CLASSICAL PLANNING WITH THE WOULDWORK PLANNER

USER MANUAL

David Alan Brown

ISBN-13: 978-1546752059

ISBN-10: 1546752056

Table of Contents

CLASSICAL PLANNING WITH THE WOULDWORK PLANNER

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 also available in booklet 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 hundreds of thousands 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 functional variables, in addition to discrete variables)
- 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
- Derived relations (a.k.a. derived predicates) for simplifying action preconditions
- 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, derived relations, 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 minimal, 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:

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

See the file 'wouldwork planner.asd' in the Wouldwork\src\ directory for what Wouldwork loads.)

First, create a new problem specification as described below, and name the file problem.lisp, since the planner always looks for a specification with this name. Then to recompile and load the planner from the source files, at the Lisp prompt execute (asdf:make "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 set additional planner parameters after compiling and loading, switch first to the planner package by executing (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 different. 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.

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 assorted 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 two conditions in predicate logic, where ?block signifies the block to be put somewhere, ?b signifies some other arbitrary block, and ?support signifies the support on which ?block will be put:

```
(and (not (exists (?b block)
           (on ?b ?block)))
     (or (table ?support)
         (and (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 new so-called derived relations, like cleartop*, which can be used to simplify action conditions.) The second condition says that the support must either be a table,which has unbounded space on which to put blocks; or that the support is a block which itself has a clear top.

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 can be concisely expressed as:

```
(assert (on ?block ?support)
        (exists (?s support)
          (if (on ?block ?s))
            (not (on ?block ?s)))))
```

In other words, ?block now will be on ?support, and ?block is no longer on whatever its previous support was. Notice that the condition in an 'if' statement is always evaluated in the context of the precondition, even though it appears in the effect.

Bringing the precondition and the effect together then produces 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 name 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 ?support support)
  (and (not (exists (?b block)
               (on ?b ?block)))
       (or (table ?support)
           (and (block ?support)
                (not (exists (?b block)
                       (on ?b ?support)))))
  (?block block ?support support)
  (assert (on ?block ?support)
          (exists (?s support)
            (if (on ?block ?s))
              (not (on ?block ?s))))))
```

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 ?support values are A, B, C, and T. This leads to a set of (?block ?support) combination pairs ((A B) (A C) (A T) (B A) (B C) (B T) (C A) (C B) (C 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. 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.

Note that each of the two ‘exists’ expressions above function somewhat differently, depending on whether it is in the precondition or the effect. A quantified expression in a precondition is simply a test, returning true or false. Since precondition statements are typically joined by ‘and’, any false statement will cause the entire precondition to fail, while a true statement will allow processing to continue on to the next statement. However in an effect, a quantified expression is a generator for generating instances of its local parameters—eg, (?s support). In the example above, if the block to be moved (?block) happens to be on any support (?s), then the block will no longer be on that support. Should the ‘exists’ expression become satisfied for some ?s, it stops generating immediately. But regardless of whether it returns true or false, processing continues with the next effect statement.

In general, logical expressions in an action can be headed by any of the following constants—exists, forall, and, or, not, if, <relation>, <derived relation>, <function>, <lisp function> (see Part 2 for a full list). An argument in a logical expression can generally be a <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 S-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 a potential 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 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 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, but the planner will 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 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

A sample output plan plus varied diagnostic information for the blocks world problem is shown below.

```
* (bnb::solve)

New path to goal found at depth = 2


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

Depth cutoff = 0

Total states processed = 25

Unique states encountered = 12
```

```
Program cycles (state expansions) = 7

Average branching factor = 3.142857


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

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

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

Shortest path solution is also a minimum duration solution

Evaluation took:
  0.038 seconds of real time
  0.015625 seconds of total run time (0.000000 user, 0.015625 system)
  42.11% CPU
  129,500,937 processor cycles
  44,768,320 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. 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 ?support support) would correctly display as (put A T) meaning “put A on T”, rather than (put T A), which would be backwards and make no sense. 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 specifications 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), while block has objects (A B C) or, perhaps in a more perspicuous specification, (block1 block2 block3).

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. 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 a 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. (However, in some circumstances it is useful to return a value of nil instead, to indicate that no proposition was found. Use the ‘bind’ operator for these cases, as discussed later.)

Also, 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.

Derived Relations (a.k.a. Derived Predicates)

As mentioned previously, action rules can oftentimes be simplified and expressed more perspicuously, by using shorthand statements that stand for complex predicate logic statements. Derived relations are used to specify these shorthand statements, allowing one statement to expand into many others before planning begins. The previous example of the blocks world action rule for putting a block on some support, repeated here, is a case in point:

```
(define-action put
    0
    (?block block ?support support)
    (and (not (exists (?b block)
                (on ?b ?block)))
         (or (table ?support)
             (and (block ?support)
                  (not (exists (?b block)
                         (on ?b ?support))))
   (?block block ?support support)
   (and (on ?block ?support)
        (exists (?b block)
          (if (on ?block ?b)
            (not (on ?block ?b)))))
```

This rule is made more readable by defining a new relation ‘cleartop*’, which tests whether there is another block on top of a block (thereby preventing the movement of the block):

```
(define-derived-relations
  (cleartop* ?block)  (not (exists (?b block)
                             (on ?b ?block))))
```

The left-hand-side of the definition specifies the derived relation and its arguments, while the right-hand-side is the expansion. The asterisk (*) attached to the name is simply a convention distinguishing derived relations from other base relations in action rules. Note also that the new derived relation’s arguments can appear as free variables in the expansion, but that all other variables in the expansion must be bound. Derived predicates may also appear in other derived predicates, making it convenient to express complex relations in terms familiar to the problem domain. Operationally speaking, all derived predicates are fully expanded and substituted into the actions in which they appear, before planning begins. Rewriting the action with cleartop* then yields:

```
(define-action put
    0
    (?block block ?support support)
    (and (cleartop* ?block)
         (or (table ?support)
             (and (block ?support)
                  (cleartop* ?support))))
   (?block block ?support support)
   (and (on ?block ?support)
        (exists (?b block)
          (if (on ?block ?b)
            (not (on ?block ?b))))))
```

Further simplification might then replace the 'or' clause with something like (accessible* ?support).

In general, derived relations can appear both in the preconditions or effects of action rules, in functions, in goals, or any other place where logical expressions are evaluated. When a derived relation appears in the effect of an action, however, it is better thought of as a concise procedure for updating a database, rather than as a test on a database.

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)))
```

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)
    (and (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

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 $real)
    (capacity jug $real))
```

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 (a real number, indicated by the required $ prefix). And both relations are clearly functional. 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 ($amt $cap) fluent)
    (and (contents ?jug $amt)
         (capacity ?jug $cap)
         (< $amt $cap))
    (?jug jug $cap fluent)
    (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. Note also that if a fluent appears as an argument of a derived relation in an action rule, the fluent must also appear in the expansion of the derived relation with the exact same name, because the expansion will simply be embedded in the rule. Since derived relations are normally written using the standard ? variables, the planner automatically substitutes fluent variables for the standard variables at the appropriate argument positions.

Parameter Lists & Let

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 ($amt $cap) fluent), and for the effect (?jug jug $amt fluent). A variable with a ? prefix tells the planner to generate instances from its associated type, listed after the variable. A $ prefix tells the planner to consult the current state's database to get the variable's value. The presentation format in a parameter list is always one or more variables followed by their type, but the order of variable-type presentation pairs is arbitrary. Complex types expressed via the 'either' construct are also allowed in parameter lists—eg, (?pet (either dog cat pig)).

As a general rule, every variable (generated or fluent) must be specified before it is subsequently used in a logical statement. Thus, a parameter list establishes a *scope* in which its variables can be used. The planner uses parameter lists mainly to control the processing of generated (?) variables, so all generated variables must appear in a governing parameter list. Fluent variables may be included in a parameter list for convenience. However, there are other occasions for using fluent variables that are not associated with generated variables. For those cases the 'let' operator establishes a scope for using fluent variables, similar to parameter lists. Its form looks like:

```
(let ($amt $cap)
  (capacity jug1 $cap)
  (contents jug1 $amt)
  (= $amt $cap))
```

This statement first specifies the local fluent variables, binds them to values, and then checks whether the jug is full or not. The let statement simply returns the value of its last statement, if needed, which in this case is either true or false. Let statements can be used anywhere to establish a local scope for fluent variables.

The more global parameter list may include several generated variables all having the same type. In the blocks world, if the size of the blocks was relevant to determining if two blocks could be stacked (eg, if only smaller blocks could be stacked on larger ones), the precondition parameter list for stacking might look like ((?block1 ?block2) block). Here there are two variables of the same type, which will each be instantiated by blocks A, B, and C. However, the planner always assumes that distinct variable names refer to different objects, so ?block1 and ?block2 will never take the same value (eg, ?block1 = A and ?block2 = A). This convention is consistent with how we normally talk about individually named objects. It also simplifies action rules, since then there is no need to specify that (not (eql ?block1 ?block2). But in the case of overlapping types, as in a blocks world parameter list like (?block block ?support support), some supports are blocks and some 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. However, the planner uses the effect parameter list to format the printout for an action, giving the user control over how each action step is displayed in the final plan. For the jugs example, an instantiated fill action could printout as (fill jug2 5), where 5 is the capacity of jug2. If the effect parameter list were left empty—ie, simply ()—the printout would instead look like (fill).

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). Global constraints place 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, evaluate to t (true) when the constraint is satisfied. 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 derived relations or 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

When fluents are needed to express variable values in action rules, it may also be useful to specify functions for calculating with those fluents. Arbitrary Common Lisp built-in functions that evaluate to t or nil may appear along with other logical statements. For example, an expression in a precondition like (< $height1 $height2) would evaluate to t (true) if the current value of $height1 were less than $height2.

More complex user-defined functions can return more than the basic truth values of t and nil. User-defined functions can also compute and instantiate fluent arguments like $height1 and $height2, where those returned fluent values can then be used in other subsequent statements in an action.

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 an optional designator distinguishing functions from logical operators or derived relations. The function that computes the elevation of a support is as follows:

```
(define-precondition-function elevation! (?support)
  (let ($h $s)  ;local variables
    (height ?support $h)
    (bind (on ?support $s))
    (if (eql $s nil)
      (return-from elevation!
        $h)
      (return-from elevation!
        (+ $h (elevation! state $s))))))
```

There are two kinds of function specification, depending on whether the function call appears in a precondition or effect part of an action. The ‘define-precondition-function’ specification is for preconditions, and returns a value that can be assigned to a fluent variable in the calling action. For example, a particular function call might instantiate as (setq $elev (elevation! A)), which recursively gets the net current elevation of block A, and saves the resulting value in $elev. Alternately, ‘define-effect-function’ is for functions appearing in an effect, and can be used for complex analysis leading to new assertions into a database, much like derived relations. Functions, however, are more flexible than derived relations, since they are called from, rather than being embedded in, action rules. Functions, for example, support recursion.

The syntax for a function definition must include a name for the function (eg, elevation!), an argument list of variables which are passed into the function (eg, ?support), a parameter list of variables which are local to the function itself (eg, $h and $s both of which are fluents), and a body of logical statements that return a value when the function is executed (eg, using the formal ‘return-from’, or more simply, the last Common Lisp form evaluated).

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. Without the bind operator, the statement (on ?support $s) would simply be false if $s did not exist, thereby exiting the precondition before checking the following if statement.

Exogenous Events

Exogenous events are events that happen in the planning environment independent of the planning agent's actions. Typically, the agent must react to or otherwise take into account such happenings along the way to achieving the goal. The user specifies exogenous events before planning begins in a program schedule, which becomes part of the problem specification. As these events are prespecified by the user, they are technically known as 'timed initial literals'. The planner then 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
  (not (exists (?s sentry ?a area)  ;kill situation
    (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 being disabled means the sentry's program schedule is temporarily 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 change 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 actions 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 init-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, derived relations, goals, init-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.

```
(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) 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 previously instantiated fluent is simply asserted, as before. 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 cannot be located in more than one place at a time.

```
(bind (loc ?sentry $area1))
```

The bind operator on a fluent statement allows binding of its fluent variables to nil, if a corresponding proposition is not found in the database. (Non-nil bindings are returned if a matching proposition is found.) It is normally used in a precondition to return true whether the proposition is in the database or not. This can allow further analysis of the proposition's absence in subsequent statements.

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

A conditional statement performs a test and depending on the true/false result selects a statement to assert. 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 one or more assertions. 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 actions required to cover a problem domain. Note that the 'assert' operators in the <then> and <else> clauses above are optional, since they are always implied.

```
(cleartop* ?block)
```

A statement that refers to a derived relation, indicated by an optional postfix asterisk (*) on a predicate. A derived relation statement is essentially a macro call. During the initialization phase of planning, all derived relation statements are replaced by their expansions, as specified in 'define-derived-relations'. Any free variables in the expansion are replaced by the corresponding order of arguments specified in the macro call. Liberal use of derived relations can make action rules much easier to read and analyze.

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

This statement incorporates a function call, indicated by the optional exclamation (!) postfix on a predicate. In this case the function takes one argument, ?support, recursively 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. Unlike derived relation statements, function statements are evaluated during planning, so they can be used to analyze dynamic situations.

```
(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 the example above 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 $area fluent)
  (and (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 should have at least one ? generator variable, and the condition is tested for each instantiation. If any instantiation of the condition is true, the entire existential statement is true, otherwise it is false. In an action precondition the resulting true/false value of the existential statement is relevant, but in an action effect, the statement is used only to make assertions about the first instantiation that satisfies the condition part.

```
(forall ((?sentry1 ?sentry2) sentry $area fluent)
  (if (and (loc ?sentry1 $area)
           (loc ?sentry2 $area))
    (conflict ?sentry1 ?sentry2)))
```

A universal statement is the quantified counterpart to an existential statement, and returns true only if all instantiations of its conditional part are true. As for the existential statement, a universal statement's return value true/false is relevant in a precondition, but not in an effect, where it is typically used to make assertions.

```
(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'.

```
(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 operator can also appear as the predicate 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* controls the level of information output to the terminal during planning, and can take a value of 0, 1, 2, 3, or 4 (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 minimal information about each planning step taken, which may help determine why a particular action failed. Level 3 outputs level 2 plus more detailed information about each step. Level 4 includes all the information output by level 3, but temporarily halts program execution after each step, allowing the user to carefully examine all possible actions before continuing to the next step.

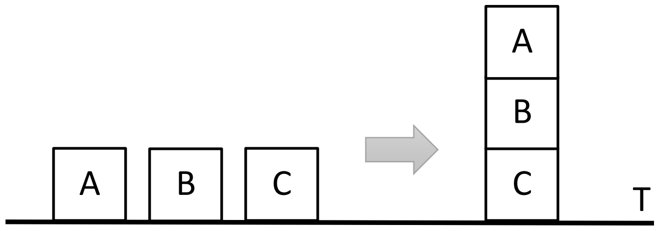
First load the program into a Lisp environment with (asdf:make “wouldwork” :force t). This should compile and load everything itemized in the ‘wouldwork planner.asd’ file. To begin debugging or monitoring after loading, switch to the planner package by executing (in-package :ww) at the Lisp prompt. Then 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, derived relations, 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
    0
    (?block block ?support support)
    (and (ut::prt ?block ?support)
         (not (exists (?b block)
                (on ?b ?block)))
         (or (table ?support)
             (and (block ?support)
                  (not (exists (?b block)
                         (on ?b ?support)))))))
   (?block block ?support support)
   (and (on ?block ?support)
        (exists (?s support)
          (if (on ?block ?s))
            (not (on ?block ?s)))))
```

APPENDIX: SAMPLE PROBLEMS

1. Blocks World Problem



Develop a plan to stack three blocks on a table.

<u>Blocks Problem Specification</u>:

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

;;; Problem specification for a blocks world problem for stacking 3 blocks
;;; named A, B, and C, on a table named T.

(in-package :pl)  ;required

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


(define-base-relations
    (on block support))


(define-action put
    0
    (?block block ?support support)
    (and (not (exists (?b block)
                      (on ?b ?block)))
         (or (table ?support)
             (and (block ?support)
                  (not (exists (?b block)
                         (on ?b ?support))))))
   (?block block ?support support)
   (and (on ?block ?support)
        (exists (?s support)
          (if (on ?block ?s))
            (not (on ?block ?s)))))
```

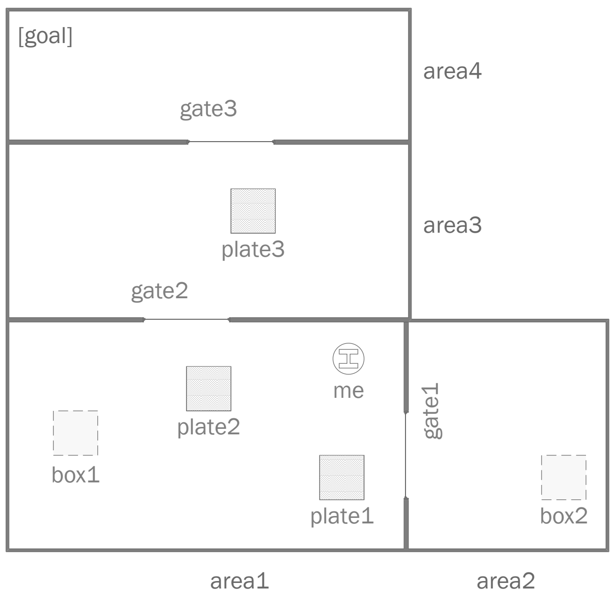
```
(define-init
  (on A T)
  (on B T)
  (on C T))


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

Blocks Problem Solution:

```
* (bnb::solve)

New path to goal found at depth = 2


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

Depth cutoff = 0

Total states processed = 25

Unique states encountered = 12

Program cycles (state expansions) = 7

Average branching factor = 3.142857

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

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

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

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

```
Evaluation took:
  0.039 seconds of real time
  0.015625 seconds of total run time (0.015625 user, 0.000000 system)
  [ Run times consist of 0.015 seconds GC time, and 0.001 seconds non-GC
time. ]
  41.03% CPU
  133,596,059 processor cycles
  44,702,944 bytes consed
```

2. Boxes Problem



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

<u>Boxes Problem Specification:</u>

```
;;;; 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-derived-relations
  (free* me)                     (not (exists (?b box)
                                        (holding me ?b)))

  (cleartop* ?plate)             (not (exists (?b box)
                                        (on ?b ?plate)))

  (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))
```

<u>Boxes Problem Solution</u>:

```
* (bnb::solve)

New path to goal found at depth = 10


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

Depth cutoff = 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

Shortest 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:
  0.044 seconds of real time
  0.031250 seconds of total run time (0.031250 user, 0.000000 system)
  [ Run times consist of 0.016 seconds GC time, and 0.016 seconds non-GC
time. ]
  70.45% CPU
  148,212,894 processor cycles
  44,833,552 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 ($amt $cap) fluent)
  (and (contents ?jug $amt)
       (capacity ?jug $cap)
       (< $amt $cap))
  (?jug jug $cap fluent)
  (assert (contents ?jug $cap)))
```

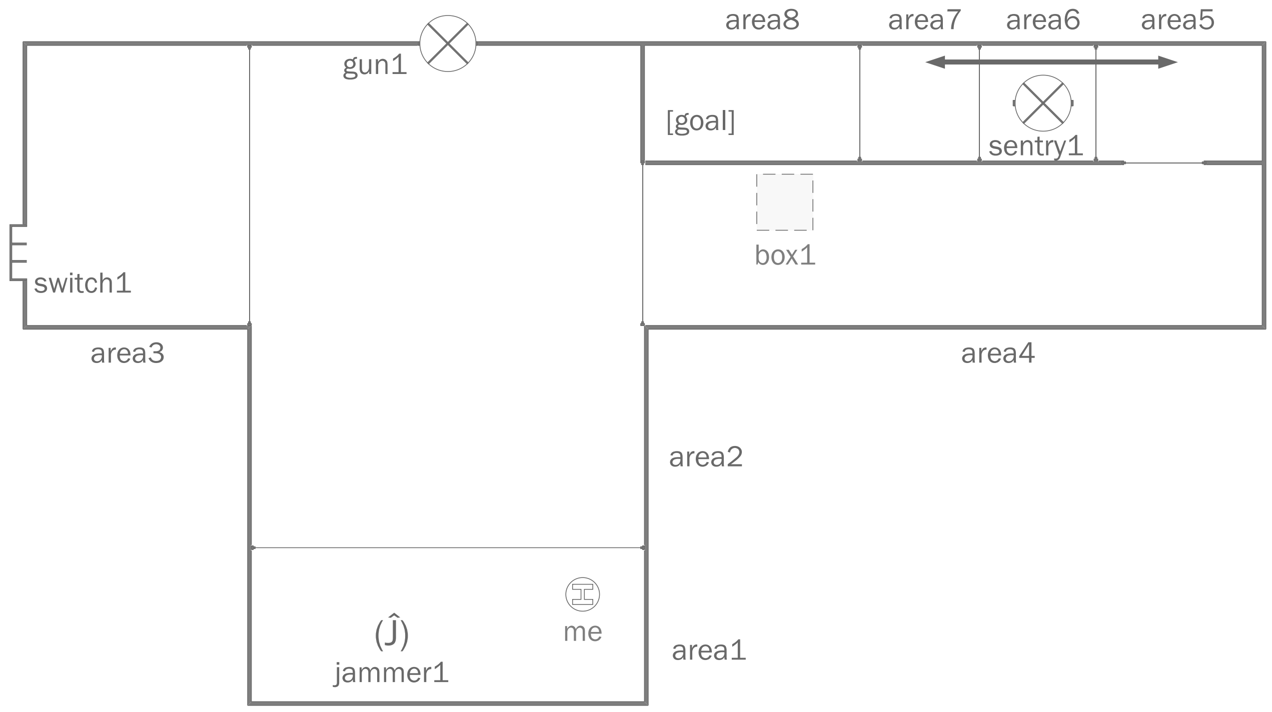
```
(define-action empty
    1
  (?jug jug $amt fluent)
  (and (contents ?jug $amt)
       (> $amt 0))
  (?jug jug)
  (assert (contents ?jug 0)))


(define-action pour  ;A into B
    1
  ((?jugA ?jugB) jug ($amtA $amtB $capB) fluent)
  (and (contents ?jugA $amtA)
       (> $amtA 0)
       (contents ?jugB $amtB)
       (capacity ?jugB $capB)
       (< $amtB $capB))
  ((?jugA ?jugB) jug ($amtA $amtB $capB) fluent)
  (assert (if (<= $amtA (- $capB $amtB))
             (and (contents ?jugA 0)
                  (contents ?jugB (+ $amtB $amtA)))
             (and (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)

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

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

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

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

Shortest path solution is also a minimum duration solution

Evaluation took:
  0.041 seconds of real time
  0.015625 seconds of total run time (0.015625 user, 0.000000 system)
  [ Run times consist of 0.016 seconds GC time, and 0.000 seconds non-GC
time. ]
  39.02% CPU
  137,039,990 processor cycles
  44,702,848 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.
;;; See sentry-problem in user manual appendix.


(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
  (always-true)
  (adjacent area area)
  (los area target)  ;line-of-sight exists
  (visible area area)  ;area is wholly visible from another area
  (controls switch gun)
  (watches gun area))


(define-derived-relations
  (free* me)                 (not (exists (?c cargo)
                                    (holding me ?c)))

  (passable* ?area1 ?area2)  (adjacent ?area1 ?area2)

  (safe* ?area)              (not (exists (?g gun)
                                    (and (watches ?g ?area)
                                         (active* ?g))))

  (active* ?threat)          (not (or (exists (?j jammer)
                                         (jamming ?j ?threat))
                                       (forall (?s switch)
                                         (and (controls ?s ?threat)
                                              (green ?s)))))
  )


(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 ?area2 area ?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 ?area area)
  (assert (if (red ?switch)
            (and (not (red ?switch))
                 (green ?switch))
            (and (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 unknown time for next exogenous event
  (?area area)
  (loc me ?area)
  (?area 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))
```

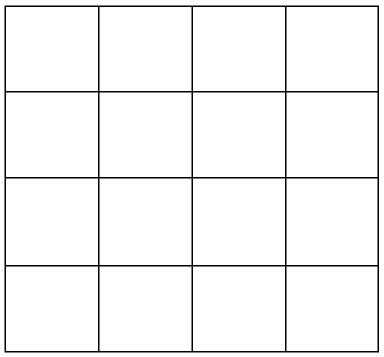
Sentry Problem Solution:

```
* (bnb::solve)

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

Total states processed = 2,064

Unique states encountered = 643

Program cycles (state expansions) = 329

Average branching factor = 2.939209

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

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

```
Final state:
((GREEN SWITCH1) (JAMMING JAMMER1 SENTRY1) (LOC BOX1 AREA4)
 (LOC JAMMER1 AREA6) (LOC ME AREA8) (LOC SENTRY1 AREA6) (LOC SWITCH1 AREA3)
 (WAITING))

Shortest path solution is also a minimum duration solution

Evaluation took:
  0.087 seconds of real time
  0.078125 seconds of total run time (0.046875 user, 0.031250 system)
  89.66% CPU
  293,598,806 processor cycles
  59,300,832 bytes consed
```

5. 4-Queens Problem

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



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.

<u>4-Queens Problem Specification:</u>

```
;;;; 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 $row fluent ?column column)
  (and (not (placed ?queen))
       (next-row $row)
       (not (exists (?q queen $r fluent ?c column)
              (and (placed ?q)
                   (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))
```

<u>4-Queens Problem Solution</u>:

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)

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

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

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

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

Shortest path solution is also a minimum duration solution

Evaluation took:
  0.063 seconds of real time
  0.031250 seconds of total run time (0.015625 user, 0.015625 system)
  49.21% CPU
  209,491,794 processor cycles
  5,034,640 bytes consed
```

6. Crater Problem

This is an example of a rather complex and lengthy specification that illustrates the integration of many Wouldwork planner 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 which controls a gate. Once the receiver detects a beam of the proper color, it opens the gate.

Crater Problem Specification:

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

;;; Problem specification (in Talos Principle)
;;; for the (second) Nexus-2 crater problem in Road to Gehenna.

(in-package :ww)  ;required

(setq *depth-cutoff* 9)

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


(define-types
  myself      (me)
  gate        (gate4 gate5 gate6)
  barrier     (barrier3 barrier4)
  jammer      (jammer1)
  connector   (connector2 connector3 connector4 connector5 connector6)
  box         (box1 box2 box3 box4 box5)
  fan         (fan1 fan2 fan3 fan4)
  gears       (gears2 gears3)
  ladder      (ladder3 ladder4)
  rostrum     (rostrum2)
  hue         (blue red)
  transmitter (transmitter2)
  receiver    (receiver7 receiver8 receiver9 receiver10 receiver11 receiver12
               receiver13 receiver14)
  area        (area1 area2 area8 area9 area10 area11 area12)
  divider     (either gate barrier)
  cargo       (either jammer connector box fan)
  target      (either gate gears)
  terminus    (either transmitter receiver connector)
  fixture     (either transmitter receiver gears ladder rostrum)
  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 gate gears))
  (inactive (either connector gate gears))
  (color terminus $hue))


(define-static-relations
  (adjacent area area)  ;agent can always move unimpeded between areas
  (locale fixture area)
  (separates divider area area)
  (climbable> ladder area area)
  (height support $real)
  (controls receiver (either $gate $gears))
  (controls2 receiver receiver $gate)  ;gate controlled by two receivers
  (los0 area (either gate fixture))  ;clear los from area to a gate/fixture
  (los1 area divider (either gate fixture))
  (los2 area divider divider (either gate fixture))
  (visible0 area area) ;could see mobile object in area from a given area
  (visible1 area divider area)
  (visible2 area divider divider area))


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

  (active (either gears connector gate)) ->
       (not (inactive (either gate gears connector)))

  (inactive (either connector gears gate)) ->
       (not (active (either connector gate gears))))


(define-derived-relations

  ;predicates for simplifying preconditions
  (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)
                                        (inactive ?g))))

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

```
(visible-thru-1-divider* ?area1 ?area2)  (exists (?d divider)
                                            (and (visible1 ?area1 ?d ?area2)
                                                 (or (barrier ?d)
                                                     (and (gate ?d)
                                                          (inactive ?d)))))


(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)
                                                     (inactive ?d2))
                                                (and (barrier ?d2)
                                                     (gate ?d1)
                                                     (inactive ?d1))
                                                (and (gate ?d1)
                                                     (inactive ?d1)
                                                     (gate ?d2)
                                                     (inactive ?d2)))))


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


(los-thru-1-divider* ?area ?station)  (exists (?d divider)
                                        (and (los1 ?area ?d ?station)
                                             (or (barrier ?d)
                                                 (and (gate ?d)
                                                      (inactive ?d)))))


(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)
                                                       (inactive ?d2))
                                                  (and (barrier ?d2)
                                                       (gate ?d1)
                                                       (inactive ?d1))
                                                  (and (gate ?d1)
                                                       (inactive ?d1)
                                                       (gate ?d2)
                                                       (inactive ?d2)))))
```

```
  (connectable* ?area ?terminus)  (or (los* ?area ?terminus)
                                      (and (connector ?terminus)
                                           (exists (?a area)
                                             (and (loc ?terminus ?a)
                                                  (visible* ?area ?a)))))

  (compatible-colors* ?hue1 ?hue2)  (or (eql ?hue1 ?hue2)
                                        (eql ?hue1 nil)
                                        (eql ?hue2 nil))


  (source* ?terminus)  (or (transmitter ?terminus)
                           (active ?terminus))


  (colorable* ?terminus)  (and (connector ?terminus)
                               (inactive ?terminus))


  ;derived relations for simplifying effects
  (disconnect-connector* ?connector)
        (forall (?t terminus)
         (if (connecting ?connector ?t)
           (assert (not (connecting ?connector ?t))
                   (if (and (active ?t)
                            (not (exists (?s (either transmitter connector))
                                   (and (connecting ?t ?s)
                                        (source* ?s)))))
                     (assert (inactive ?t)
                             (bind (color ?t $hue))
                             (if (not (eql $hue nil))
                               (not (color ?t $hue)))))))))


  (activate-terminus1-given-terminus2* ?terminus1 ?terminus2)
           (if (and (connector ?terminus1)
                    (inactive ?terminus1)
                    (source* ?terminus2))
             (active ?terminus1))
)


(define-precondition-function elevation! (?support)
  (($h $s) fluent)
  (and (height ?support $h)
       (bind (on ?support $s))
       (if (eql $s nil)
         (return-from elevation!
           $h)
         (return-from elevation!
           (+ $h (elevation! $s))))))
```

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


(define-action connect-to-2-terminus  ;using held connector
    1
  ($cargo fluent (?terminus1 ?terminus2) terminus ($area $hue1 $hue2) fluent)
  (and (holding me $cargo)
       (connector $cargo)
       (loc me $area)
       (connectable* $area ?terminus1)
       (connectable* $area ?terminus2)
       (bind (color ?terminus1 $hue1))
       (bind (color ?terminus2 $hue2))
       (compatible-colors* $hue1 $hue2))
  ($cargo fluent ?terminus1 terminus $hue1 fluent ?terminus2 terminus
   $hue2 fluent $area fluent)
  (assert (not (holding me $cargo))
          (loc $cargo $area)
          (connecting $cargo ?terminus1)
          (connecting $cargo ?terminus2)
          (if (not (eql $hue1 nil))
            (assert (color $cargo $hue1)
                    (if (colorable* ?terminus2)
                      (color ?terminus2 $hue1))))
          (if (not (eql $hue2 nil))
            (assert (color $cargo $hue2)
                    (if (colorable* ?terminus1)
                      (color ?terminus1 $hue2))))
          (activate-terminus1-given-terminus2* ?terminus1 ?terminus2)
          (activate-terminus1-given-terminus2* ?terminus2 ?terminus1)
          (if (or (source* ?terminus1)
                  (source* ?terminus2))
            (active $cargo))))



(define-action connect-to-1-terminus  ;using held connector
    1
  ($cargo fluent ?terminus terminus ($area $hue) fluent)
  (and (holding me $cargo)
       (connector $cargo)
       (loc me $area)
       (connectable* $area ?terminus)
       (bind (color ?terminus $hue)))
  ($cargo fluent ?terminus terminus ($hue $area) fluent)
  (assert (not (holding me $cargo))
          (loc $cargo $area)
          (connecting $cargo ?terminus)
          (if (not (eql $hue nil))
            (color $cargo $hue))
          (activate-terminus1-given-terminus2* $cargo ?terminus)))
```

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


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

(define-action pickup-connector
    1
  (?connector connector ($area $hue) fluent)
  (and (free me)
       (loc me $area)
       (loc ?connector $area)
       (bind (color ?connector $hue)))
  (?connector connector ($area $hue) fluent)
  (assert (holding me ?connector)
          (not (loc ?connector $area))
          (if (not (eql $hue nil))
            (not (color ?connector $hue)))
          (if (active ?connector)
            (not (active ?connector)))
          (disconnect-connector* ?connector)))

(define-action put
    1
  (?support support ($cargo $elev $area) fluent)
  (and (holding me $cargo)
       (loc me $area)
       (loc ?support $area)
       (not (exists (?c cargo)  ;cleartop ?support
              (on ?c ?support)))
       (setq $elev (elevation! ?support)) ;function, not derived predicate
       (<= $elev 1))
  ($cargo fluent ?support support ($elev $area) fluent)
  (assert (on $cargo ?support)
          (loc $cargo $area)
          (not (holding me $cargo))))
```

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


(define-action move
    1
  ($area1 fluent ?area2 area)
  (and (loc me $area1)
       (different $area1 ?area2)
       (passable* $area1 ?area2))
  ($area1 fluent ?area2 area)
  (assert (not (loc me $area1))
          (loc me ?area2)))


(define-init
  ;dynamic
  (loc me area1)
  (free me)
  (loc jammer1 area1)
  (loc connector2 area2) (loc connector3 area8) (loc connector4 area10)
  (loc connector5 area11) (loc connector6 area11)
  (loc box1 area2) (loc box3 area1) (loc box4 area1) (loc box5 area1)
  (on box3 box4)
  (loc fan2 area12) (loc fan3 area2) (loc fan1 area1)
  (attached fan3 gears3)
  (on connector2 box1)
  (active gate4) (active gate5) (active gate6)

  ;static
  (adjacent area8 area9)
  (height box2 1) (height box3 1) (height box4 1) (height box5 1)
  (height rostrum2 0.5)
  (locale transmitter2 area8)
  (locale rostrum2 area10)
  (locale ladder3 area10) (locale ladder4 area10)
  (locale gears2 area10) (locale gears3 area2)
  (locale receiver7 area9) (locale receiver8 area10)
  (locale receiver9 area10)
  (locale receiver10 area10) (locale receiver11 area11)
  (locale receiver12 area11)
  (locale receiver13 area11) (locale receiver14 area2)
  (color transmitter2 red)
  (color receiver7 blue) (color receiver8 blue) (color receiver9 blue)
  (color receiver10 red) (color receiver11 red) (color receiver12 blue)
  (color receiver13 red) (color receiver14 red)
  (controls receiver7 gate4) (controls receiver8 gate4)
  (controls receiver9 gears2)
  (controls receiver10 gate5) (controls receiver11 gate5)
  (controls receiver12 gate6)
  (controls receiver13 gate6) (controls receiver14 gears3)
  (separates barrier3 area1 area8) (separates barrier4 area1 area11)
  (separates gate4 area9 area10) (separates gate5 area10 area11)
  (separates gate6 area11 area12)
  (climbable> ladder3 area10 area8) (climbable> ladder4 area10 area11)

;los is line-of-sight from an area to a fixed station
  (los0 area1 transmitter2) (los0 area2 transmitter2)
  (los0 area1 receiver7) (los0 area8 receiver7)
  (los0 area1 gate4) (los0 area8 gate4) (los1 area11 gate5 gate4)
  (los1 area11 gate5 receiver8)
  (los1 area9 gate4 receiver9) (los1 area11 gate5 receiver9)
  (los1 area9 gate4 receiver10)
  (los0 area1 gate5) (los1 area9 gate4 gate5) (los1 area12 gate6 gate5)
  (los0 area1 receiver11) (los1 area12 gate6 receiver11)
  (los2 area9 gate4 gate5 receiver12) (los1 area10 gate5 receiver12)
  (los1 area12 gate6 receiver12)
  (los2 area9 gate4 gate5 receiver13) (los1 area10 gate5 receiver13)
  (los1 area12 gate6 receiver13)
  (los1 area10 gate5 gate6)
  (los0 area1 receiver14) (los1 area10 gate5 receiver14)
  (los0 area11 receiver14)

;visibility is from an area to an area
;potentially containing a movable target or terminus
  (visible0 area1 area9) (visible0 area8 area2)
  (visible1 area1 gate4 area10)
  (visible1 area8 gate4 area10)
  (visible2 area9 gate4 gate5 area11)
  (visible1 area10 gate5 area1) (visible2 area10 gate5 gate6 area12)
)

;the following init-actions save tedious listing systematic facts

(define-init-action init-los0) ;los exists to a 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
    0
  (?divider divider (?area1 ?area2) area)
  (separates ?divider ?area1 ?area2)
  (?divider divider (?area1 ?area2) area)
  (assert (visible1 ?area1 ?divider ?area2)))


(define-init-action inactive-connectors
    0
  (?connector connector)
  (always-true)
  (?connector connector)
  (assert (inactive ?connector)))


(define-goal  ;always put this last
    (and (free me)
         (loc me area10)
         (loc jammer1 area1)
         (jamming jammer1 gate4)
         (inactive gate4)
         (loc connector3 area8)
         (connecting connector3 transmitter2)
         (color connector3 red)
         (active connector3)
         (loc connector4 area10)
         (connecting connector4 connector3)
         (connecting connector4 receiver10)
         (color connector4 red)
         (active connector4)
))
```

Crater Problem Solution:

```
* (bnb::solve)

New path to goal found at depth = 9
New path to goal found at depth = 9
New path to goal found at depth = 9

total states processed so far = 100,000
average branching factor = 5.1556463

total states processed so far = 200,000
average branching factor = 4.81544

total states processed so far = 300,000
average branching factor = 3.955561


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

```
Depth cutoff = 0

Total states processed = 315,339

Program cycles (states expanded) = 14,810

Average branching factor = 3.907626

Start state:
((ACTIVE GATE4) (ACTIVE GATE5) (ACTIVE GATE6) (ATTACHED FAN3 GEARS3)
 (COLOR RECEIVER10 RED) (COLOR RECEIVER11 RED) (COLOR RECEIVER12 BLUE)
 (COLOR RECEIVER13 RED) (COLOR RECEIVER14 RED) (COLOR RECEIVER7 BLUE)
 (COLOR RECEIVER8 BLUE) (COLOR RECEIVER9 BLUE) (COLOR TRANSMITTER2 RED)
 (FREE ME) (INACTIVE CONNECTOR2) (INACTIVE CONNECTOR3) (INACTIVE CONNECTOR4)
 (INACTIVE CONNECTOR5) (INACTIVE CONNECTOR6) (LOC BOX1 AREA2)
 (LOC BOX3 AREA1) (LOC BOX4 AREA1) (LOC BOX5 AREA1) (LOC CONNECTOR2 AREA2)
 (LOC CONNECTOR3 AREA8) (LOC CONNECTOR4 AREA10) (LOC CONNECTOR5 AREA11)
 (LOC CONNECTOR6 AREA11) (LOC FAN1 AREA1) (LOC FAN2 AREA12) (LOC FAN3 AREA2)
 (LOC JAMMER1 AREA1) (LOC ME AREA1) (ON BOX3 BOX4) (ON CONNECTOR2 BOX1))

Goal:
(AND (FREE ME) (LOC ME AREA10) (LOC JAMMER1 AREA1) (JAMMING JAMMER1 GATE4)
(INACTIVE GATE4) (LOC CONNECTOR3 AREA8) (CONNECTING CONNECTOR3 TRANSMITTER2)
(COLOR CONNECTOR3 RED) (ACTIVE CONNECTOR3) (LOC CONNECTOR4 AREA10)
(CONNECTING CONNECTOR4 CONNECTOR3) (CONNECTING CONNECTOR4 RECEIVER10)
(COLOR CONNECTOR4 RED) (ACTIVE CONNECTOR4))

Total solutions found = 3
(Check ww::*solutions* for list of all solutions.)

Number of steps in minimum path length solution = 9


Shortest solution path from start state to goal state:
(1 (PICKUP-JAMMER JAMMER1 AREA1 NIL))
(2 (JAM GATE4 JAMMER1 AREA1))
(3 (MOVE AREA1 AREA8))
(4 (PICKUP-CONNECTOR CONNECTOR3 AREA8 NIL))
(5 (CONNECT-TO-1-TERMINUS CONNECTOR3 TRANSMITTER2 RED AREA8))
(6 (MOVE AREA8 AREA9))
(7 (MOVE AREA9 AREA10))
(8 (PICKUP-CONNECTOR CONNECTOR4 AREA10 NIL))
(9 (CONNECT-TO-2-TERMINUS CONNECTOR4 RECEIVER10 RED CONNECTOR3 RED AREA10))

Final state:
((ACTIVE CONNECTOR3) (ACTIVE CONNECTOR4) (ACTIVE GATE5) (ACTIVE GATE6)
 (ATTACHED FAN3 GEARS3) (COLOR CONNECTOR3 RED) (COLOR CONNECTOR4 RED)
 (COLOR RECEIVER10 RED) (COLOR RECEIVER11 RED) (COLOR RECEIVER12 BLUE)
 (COLOR RECEIVER13 RED) (COLOR RECEIVER14 RED) (COLOR RECEIVER7 BLUE)
 (COLOR RECEIVER8 BLUE) (COLOR RECEIVER9 BLUE) (COLOR TRANSMITTER2 RED)
 (CONNECTING CONNECTOR3 CONNECTOR4) (CONNECTING CONNECTOR3 TRANSMITTER2)
 (CONNECTING CONNECTOR4 CONNECTOR3) (CONNECTING CONNECTOR4 RECEIVER10)
 (CONNECTING RECEIVER10 CONNECTOR4) (CONNECTING TRANSMITTER2 CONNECTOR3)
 (FREE ME) (INACTIVE CONNECTOR2) (INACTIVE CONNECTOR5) (INACTIVE CONNECTOR6)
 (INACTIVE GATE4) (JAMMING JAMMER1 GATE4) (LOC BOX1 AREA2) (LOC BOX3 AREA1)
 (LOC BOX4 AREA1) (LOC BOX5 AREA1) (LOC CONNECTOR2 AREA2)
 (LOC CONNECTOR3 AREA8) (LOC CONNECTOR4 AREA10) (LOC CONNECTOR5 AREA11)
 (LOC CONNECTOR6 AREA11) (LOC FAN1 AREA1) (LOC FAN2 AREA12) (LOC FAN3 AREA2)
 (LOC JAMMER1 AREA1) (LOC ME AREA10) (ON BOX3 BOX4) (ON CONNECTOR2 BOX1))

Shortest path solution is also a minimum duration solution

Evaluation took:
  24.153 seconds of real time
  24.109375 seconds of total run time (23.968750 user, 0.140625 system)
  [ Run times consist of 0.297 seconds GC time, and 23.813 seconds non-GC
time. ]
  99.82% CPU
  81,303,117,403 processor cycles
  5,804,444,400 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-derived-relations
    (already-placed* ?object)  (let ($r $c)
                                 (loc ?object $r $c))

    (in-same-row* ?object1 ?object2)  (let ($r1 $c1 $r2 $c2)
                                        (loc ?object1 $r1 $c1)
                                        (loc ?object2 $r2 $c2)
                                        (= $r1 $r2))

    (in-same-col* ?object1 ?object2)  (let ($r1 $c1 $r2 $c2)
                                        (loc ?object1 $r1 $c1)
                                        (loc ?object2 $r2 $c2)
                                        (= $c1 $c2))

    (in-col* ?object ?column)  (let ($r $c)
                                 (loc ?object $r $c)
                                 (= $c ?column))

    (vert-next-to* ?object1 ?object2)  (let ($r1 $c1 $r2 $c2)
                                         (loc ?object1 $r1 $c1)
                                         (loc ?object2 $r2 $c2)
                                         (and (= $c1 $c2)
                                              (or (= $r1 (1+ $r2))
                                                  (= $r1 (1- $r2)))))
    (diag-next-to* ?object1 ?object2)  (let ($r1 $c1 $r2 $c2)
                                         (loc ?object1 $r1 $c1)
                                         (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 ($row $col) fluent)
  (and (not (already-placed* ?object))
       (next-row $row)
       (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)  ;if next-row /= 4, no need to check constraints
         (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)

total states processed so far = 100,000
average branching factor = 2.392696

total states processed so far = 200,000
average branching factor = 2.392447

total states processed so far = 300,000
average branching factor = 2.3923373

total states processed so far = 400,000
average branching factor = 2.3922532

total states processed so far = 500,000
average branching factor = 2.3922565

New path to goal found at depth = 9
New path to goal found at depth = 9
New path to goal found at depth = 9
New path to goal found at depth = 9
New path to goal found at depth = 9
New path to goal found at depth = 9
New path to goal found at depth = 9
New path to goal found at depth = 9
New path to goal found at depth = 9
New path to goal found at depth = 9
New path to goal found at depth = 9
New path to goal found at depth = 9

total states processed so far = 600,000
average branching factor = 2.3922727

New path to goal found at depth = 9
New path to goal found at depth = 9
New path to goal found at depth = 9
New path to goal found at depth = 9
New path to goal found at depth = 9
New path to goal found at depth = 9
New path to goal found at depth = 9
New path to goal found at depth = 9
New path to goal found at depth = 9
New path to goal found at depth = 9
New path to goal found at depth = 9
New path to goal found at depth = 9

total states processed so far = 700,000
average branching factor = 2.392159

total states processed so far = 800,000
average branching factor = 2.3920915
```

```
total states processed so far = 900,000
average branching factor = 2.3920126


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

Depth cutoff = 9

Total states processed = 986,386

Program cycles (states expanded) = 260,651

Average branching factor = 2.3919902

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 = 24
(Check ww::*solutions* for list of all solutions.)

Number of steps in minimum path length solution = 9

Shortest solution path from start state to goal state:
(1 (PUT GUARD2 1 1))
(2 (PUT GRASS3 1 2))
(3 (PUT CREW2 1 3))
(4 (PUT GRASS2 2 1))
(5 (PUT GRASS1 2 2))
(6 (PUT JOHN 2 3))
(7 (PUT WASP 3 1))
(8 (PUT GUARD1 3 2))
(9 (PUT CREW1 3 3))
```

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

Shortest path solution is also a minimum duration solution

Evaluation took:
  6.521 seconds of real time
  6.484375 seconds of total run time (6.437500 user, 0.046875 system)
  [ Run times consist of 0.125 seconds GC time, and 6.360 seconds non-GC
time. ]
  99.43% CPU
  21,951,574,738 processor cycles
  3,510,910,928 bytes consed
```