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 an idealized 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.

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 exhaustive. 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 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 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, action schedules)
- Global constraint specification, independent of action preconditions
- Derived relations for simplifying action preconditions

- Function specification for on-the-fly, possibly recursive, computations
- Inclusion of arbitrary Lisp code in action rules, derived relations, constraints, and functions
- User control over search depth
- Generation of shortest plan found, plus other possible plans
- 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 blocks-problem.lisp in the Wouldwork\src\ directory.
3) Review the planning results in the command window.

If you would like to see solutions to the other sample problems, run the other executables boxes.exe (boxes-problem.lisp), jugs.exe (jugs-problem.lisp), and sentry.exe (sentry-problem.lisp). These problems are discussed in the Appendix.

If you also 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’. If you wish to set additional program parameters, switch to the planner package by executing (in-package :pl). 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 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, but at the expense of being able to represent some more complex planning scenarios falling outside the scope of the 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 Object 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 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 (see below).

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

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 (?b) on a support (?s), the precondition must specify that there is not another block on top of ?b, and in addition, that there is not another block on top of ?s. Note that 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 predicates, like cleartop>, which can be used to simplify action conditions.) And 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 ?block is put on some ?support), is that ?block will now be on ?support; and also that ?block will no longer be on what was previously supporting it. This effect can be concisely expressed as:

```
(and (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 0 means instantaneous), 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
    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))))))
```

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.

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>, <specified planning function>, <lisp function>. 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), <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 assertions, 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 scientific 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 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.

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, or merely the first 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 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.

Problem Specification Parameters

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

depth-cutoff 20
specifies the max search depth; negative or 0 means no cutoff, which may search forever

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 (ie, 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 10000000
specifies the estimated total maximum number of unique states to be explored during search; if this number is exceeded, hash table resizing will 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

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

Depth cutoff = 2

Total states processed = 25
Unique states encountered = 7

Program cycles = 1

Average branching factor = 7.0

Total solutions found = 1
(Check ww::*solutions* for best solution to each distinct goal.)

Shallowest solution depth = 2

Shortest solution path length from start state to goal state:
(0 (PUT B C))
(0 (PUT A B))

Evaluation took:
  0.125 seconds of real time
  0.109375 seconds of total run time (0.062500 user, 0.046875 system)
  [ Run times consist of 0.046 seconds GC time, and 0.064 seconds non-GC time. ]
  87.20% CPU
  438,101,052 processor cycles
  454,746,992 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 whether the program finished normally, and the depth cutoff (ie, consideration of all possible plans less than or equal to the cutoff in length). The total number of states encountered during planning is also listed, of which only a smaller number of unique states were actually analyzed. The number of program cycles reports how many sweeps through all possible actions from each unique state there were, in this case only one, since the goal check does not count as a complete sweep. 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 complete 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 take time to complete—they are instantaneous in this case. 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. 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.

Object Relations

Every relevant relation (and property) of *all* objects must be specified under ‘define-base-relations’. To avoid errors of omission, it is usually worthwhile to review the completed problem specification (actions, initial & goal conditions, etc) for missing relations.

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.

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 will automatically check both. Whenever one proposition like (separates gate1 area1 area2) becomes true or false in a state, the reciprocal proposition (separates gate1 area2 area1) also becomes true or false.

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 do not appear as explicit propositions in a state. But they are implicitly derived, when needed during analysis, from state propositions instantiating the base relations. 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-relation
  (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 angle bracket (>) attached to the name is simply a convention distinguishing derived relations from other base relations in action rules. Note also that the new relation’s arguments can appear as free variables in the expansion, but that all other variables in the expansion must be bound. Other derived predicates may appear in the expansion, making it convenient to express complex relations in terms familiar to the problem domain. Substituting cleartop> into the action rule 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).

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) will 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 include a statement like (not (= ?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.

Numerical Fluents

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 a other problems it is useful to allow continuous variables as well, perhaps representing the height and weight of a block. Accordingly, 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. 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.

As a more complete example, consider the specification of the classic "jugs" problem. Two jugs of different size will be used to measure out a specific amount of water taken from a large reservoir. There are no markings on the jugs to indicate 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). The action of filling a jug with water from the reservoir then takes the form of the following rule:

```
(define-action fill
    0
    (?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 $cap to represent the current amount in a jug and its capacity, respectively. The fluent prefix ($) is required. 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 appear in the expansion of the derived relation with the exact same name, since the expansion will simply be embedded in the rule.

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). 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 actions. 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). 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 real values in action rules, it may also be useful to specify functions for calculating with those fluents. Arbitrary Common-Lisp functions that evaluate to ‘t‘ or ‘nil’ may appear along with other expressions in an action, constraint, or derived relation. For example, an expression in a precondition expression like (< $height1 $height2) would be 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 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 (elevation ?support $e) appears in the precondition of an action, where ?support is already instantiated, then the resulting value of $e after evaluation will be the elevation of that support. This value of $e might then be used subsequently in an expression like (< $e 10) to check whether the total elevation of that support is less than 10. The function that computes the elevation of a support is as follows:

```
(define-function elevation (state ?support)
  (and (height ?support $h)
       (exists (?s support)
         (if (on ?support ?s)
             (return-from elevation
               (+ $h (elevation state ?s)))
           (return-from elevation $h)))))
```

This function first gets the height ($h) of the support (?support). Next, if there is some other support ?s, which ?support is on, then return $h plus the elevation of ?s as the net elevation of ?support. Otherwise, just return $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. Note that the first argument to any user-defined function must be the keywork 'state', which will provide the current context for the analysis.

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, 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.

In this problem the agent must avoid contact with the sentry, but can pass through a patrolled area as long as the sentry is in a different area. Exogenous events often place global constraints on the planning agent, which can be added to the problem specification:

```
(define-constraint
  (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 (me) and the sentry are both located in that area, and the sentry is not disabled (a kill situation). If the constraint evaluates to true 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 not 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. Another optional keyword is :rebound, which can be used to reverse the programmed happening events mid-stream if some situation occurs (eg, if the sentry's path becomes blocked at some point, at which time the sentry reverses direction and continues on). Adding these specifications to the happenings definition then gives:

```
(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))
  :rebound (exists (?o obstacle ?a area)
             (and (loc sentry1 ?a)
                  (loc ?o ?a))))
```

Waiting

When there are exogenous events happening in the planning environment, it may sometimes become expedient to simply wait for the situation to change. For example, in the patrolling sentry problem discussed in the Appendix, the planning agent needs to wait for a time interval before moving, 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)
  ()
  (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. (Note that it must be 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.
Also, whenever a wait action is included in a final plan, it means there is no longer a guarantee that the plan is the shortest possible plan. This limitation occurs because of the way wait is implemented in the planner. Use of a different algorithm might restore optimality.

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.

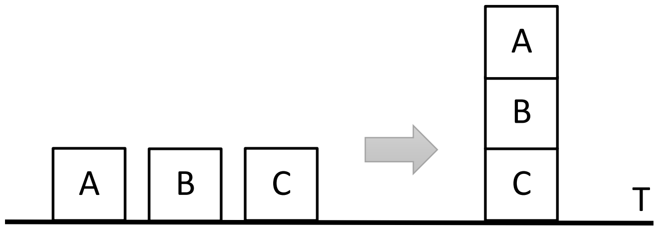
To assist with debugging individual actions, constraints, derived relations, functions, etc, you can also insert arbitrary lisp code among logic expressions—for example to print variable bindings as they are assigned during execution. The following action from the blocks world problem will print the bindings for three 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)
                         (ut::prt ?b)
                         (on ?b ?support))))))
   (?block block ?support support)
   (and (on ?block ?support)
        (exists (?s support)
          (if (on ?block ?s))
            (not (on ?block ?s)))))
```

For some problems with lots of objects and relations, each new state produced by the planner at each step may be large. Since debugging typically prints out each state, it may be useful to limit the display to only those objects and relations of interest. The user specifies which relations to display by including a statement like (define-monitored-relations on height …) in the problem specification. Only those relations (and the objects instantiating them) will then show in a state display—in this case ‘on’ and ‘height’. The program, however, still analyzes the complete state when making decisions. This feature is pehaps most useful for monitoring dynamic relations (eg, what block is on what at some time) rather than static unchanging relations (eg, the height of a block).

APPENDIX: SAMPLE PROBLEMS

1. Blocks World Problem



<u>Blocks Problem Specification</u>:

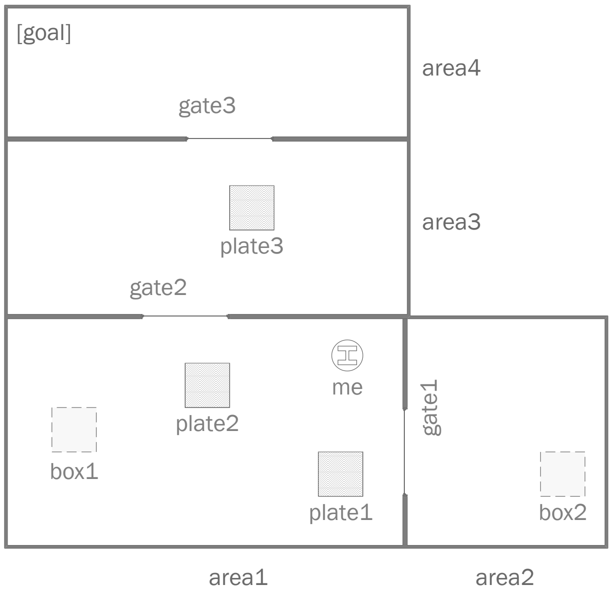
```
;;;; Filename: blocks-problem.lisp

;;; 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

(setq *depth-cutoff* 2)  ;max # of steps to goal

(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

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

Depth cutoff = 2
Total states processed = 25

Unique states encountered = 7

Program cycles = 1

Average branching factor = 7.0

Total solutions found = 1
(Check ww::*solutions* for best solution to each distinct goal.)

Shallowest solution depth = 2

Shortest solution path length from start state to goal state:
(0 (PUT B C))
(0 (PUT A B))

Evaluation took:
  0.159 seconds of real time
  0.140625 seconds of total run time (0.078125 user, 0.062500 system)
  [ Run times consist of 0.078 seconds GC time, and 0.063 seconds non-GC time. ]
  88.68% CPU
  537,624,242 processor cycles
  454,701,872 bytes consed
```

2. Boxes Problem



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

Boxes Problem Specification:

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

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


(in-package :pl)  ;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-base-relations
  (holding myself box)
  (loc (either myself box plate) area)
  (on box plate)
  (controls plate gate)
  (separates gate area area))

(define-monitored-relations
  holding loc on)

(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)
  (and (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 ?plate plate ?area area)
  (if (on ?box ?plate)
      (and (not (on ?box ?plate))
           (not (loc ?box ?area))
           (holding me ?box))
      (if (not (exists (?p plate)
                 (on ?box ?p)))
          (and (not (loc ?box ?area))
               (holding me ?box)))))

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

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

Depth cutoff = 10

Total states processed = 49

Unique states encountered = 37
Program cycles = 21

Average branching factor = 1.9047619

Total solutions found = 1
(Check ww::*solutions* for best solution to each distinct goal.)

Shallowest solution depth = 10
```

```
Shortest solution path length from start state to goal state:
(1 (PICKUP BOX1 NIL AREA1))
(2 (DROP BOX1 PLATE1 AREA1))
(3 (MOVE AREA1 AREA2))
(4 (PICKUP BOX2 NIL AREA2))
(5 (MOVE AREA2 AREA1))
(6 (DROP BOX2 PLATE2 AREA1))
(7 (PICKUP BOX1 PLATE1 AREA1))
(8 (MOVE AREA1 AREA3))
(9 (DROP BOX1 PLATE3 AREA3))
(10 (MOVE AREA3 AREA4))


Evaluation took:
  0.139 seconds of real time
  0.109375 seconds of total run time (0.062500 user, 0.046875 system)
  [ Run times consist of 0.047 seconds GC time, and 0.063 seconds non-GC time. ]
  78.42% CPU
  467,500,956 processor cycles
  455,359,648 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.

<u>Jugs Problem Specification:</u>

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

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

(in-package :pl)  ;required

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

(define-types
    jug (jug1 jug2))

(define-base-relations
    (contents jug !real)
    (capacity jug !real))

(define-monitored-relations
    contents capacity)

(define-action fill
    0
  (?jug jug ($amt $cap) fluent)
  (and (contents ?jug $amt)
       (capacity ?jug $cap)
       (< $amt $cap))
  (?jug jug $cap fluent)
  (contents ?jug $cap))
```

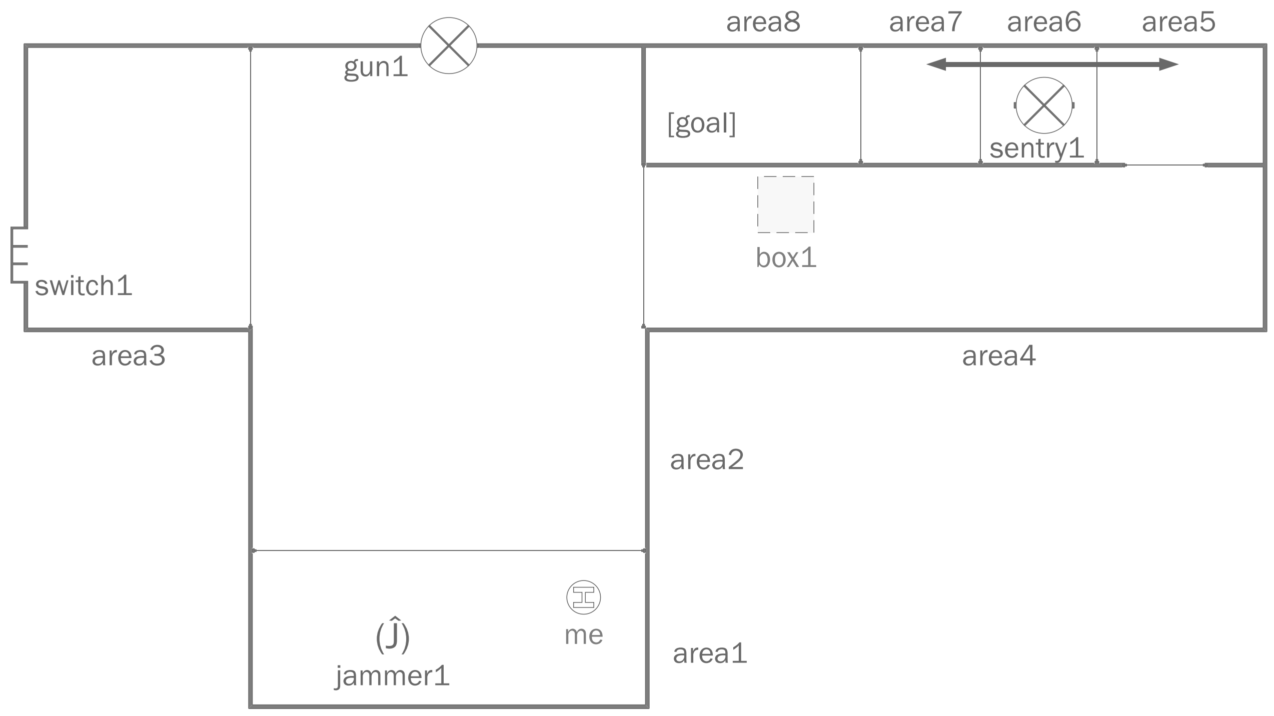
```
(define-action empty
    0
  (?jug jug $amt fluent)
  (and (contents ?jug $amt)
       (> $amt 0))
  (?jug jug)
  (contents ?jug 0))

(define-action pour
    0
  ((?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)
  (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)))
```

<u>Jugs Problem Solution:</u>

```
* (bnb::solve)

New path to goal found at depth = 6

New path to goal found at depth = 4

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 = 7

Average branching factor = 2.5714285

Total solutions found = 2
(Check ww::*solutions* for best solution to each distinct goal.)

Shallowest solution depth = 4

Shortest solution path length from start state to goal state:
(0 (FILL JUG2))
(0 (POUR JUG2 JUG1))
(0 (EMPTY JUG1))
(0 (POUR JUG2 JUG1))

Evaluation took:
  0.167 seconds of real time
  0.140625 seconds of total run time (0.078125 user, 0.062500 system)
  [ Run times consist of 0.079 seconds GC time, and 0.062 seconds non-GC time. ]
  84.43% CPU
  561,504,960 processor cycles
  454,704,832 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 patrols area5, area6, area7.

<u>Sentry Problem Specification:</u>

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

;;; Problem specification for getting by an automated sentry by jamming it.
;;; See sentry-problem in user manual appendix.

(in-package :pl)  ;required

(setq *depth-cutoff* 16)

(define-types
  myself     (me)
  box        (box1)
  jammer     (jammer1)
  gun        (gun1)
  sentry     (sentry1)
  switch     (switch1)
  red       ()
  green     ()
  area      (area1 area2 area3 area4 area5 area6 area7 area8)
  cargo     (either  jammer) ;box
  threat    (either gun sentry)
  target    (either threat))

(define-base-relations
  ;dynamic
  (holding myself cargo)
  (loc (either myself cargo threat) area)
  (red switch)
  (green switch)
  (jamming jammer target)
  (waiting)
  ;static
  (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-monitored-relations
  holding loc jamming)

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

  (passable> ?area1 ?area2)  (adjacent ?area1 ?area2)

  (safe> ?area)              (forall (?g gun)
                               (if (watches ?g ?area)
                                 (disabled> ?g)))

  (disabled> ?threat)        (or (exists (?j jammer)
                                   (jamming ?j ?threat))
                                 (exists (?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
  :rebound
    (exists (?c cargo ?a area)
      (and (loc sentry1 ?a)
           (loc ?c ?a)))
  :interrupt
    (exists (?j jammer)
      (jamming ?j sentry1)))
```

```
(define-constraint
  ;Constraints only needed for happening events.
  ;Global constraints included here. Return nil if constraint violated.
  (exists (?s sentry ?a area)
    (and (loc me ?a)
         (loc ?s ?a)
         (not (disabled> ?s)))))

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

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

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

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

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

<u>Sentry Problem Solution</u>:

```
* (bnb::solve)

New path to goal found at depth = 16

New path to goal found at depth = 16

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

Depth cutoff = 16

Total states processed = 664

Unique states encountered = 245

Program cycles = 142

Average branching factor = 2.619718

Total solutions found = 2
(Check ww::*solutions* for best solution to each distinct goal.)

Shallowest solution depth = 16

Shortest solution path length from start state to goal state:
(1 (PICKUP JAMMER1 AREA1))
(2 (JAM GUN1 JAMMER1 AREA1))
(3 (MOVE AREA1 AREA2))
(4 (MOVE AREA2 AREA3))
(5 (THROW-SWITCH SWITCH1))
(6 (MOVE AREA3 AREA2))
(7 (MOVE AREA2 AREA1))
(8 (PICKUP JAMMER1 AREA1))
(9 (MOVE AREA1 AREA2))
```

```
(10 (MOVE AREA2 AREA4))
(11 (WAIT 1))
(12 (MOVE AREA4 AREA5))
(13 (JAM SENTRY1 JAMMER1 AREA5))
(14 (MOVE AREA5 AREA6))
(15 (MOVE AREA6 AREA7))
(16 (MOVE AREA7 AREA8))

Evaluation took:
  0.284 seconds of real time
  0.250000 seconds of total run time (0.156250 user, 0.093750 system)
  [ Run times consist of 0.140 seconds GC time, and 0.110 seconds non-GC time. ]
  88.03% CPU
  955,324,724 processor cycles
  486,006,128 bytes consed
```

5. 4-Queens Problem

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

Place 4 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: 4queens-problem.lisp

;;; Problem specification for 4-queens.


(in-package :ww)  ;required


(setq *depth-cutoff* 4)


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


(define-base-relations
  (loc queen !fixnum column)
  (placed queen)
  (next-row !fixnum))


(define-monitored-relations
  loc next-row)  ;but if loc then placed
```

```
(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)
  (and (loc ?queen $row ?column)
       (placed ?queen)
       (next-row (1+ $row))))


(define-init
    (next-row 1))


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

4-Queens Problem Solution:

There are exactly 48 unique solutions to the 4-queens problem taking into account all possible successful arrangements of four distinct queens labeled queen1, queen2, queen3, queen4. Considering only the arrangements of 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 3) means put queen4 in the 3rd column of row 1. For large versions of this problem, such as with eight queens, setting *tree-or-graph* = ‘tree, and *first-solution-sufficient* = t, should provide a solution within a couple of minutes.

```
* (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

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

Depth cutoff = 4

Total states processed = 185

Unique states encountered = 232

Program cycles = 65

Average branching factor = 2.8307693

Total solutions found = 48
(Check ww::*solutions* for shortest solution to each distinct goal.)

Shallowest solution depth = 4

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

Evaluation took:
  0.237 seconds of real time
  0.140625 seconds of total run time (0.078125 user, 0.062500 system)
  [ Run times consist of 0.062 seconds GC time, and 0.079 seconds non-GC time. ]
  59.49% CPU
  800,440,737 processor cycles
  456,376,576 bytes consed
```