))
(7 (PUT WASP 3 1))
(8 (PUT GUARD2 3 2))
(9 (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:
  4.060 seconds of real time
  3.984375 seconds of total run time (3.906250 user,
0.078125 system)
  [ Run times consist of 0.109 seconds GC time, and 3.876
seconds non-GC time. ]
  98.13% CPU
  13,668,204,307 processor cycles
  1,858,028,928 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 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-peg.lisp

;;; Problem specification for triangle peg problem.

(in-package :ww)  ;required

(setq *tree-or-graph* 'graph)

(setq *first-solution-sufficient* t)

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

```
(define-types
  peg (compute (loop for i from 1
                     below (/ (* *N* (1+ *N*)) 2)
                 collect (intern (format nil "PEG~D" i))))
  coord (compute (loop for i from 1 to *N*
                   collect i)))


(define-dynamic-relations
  (loc peg $coord $coord $coord))


(define-query empty! (?x ?y ?z)  ;coordinates of a hole
  (not (exists (?p peg)
             (loc ?p ?x ?y ?z))))   ;x=row from left,
                                     ;y=row from right,
                                     ;z=row from bottom

(define-action jump
    1
  ((?peg1 ?peg2) peg)
  (and (bind (loc ?peg1 $x1 $y1 $z1))
       (bind (loc ?peg2 $x2 $y2 $z2))
       (or (and (= $x1 $x2)  ;aligned in x direction
                (< $x2 (1- *N*))
                (<= $y2 (- *N* $x2))
                (> $y2 1)
                (setq $delta (- $y2 $y1))
                (= (abs $delta) 1)
                (setq $target-x $x2)
                (setq $target-y (+ $y2 $delta))
                (setq $target-z (- $z2 $delta))
                (empty! $target-x $target-y $target-z))
           (and (= $y1 $y2)  ;aligned in y direction
                (< $y2 (1- *N*))
                (<= $z2 (- *N* $y2))
                (> $z2 1)
                (setq $delta (- $z2 $z1))
                (= (abs $delta) 1)
                (setq $target-x (- $x2 $delta))
                (setq $target-y $y2)
                (setq $target-z (+ $z2 $delta))
                (empty! $target-x $target-y $target-z))
           (and (= $z1 $z2)  ;aligned in z direction
                (< $z2 (1- *N*))
                (<= $x2 (- *N* $z2))
                (> $x2 1)
                (setq $delta (- $x2 $x1))
                (= (abs $delta) 1)
                (setq $target-x (+ $x2 $delta))
                (setq $target-y (- $y2 $delta))
```

```
                (setq $target-z $z2)
                (empty! $target-x $target-y $target-z))))
  (?peg1 peg ($x1 $y1 $z1) fluent ?peg2 peg
   ($x2 $y2 $z2) fluent)
  (assert (not (loc ?peg1 $x1 $y1 $z1))
          (not (loc ?peg2 $x2 $y2 $z2))
          (loc ?peg1 $target-x $target-y $target-z)))


(progn (format t "~&Initializing database...~%")
  (loop with pegs = (gethash 'peg *types*)
    for ?x from 1 to *N*
    do (loop with max = (1+ (- *N* ?x))
           for ?y from 1 to max
           for ?z from max downto 1
           unless (and (= ?x 1) (= ?y 1) (= ?z *N*))
           ;*db* is the name of the initial database
           ;update is the function that asserts a proposition
           ;into the database
           do (update *db* `(loc ,(pop pegs) ,?x ,?y ,?z)))))


(define-goal  ;only one peg left
  (exists (?p1 peg)
    (and (bind (loc ?p1 $x1 $y1 $z1))
         (not (exists (?p2 peg)
                (and (different ?p2 ?p1)
                     (bind (loc ?p2 $x2 $y2 $z2))))))))
```

Triangle Peg Problem Solution:

```
* (bnb::solve)

working ...

New path to goal found at depth = 13


Graph search ended with first solution found.

Depth cutoff = 0

Maximum depth explored = 13

Total states processed = 38

Unique states encountered = 38

Program cycles (state expansions) = 14
```

```
Average branching factor = 2.642857

Start state:
((LOC PEG1 1 2 4) (LOC PEG10 3 2 2) (LOC PEG11 3 3 1)
 (LOC PEG12 4 1 2) (LOC PEG13 4 2 1) (LOC PEG14 5 1 1)
 (LOC PEG2 1 3 3) (LOC PEG3 1 4 2) (LOC PEG4 1 5 1)
 (LOC PEG5 2 1 4) (LOC PEG6 2 2 3) (LOC PEG7 2 3 2)
 (LOC PEG8 2 4 1) (LOC PEG9 3 1 3))

Goal:
(EXISTS (?P1 PEG)
 (AND (BIND (LOC ?P1 $X1 $Y1 $Z1))
      (NOT (EXISTS (?P2 PEG)
             (AND (DIFFERENT ?P2 ?P1)
                  (BIND (LOC ?P2 $X2 $Y2 $Z2)))))))

Total solutions found = 1
(Check ww::*solutions* for list of all solutions.)

Number of steps in minimum path length solution = 13

Solution path from start state to goal state:
(1 (JUMP PEG2 1 3 3 PEG1 1 2 4))
(2 (JUMP PEG4 1 5 1 PEG3 1 4 2))
(3 (JUMP PEG10 3 2 2 PEG6 2 2 3))
(4 (JUMP PEG8 2 4 1 PEG7 2 3 2))
(5 (JUMP PEG5 2 1 4 PEG8 2 2 3))
(6 (JUMP PEG10 1 2 4 PEG4 1 3 3))
(7 (JUMP PEG10 1 4 2 PEG5 2 3 2))
(8 (JUMP PEG12 4 1 2 PEG9 3 1 3))
(9 (JUMP PEG2 1 1 5 PEG12 2 1 4))
(10 (JUMP PEG13 4 2 1 PEG11 3 3 1))
(11 (JUMP PEG2 3 1 3 PEG10 3 2 2))
(12 (JUMP PEG13 2 4 1 PEG2 3 3 1))
(13 (JUMP PEG14 5 1 1 PEG13 4 2 1))

Final state:
((LOC PEG14 3 3 1))

Duration of minimum time solution = 13

Minimum time solution path from start state to goal
state:
(1 (JUMP PEG2 1 3 3 PEG1 1 2 4))
(2 (JUMP PEG4 1 5 1 PEG3 1 4 2))
(3 (JUMP PEG10 3 2 2 PEG6 2 2 3))
(4 (JUMP PEG8 2 4 1 PEG7 2 3 2))
```

```
(5 (JUMP PEG5 2 1 4 PEG8 2 2 3))
(6 (JUMP PEG10 1 2 4 PEG4 1 3 3))
(7 (JUMP PEG10 1 4 2 PEG5 2 3 2))
(8 (JUMP PEG12 4 1 2 PEG9 3 1 3))
(9 (JUMP PEG2 1 1 5 PEG12 2 1 4))
(10 (JUMP PEG13 4 2 1 PEG11 3 3 1))
(11 (JUMP PEG2 3 1 3 PEG10 3 2 2))
(12 (JUMP PEG13 2 4 1 PEG2 3 3 1))
(13 (JUMP PEG14 5 1 1 PEG13 4 2 1))

Final state:
((LOC PEG14 3 3 1))

Evaluation took:
  0.111 seconds of real time
  0.062500 seconds of total run time (0.015625 user,
0.046875 system)
  [ Run times consist of 0.016 seconds GC time, and 0.047
seconds non-GC time. ]
  55.86% CPU
  370,632,942 processor cycles
  45,422,896 bytes consed
```