GPT: A Tool for Planning with Uncertainty and Partial Information

Blai Bonet
Departamento de Computación
Universidad Simon Bolívar
Caracas, Venezuela
bonet@ldc.usb.ve

Héctor Geffner
Departamento de Computación
Universidad Simon Bolívar
Caracas, Venezuela
hector@ldc.usb.ve

1 Introduction

We describe the GPT system and its utilization over a number of examples. GPT (General Planning Tool) is an *integrated* software tool for modeling, analyzing and solving a wide range of planning problems dealing with uncertainty and partial information, that has been used for us and others for research and teaching. Our approach is based on different state models that can handle various types of *action dynamics* (deterministic and probabilistic) and *sensor feedback* (null, partial, and complete). The system consists mainly of a *high-level language* for expressing actions, sensors, and goals, and a bundle algorithms based on heuristic search for solving them. The language is one of GPT's strengths since it presents the user a consistent and unified framework for the planning task. These descriptions are then solved by appropriate algorithms chosen from the bundle. The output for all models is a plan that is *ready* for execution and analysis; moreover, for a class of problems described below, it is an *optimal* plan.

The rest of the paper is organized as follows. In next two sections, we describe the mathematical models that formalize the tasks GPT is able to solve, and the algorithms used to solve them. In Sect. 4, we show GPT's language with an example and give an overview of its syntax. At the end, Sect. 6 shows results over selected examples from different classes of problems; it contains actual input and output to GPT as codification of problems in the language and the resulting plans generated by GPT.

2 State Models

We briefly review the mathematical models that make various forms of planning precise according to the type of action dynamics and sensor feedback. In each case, the models determine what the planning task is and what is the form of the solution. A formal and through-full exposition can be found in [5].

State models are the most basic models in AI and consist of a finite set of states S, a finite set of actions A, and a state transition function that describe how actions map states into successor states. They are used to represent the world when building a *controller* whose job is to choses actions such that certain goals can be achieved. There are two types of controllers: open-loop in which a control plan or policy is chosen off-line and then applied to the world without considering possible feedback, and closed-loop, a more robust one, in which the controller choses actions based on the sequence of past observations.

The planning problem can then be cast as a control problem in a state space where the task is to take actions that lead an initial state into a set of goal states. Systems with complex dynamics can behave stochastically when operated by a given controller; that is, the sequence of visited states may vary for different executions started from a common initial state. Each such sequence has an associated cost and probability; the cost given by the cost of the actions in the sequence and the probability determined by the dynamics. The costs and probabilities are then combined to define a unique cost measure associated with the controller, so that different controller can be compared by comparing their costs. Thus, we say that a controller (that we also called a plan or policy) is optimal if it is of minimum expected cost.

We'll consider the different classes of problems that result from combining different dynamics and sensor feedback. For system dynamics, we consider three possibilities

- **Deterministic:** in this case the dynamics is represented by a deterministic state-transition function $next(\cdot, \cdot)$ so that $next(s, a)$ stands for the *unique* next state that result from doing action a in state s, for $a \in A(s)$ (the set of applicable actions in state s).
- **Non-Deterministic:** in these problems, actions may result in more than one successor state when applied to certain states. They are represented by a *set* function $next(\cdot, \cdot) \subseteq S$ that maps a state s and action a into the set of possible next states $next(s, a)$.
- **Probabilistic:** these are models with non-deterministic dynamics extended with probability distributions $P(\cdot; s, a)$ over $next(s, a)$ for all s and a. They are used when a more precise description of the dynamics is needed.

Although, in principle, a non-deterministic model is equivalent to a probabilistic one in which all distributions are uniform, we introduce a distinction that makes them different. Namely, that the user in the non-deterministic case is interested in minimizing the cost of the *worst-possible* trajectory no matter how probable it is. Thus, both models are different since it is not hard to build examples in which the optimal policies differ for the cases mentioned before.

The feedback is what the controller gets from the environment after applying the actions. It is described by observations $o(s, a)$ that are obtained when then *real* state produced

		Transition Function		
		Deterministic	Non-deterministic	Probabilistic
Feedback	Complete	Classical[a]	Non-deterministic	Probabilistic
	Partial	Contingent	Contingent	Prob. Contingent
	Null	Conformant	Conformant	Prob. Contingent

[a]Closed-loop controller.

Table 1: The different planning tasks along three different dimensions: feedback, transition function, and initial states.

by the action is s. The different possibilities for the sensor model are

- **Complete:** the controller has full information about the state of the system at any moment. They are characterized by requiring $o(s,a) = o(s',a)$ if and only if $s = s'$; e.g., $o(s,a) = s$.
- **Partial:** in this case, the observations don't identify states *uniquely* since different states may incur in the same observation. However, after obtaining $o(s,a)$, the controller knows that the true state is not s' for $o(s',a) \neq o(s,a)$.
- **Null:** this extreme case of partial sensing corresponds to a blind controller, and is characterized by letting $o(s,a) = o(s',a)$ for all s, s'.

Another dimension to consider is given by the number of possible initial states of the system. On one hand, as in classical planning, is to assume the system always starts at one and the same initial state. The other possibility is to allow for multiple possible initial states of the system, that is often used to model systems with a unique but unknown initial state. These three features, dynamics, information, and number of initial situations, are combine to define the classes of planning problems shown in Table 1. Each such class is a special case of probabilistic contingent planning, yet better algorithms can be given when they are considered separately. We study such algorithms next.

3 Algorithms

The two main algorithms GPT uses for solving planning problems are optimal heuristic search A* and a version of Dynamic Programming called RTDP. The selection depends in the class of the problem: A* for conformant planning, and RTDP for non-deterministic, probabilistic, contingent and probabilistic contingent planning.[1] In all cases the output is a plan that takes the controller to a goal with probability 1. Moreover, except for probabilistic contingent planning, GPT returns an optimal policy for the input task.

3.1 A*

The model for conformant planning can be solved by any standard search algorithm. GPT uses the A* algorithm with a domain-independent heuristic derived from the problem by a general transformation. Basically, we compute the optimal cost function V^*_{dp} over the states of a 'relaxed' problem where *full state observability* is assumed (see [5] for details). That function is then used to compute another heuristic function that is plugged into the search algorithm. It is simple to show the resulting heuristic is admissible and hence the solutions found by A* are guaranteed to be optimal.

3.2 Real Time Dynamic Programming (RTDP)

RTDP is a generalization of Korf's LRTA* [8] due to Barto et al. [1]. It is a version of dynamic programming that finds the optimal policy function for the set of *relevant* states; i.e., those states that may be visited with positive probability by an optimal policy. This partial solution is enough for solving the problem optimally when started in one of the initial states. RTDP can be understood as a modification of *greedy search* in which the heuristic function is updated every time an action is taken (see [1; 5]). RTDP solves the problem by successive trials each one starting at an initial state and ending in a goal (or after a predefined number of moves).[2] A main problem with RTDP is that it doesn't specify when to stop the trials so to guarantee the result is an optimal policy. We have developed a *stopping rule*, parametrized in a small $\epsilon \geq 0$, that solves this problem; i.e., a rule that stops the algorithm and guarantees an optimal policy when ϵ is sufficiently small (see Appendix for a brief description of the stopping rule).

4 Language

Above control problems are useful *models* for making explicit the mathematical structure of a wide class of planning problems. Often, however, they are not good *languages* for describing them. This is due to the number and size of the relations and parameters involved. In AI, it has been common to describe planning problems compactly in terms of *modular* and *high-level* languages such as STRIPS. In recent years similar languages have been defined for describing probabilistic actions and general POMDPs (see [6] and references therein). We illustrate the latter with a problem of planning and incomplete information from [9].

4.1 Example

The problem involves an agent that has a large supply of eggs and whose goal is to get three good eggs and no bad ones into one of two bowls. The eggs can be either good or bad, and at any time the agent can find out whether a bowl contains a bad egg by inspecting the bowl. In [3] this problem is encoded by expressions such as the ones in Fig.1 which are compiled into a probabilistic contingent planning problem and solved by the RTDP algorithm.

[1]The case of classical planning is considered in a separate work in [4].

[2]For probabilistic contingent problems in which the state space is infinite, a version of RTDP that uses a discretization of belief states is needed [3; 5]. That version is called RTDP-BEL.

```
(define (domain omelette)
  (:model (:dynamics :probabilistic) (:feedback :partial))
  (:types BOWL)
  (:functions (ngood BOWL :integer[0,3])
              (nbad BOWL :integer[0,3])
              (number BOWL :integer[0,3]))
  (:objects  good holding - :boolean)

  (:axiom set_number
    :parameters ?b - BOWL
    :effect (:set (number ?b) (+ (ngood ?b) (nbad ?b))))

  (:action grab
    :precondition (= holding false)
    :effect (:probabilistic (0.5 (:set good true)
                                 (:set holding true))
                            (0.5 (:set good false)
                                 (:set holding true))))

  (:action break_egg
    :parameters ?b - BOWL
    :precondition (:and (< (number ?b) 3)
                        (= holding true))
    :effect (:when (= good true)
               (:set holding false)
               (:set (ngood ?b) (+ (ngood ?b) 1)))
            (:when (= good false)
               (:set holding false)
               (:set (nbad ?b) (+ (nbad ?b) 1))))

  (:action pour
    :parameters ?b1 ?b2 - BOWL
    :precondition (:and (:not (= ?b1 ?b2))
                        (= holding false)
                        (<= (+ (number ?b1) (number ?b2)) 3))
    :effect (:set (ngood ?b2) (+ (ngood ?b2) (ngood ?b1)))
            (:set (nbad ?b2) (+ (nbad ?b2) (nbad ?b1)))
            (:set (ngood ?b1) 0)
            (:set (nbad ?b1) 0))

  (:action clean
    :parameters ?b - BOWL
    :precondition (= holding false)
    :effect (:set (ngood ?b) 0)
            (:set (nbad ?b) 0))

  (:action inspect
    :parameters ?b - BOWL
    :precondition (= holding false)
    :observation (= (nbad ?b) 0)))

(define (problem eggs)
  (:domain omelette)
  (:objects small large - BOWL)
  (:init (:set (ngood small) 0)
         (:set (nbad small) 0)
         (:set (ngood large) 0)
         (:set (nbad large) 0)
         (:set holding false)
         (:set good false))
  (:goal (:and (= (ngood large) 3)
               (= (nbad large) 0))))
```

Figure 1: Full description of the Omelette problem.

The language illustrated in Fig.1 extends STRIPS in several ways: states are not associated with a set of atoms but with assignments to arbitrary fluents; probabilities, costs and primitive operations like '+' are included, and an `:observation` section may be used in the actions to indicate observability. The fluents in this problem are the number of a good and bad eggs in each bowl (`ngood` and `nbad`), and the boolean variables `holding` and `good` that represent whether the agent is holding an egg and whether such egg is good or bad. The fluent `holding` is always observable, but the value of the expression '`nbad(bowl)` > 0' is observed after doing the action `inspect(bowl)` only. We provide the main ideas of the language next.

4.2 Language and States

The *language* is a typed logical language that involves a number of constant, function, and predicate symbols from which atoms, terms, and formulas are defined in the standard way. For example, '`ngood(bowl)` + `nbad(bowl)` ≤ 3' is a formula expressing that the total number of eggs in `bowl` is less than or equal to 3.

Given a language with the relevant type and object declarations, the *states* are the *logical interpretations* over such language. That is, a state s assigns a denotation x^s to any symbol x from which the denotations of all atoms, terms, and formulas are obtained following the standard composition rules. Symbols like '≤ 3', and others, have a denotation that is fixed and is independent of the state. States thus have to assign a denotation to *fluent* symbols only; symbols like `ngood`, `nbad`, etc. *Type* and *object* declarations for these symbols define the possible set of denotations (values) and all together implicitly define the *state space*. Action *preconditions* define the set $A(s)$ of actions applicable in each state s (the actions whose preconditions have a true denotation in s) and action *effects* define the state *transition functions* or transition *probabilities*.

Assuming that the cost of all actions is 1, such a language can be used to define state models for probabilistic and conformant planning problems. For describing the other models, it is necessary to describe also what is *observable*. That's the role of the `:observation` section like the one in action `inspect(bowl)` in Fig.1. Not only deterministic observations are supported but also probabilistic ones that are encoded by syntax similar to the one for probabilistic effects (e.g., in the `grab` action) that define the sensor model $P(o; s, a)$. An action a that makes the expression x observable for a term or formula x produces observations $o\!:\!(x = v)$ for each possible denotation v of x with different probabilities that depend on the belief state where the action was done and the sensor model [5].

The language also provides facilities for expressing deterministic and probabilistic *ramification* rules. While action rules express the value of a variable in terms of the value of the variables at the *previous* time, ramification rules express the value of a variable as a function of variables *at the same time point*. Ramification rules are useful in a number of circumstances as when the user need to specify indirect effect of actions and domain constraints. In Fig.1 for example, we use a ramification to keep track of the total number of eggs in each bowl by means of the rule `number(bowl)` $=$ `ngood(bowl)` + `nbad(bowl)`.

4.3 Overview of Syntax

GPT's language has a close syntax to the PDDL language for STRIPS planning [10]. Since the first planning competition in AIPS-98, PDDL has become the de-facto standard for STRIPS planning; all major STRIPS planner read PDDL input, old a new problems are distributed in PDDL format, etc. As in PDDL, a problem description is made from two sections. The first section, called domain definition, is a "general" description of the domain of the problem and contains definitions for types, functions, actions and ramification rules. The second section, called problem definition and always associated with a domain definition, is a description of an *instance* of the domain and contains definitions for typed objects and the initial and goal situations.

Each section needs to be a *complete compilation unit*; that is, every referenced symbol has to be lexically bounded. In Fig. 1, for example, the domain definition uses fluents `good` and `holding` so they must be defined in it. The fluents `small` and `large`, on the other hand, are only used in the

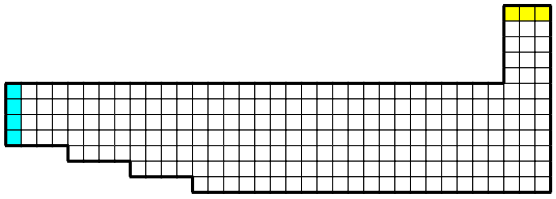

Figure 2: Racetrack grid. The left dark squares are starting position and the top right ones the goal line.

instance definition so they don't need to be defined in the domain. In this example, the separation between domain and problem allows the user to easily define variations of the problem as when there are more than two bowls and different initial or goal situations.

A typical domain definition contains: a keyword specifying the class of the problem ('`:model`'), declarations for types and symbols used in the unit's logical language ('`:types`', '`:functions`', and '`:objects`'), a (possible empty) list of ramification rules ('`:axiom`'), and one or more actions ('`:action`'). Additionally, it may contain information about which symbols will be denoted by external functions (see Racetrack below). A typical problem definition, on the other hand, contains a pointer to a domain definition ('`:domain`'), declarations of extra symbols, and the initial and goal situations ('`:init`' and '`:goal`').

5 General Planning Tool

The GPT system integrates above models and algorithms into a single tool for problem modeling, solving and analysis. It is is written in C++ and consists about 20k lines of codes and 100 files. The utilization of GPT consists of a *design* $\rightarrow$ *compile* $\rightarrow$ *solve* $\rightarrow$ *analyze* cycle. The first step refers to the modeling of the problem in GPT's language. This description is then compiled into *machine code* by a parser, that translate it into C++, and by `g++` that generates the object code. This phase is designed to offer the user the possibility to link *external* functions into the description of the problem and to provide alternate heuristic functions. In the solve phase, GPT calls the appropriate solver depending in the input problem. The output can then be analyzed qualitatively and quantitatively. The former by inspection of the resulting policy and the latter by analysis of 'extra' data; i.e., data generated by user request as the expected cost of solution, last value function, etc. The first three phases take place inside a shell-like environment under user commands. The output consist of a standard graph representation of the policy plus possibly few files with the quantitative information.[3]

6 Examples

In this section we show selected examples of actual GPT's input and output for different planning problem classes. We also show a transcript of GPT use for the Omelette problem.

[3] A formal definition of the policy graph is in [5]. Some intuitive examples are in Sect. 6.

```
(define (domain racetrack)
  (:model (:dynamics :probabilistic) (:feedback :complete))
  (:types STATE)
  (:functions (x  STATE :integer[0,34])
              (y  STATE :integer[0,11])
              (dx STATE :integer[-5,5])
              (dy STATE :integer[-5,5])
              (valid STATE :boolean))
  (:external valid)
  (:objects state lastState - STATE)

  (:axiom constraints
    :effect (:when (= (valid state) false)
               (:set state lastState)
               (:set (dx state) 0)
               (:set (dy state) 0)))

  (:action control
    :parameters ?ax ?ay - :integer[-1,1]
    :effect
     (:probabilistic
       (0.9 (:set lastState state)
            (:set (x state) (+ (x state) (+ (dx state) ?ax)))
            (:set (y state) (+ (y state) (+ (dy state) ?ay)))
            (:set (dx state) (+ (dx state) ?ax))
            (:set (dy state) (+ (dy state) ?ay)))
       (0.1 (:set lastState state)
            (:set (x state) (+ (x state) (dx state)))
            (:set (y state) (+ (y state) (dy state)))))))

(define (problem race)
  (:domain racetrack)
  (:init (:set (dx state) 0)
         (:set (dy state) 0)
         (:set (x state) 0)
         (:set (y state) :in { 5 6 7 8 })
         (:set lastState state))
  (:goal (:and (= (y state) 0)
               (:in (x state) { 32 33 34 }))))
```

Figure 3: Full description of the Racetrack problem.

6.1 Probabilistic Planning

Racetrack. The problem is to drive a car starting in a set of possible initial states to a set of goal states. This problem is a variation of the small racetrack in [1] so that when the car hit a wall it remains at that position with zero speed. The racetrack, shown in Fig. 2, is a grid of 35×12 positions (only the squares belonging to the racetrack are drawn); the $(0, 0)$ position (not shown) corresponds to the upper-left corner and the $(34, 11)$ the bottom-right corner of the racetrack. The possible start positions are the dark squares to the left and the goals are in the top-right. The state space consists of 4-tuples (x, y, dx, dy) where the first two components are the coordinates of the car and the last two the car's speed with respect to each dimension. The possible actions are pairs (ax, ay) of instantaneous accelerations where $ax, ay \in \{-1, 0, 1\}$. We also assume that the pavement is wet so the actions have probability 0.9 (resp. 0.1) of success (resp. failure). To give the transition function, we use a projection operator that maps states and actions to states by $proj(s, a) = (x + dx + ax, y + dy + ay, dx + ax, dy + ay)$. Then, the result of action a into state s is defined as: *(i)* s if the action fails, *(ii)* $proj(s, a)$ if the action succeed and $proj(s, a)$ is a valid racetrack position, or *(iii)* $(x, y, 0, 0)$ otherwise. Fig. 3 shows an encoding of this problem into GPT's language. In it, the effects *(i)* and *(ii)* are modeled in the effect for the unique action `control(ax,ay)`, and *(iii)* is modeled by using the ramification rule `constraints` and the variable `lastState`; other encodings that don't use ramification rules are also possible. Also, note how the information about the 'shape' of the racetrack is hooked into the description by using the 'external' function `valid`.

```
(define (domain sortnet)
  (:model (:dynamics :deterministic) (:feedback :null))
  (:objects array - :array[4] :integer[1,4])

  (:axiom axiom1
    :formula (:and (:not (= array[0] array[1]))
                   (:not (= array[0] array[2]))
                   (:not (= array[0] array[3]))
                   (:not (= array[1] array[2]))
                   (:not (= array[1] array[3]))
                   (:not (= array[2] array[3]))))

  (:action cmpswap
    :parameters ?i ?j - :integer[0,3]
    :precondition (< ?i ?j)
    :effect (:when (< array[?j] array[?i])
                   (:set array[?i] array[?j])
                   (:set array[?j] array[?i]))))


(define (problem p4)
  (:domain sortnet)
  (:init (:set array[0] :in :integer[1,4])
         (:set array[1] :in :integer[1,4])
         (:set array[2] :in :integer[1,4])
         (:set array[3] :in :integer[1,4]))
  (:goal (:and (< array[0] array[1])
               (< array[1] array[2])
               (< array[2] array[3]))))
```

Figure 4: Full description of the SORTN(5) problem.

6.2 Conformant Planning

Sorting Networks. A sorting network refers to a sorting algorithm in which comparisons and swaps are merged into a single operation that takes two entries i and j and swaps them if and only if they are not ordered. A conformant plan is given by the sequence of pairs i and j on which to apply this operation. The number of states in the problem is given by the possible ways in which the entries can be ordered; this is $n!$ for SORTN(n). The optimal cost of these problems is known for small values of n only ($n \leq 8$ according to [7]). GPT find optimal solutions, using A*, in a couple of minutes for n's up to 6. Fig. 4 shows the codification of sorting networks; note how we use the axiom to prune from the initial states those arrays with repeated integers. Instead of pruning such bad arrays, we can just not "generate" them by using

```
(:init (:set array[0] :in :integer[1,4])
       (:set array[1] :in :integer[1,4]
             :assert (:not (= array[0] array[1])))
       (:set array[2] :in :integer[1,4]
             :assert (:and (:not (= array[0] array[2]))
                           (:not (= array[1] array[2]))))
       (:set array[3] :in :integer[1,4]
             :assert (:and (:not (= array[0] array[3]))
                           (:not (= array[1] array[3]))
                           (:not (= array[2] array[3])))))
```

6.3 Contingent Planning

Medical. This problem, taken from [11], involves a patient that can be heathly or may have n diseases ($n = 1 \ldots 5$). The medication cures the patient if he has the right disease but kills the patient otherwise. The version with n diseases is denoted by MEDICAL(n). GPT solves this problems optimally by using the RTDP algorithm with a perfect stopping rule (i.e., $\epsilon = 0$). The MEDICAL(5) is solved by GPT in 1.759 seconds, it consists of 48 system states and the expected cost of the optimal policy is 4.6 steps. A codification of the problem appears in Fig.5 while a graph representation of the optimal policy is in Fig. 6.

6.4 Probabilistic Contingent Planning

Omelette. This problem, described before, is solved by GPT using a stopping rule with $\epsilon = 0.001$. The reported optimal

```
(define (domain medical)
  (:model (:dynamics :deterministic) (:feedback :partial))
  (:objects ill - :integer[0,5]
            stain_result - :integer[0,3]
            high_cell_count - :boolean
            dead - :boolean)

  (:action stain
     :effect
        (:when (:or (= ill 3) (= ill 4)) (:set stain_result 1))
        (:when (:or (= ill 1) (= ill 2)) (:set stain_result 2))
        (:when (= ill 5) (:set stain_result 3)))

  (:action count_white_cells
     :effect
        (:when (:or (= ill 1) (= ill 3) (= ill 5))
               (:set high_cell_count true)))

  (:action inspect
     :observation stain_result)

  (:action analyze_blood
     :observation high_cell_count)

  (:action medicate
     :parameters ?i - :integer[0,5]
     :precondition (:not (= ?i 0))
     :effect
        (:when (= ill ?i) (:set ill 0))
        (:when (:not (= ill ?i)) (:set dead true))))

(define (problem p5)
  (:domain medical)
  (:init (:set stain_result 0)
         (:set high_cell_count false)
         (:set ill :in { 1 2 3 4 5 })
         (:set dead false))
  (:goal (:and (= ill 0) (:not (= dead true)))))
```

Figure 5: Full description of the MEDICAL(5) problem.

expected cost is 23 steps which agrees with the analytical solution of $11 + 12(1-p)/p$ expected cost when the probability of a good egg is p. Fig. 7 shows the optimal policy function found by GPT. Observe that the solution has three cycles; each one guaranteeing a good egg. The following transcript shows how GPT's shell is used to compile, solve and generate the policy graph for it:

```
$ gpt
Welcome to GPT, Version 1.20
gpt> parse eggs omelette.pddl
0.129 seconds.
Successful parse.
Generated Files: eggs_omelette.h eggs_omelette-space.cc
Problem setted to "eggs_omelette".
gpt> compile
1.077 seconds.
Successful compilation.
Generated Files: egg_omelette.o
gpt> use stoprule .001
gpt> solve
184.683 seconds.
Generated Files: eggs_omelette.core
gpt> print graph > eggs_omelette.gml
gpt> bye
Good Bye.
```

7 Summary

We have described the GPT system for modeling, solving and analyzing different types of planning problems. The problems, characterized by the mathematical models of probabilistic, conformant, contingent and probabilistic contingent planning problems, can be described with GPT's high-level language. The language integrates ideas from STRIPS, first-order logic, and POMDPs, and is general enough for specifying a wide range of planning problems dealing with uncertainty and partial information. Future work on the GPT system includes the unification of GPT with the HSP planner [4], future improvements of the language and algorithms, and the development of a more friendly environment.

References

[1] A. Barto, S. Bradtke, and S. Singh. Learning to act using real-time dynamic programming. *Artificial Intelligence*, 72:81–138, 1995.

[2] D. Bertsekas. *Dynamic Programming and Optimal Control, Vols 1 and 2*. Athena Scientific, 1995.

[3] B. Bonet and H. Geffner. High-level planning and control with incomplete information using POMDPs. In *Proceedings AAAI Fall Symp. on Cognitive Robotics*, 1998.

[4] B. Bonet and H. Geffner. HSP: Planning as heuristic search. `http://www.ldc.usb.ve/~hector`, 1998.

[5] B. Bonet and H. Geffner. Planning with incomplete information as heuristic search in belief space. In *Proc. AIPS*, 2000.

[6] H. Geffner. Functional strips: a more general language for planning and problem solving. Logic-based AI Workshop, Washington D.C., 1999.

[7] D. Knuth. *The Art of Computer Programming, Vol. III: Sorting and Searching*. Addison-Wesley, 1973.

[8] R. Korf. Real-time heuristic search. *Artificial Intelligence*, 42:189–211, 1990.

[9] H. Levesque. What is planning in the presence of sensing. In *Proceedings AAAI-96*, pages 1139–1146, Portland, Oregon, 1996. MIT Press.

[10] D. McDermott. PDDL – the planning domain definition language. Available at `http://www.cs.yale.edu/dvm`, 1998.

[11] D. Weld, C. Anderson, and D. Smith. Extending Graphplan to handle uncertainty and sensing actions. In *Proc. AAAI-98*, pages 897–904. AAAI Press, 1998.

Appendix: Stopping Rule

Here, we briefly describe the stopping rule and its correctness. It is assumed the reader has knowledge about MDPs and the RTDP algorithm; good references are [2; 1].

RTDP iteratively solves the Bellman equations for the relevant states for both discounted and shortest-path problems [1]. The stopping rule is a *labeling* method that detects when the V-value has converged for the relevant states. Initially, all states except the goal states are labeled as *unsolved* and the goals ones as *solved*. The algorithm is stopped when all possible initial states are labeled as solved. We explain the procedure next.

Fix a trial $T_t = (s_0, s_1, \ldots, s_n)$ at time t. Let V_t be the value function right after the end of trial t; i.e., right after the last update for s_{n-1}. For $j = n \ldots 1$, let $K_j \subseteq S$ be the set of unsolved states reachable from s_j through V_t-optimal actions; i.e., K_j is a minimal set of unsolved states such that $s_j \in K_j$ and $(\forall s' \in K_j)(\exists s \in K_j)\big[P(s'; s, \pi_{V_t}(s)) > 0\big]$ where $\pi_{V_t}(s)$ is the V_t-optimal action in s. Then, at the end of each trial the algorithm labels as solved all states in set K_j in sequence from $j = n$ to $j = 1$, or until one of the next to conditions fails:

i) all states in sets K_l, for $l = n \ldots j+1$, are solved, and

ii) for all $s \in K_j$,

$$V_t(s) = c(s, \pi_{V_t}(s)) + \sum_{s' \in S} P(s'; s, a) V_t(s').$$

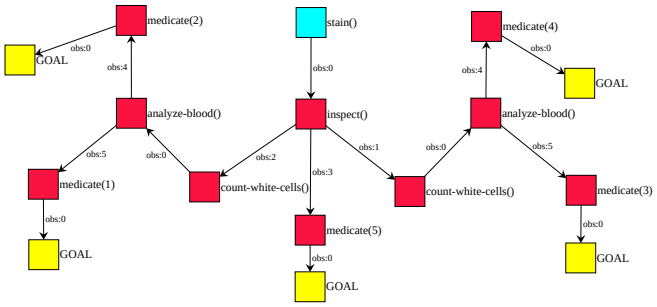

Figure 6: Optimal policy found by GPT for MEDICAL(5) using a perfect stopping rule; i.e., $\epsilon = 0$. '`obs:<n>`' refers to the feedback received by the controller after each action as follows: `obs:0` is no feedback, `obs:`n is `stain_result=`n for $n = 1, 2, 3$, `obs:4` is `high_cell_count=false` and `obs:5` is `high_cell_count=true`.

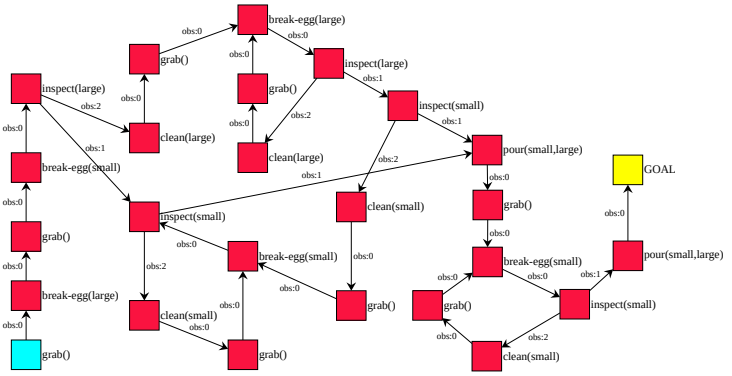

Figure 7: Optimal policy found by GPT for omelette using a stopping rule with $\epsilon = 0.001$. '`obs:<n>`' refers to the feedback received by the controller after each action as follows: `obs:0` is no feedback, `obs:1` is `no-bad-egg`, and `obs:2` is `bad-egg`.

By using induction on the trial number t and the set index j, it can be shown that every state s labeled as 'solved' at the end of trial t satisfy $V^*(s) = V_{t'}(s)$ for $t' > t$. Since RTDP converges to V^* over the relevant states, and the state space is finite, then RTDP will label as solved all relevant states. Also, it is not hard to show that this process finishes when all initial states are labeled solved.

Above method is referred to the perfect stopping rule. Since achieving *(ii)* could be very costly (depending on the lengths of the cycles in the state graph), we have implemented an ϵ version by relaxing the condition to

$$\max_{s \in K_j} \left| V_t(s) - c(s, \pi_{V_t}(s)) - \sum_{s' \in S} P(s'; s, a) V_t(s') \right| < \epsilon.$$

When the maximum is over all states, the quantity is known as the *Bellman residual*. As in value iteration where a zero Bellman residual is not necessary for an optimal induced policy, the stopping rule with above equation generates an optimal policy for sufficiently small ϵ.