GPT Meets PSR

Blai Bonet
Computer Science Department
University of California, Los Angeles
Los Angeles, CA 90024, USA
bonet@cs.ucla.edu

Sylvie Thiébaux
Computer Sciences Laboratory
The Australian National university
Canberra, ACT 0200, Australia
Sylvie.Thiebaux@anu.edu.au

Abstract

We present a case study in confronting the GPT general-purpose planner with the challenging power supply restoration (PSR) benchmark for contingent planning. PSR is derived from a real-world problem, and the dif£culty of modeling and solving it contrasts with that of the purely arti£cial benchmarks commonly used in the literature. This confrontation leads us to improve general techniques for contingent planning, to provide a PDDL-syle encoding of PSR which we hope to see used in planning competitions, and to report the £rst results on generating *optimal* policies for PSR.

Introduction

Coping with partially observable domains is arguably one of the most exciting and dif£cult challenges the £eld of planning has been faced with. Despite promising advances in using compact symbolic representations, heuristic search, and domain-speci£c control knowledge (Bertoli *et al.* 2001; Bonet & Geffner 2000; Hansen & Feng 2000; Karlsson 2001; Majercik & Littman 1999), the perception is that partial observability still lies on the critical path between planning research and applications. This perception is supported by the fact that, with very few exceptions (Cassandra *et al.* 1996; Bertoli *et al.* 2002), experimental results reported in the literature are con£ned to a well worn set of purely arti£cial benchmarks, featuring bombs in toilets, tigers hidden behind doors, and rotten eggs. We have reached the point where assessing the accuracy of this perception and pushing the envelope require that general-purpose uncertainty planners confront benchmarks derived from real-world applications, and that such confrontations begin to inform their development.

This paper reports a case study in doing just that. We confront the General Planning Tool (GPT) (Bonet & Geffner 2000) with the power supply restoration (PSR) challenge issued in (Thiébaux & Cordier 2001), and examine the impact that this benchmark had on new developments to GPT.

GPT's input language is close to PDDL, but additionally accounts for the possibility of partial sensor feedback and non-deterministic dynamics. GPT views planning under incomplete information (or contingent planning) as a problem of heuristic search in the belief space, which it solves optimally in £nite time using an off-line variant of Real-Time Dynamic Programming (RTDP) (Barto *et al.* 1995).

The PSR problem consists in planning actions to recon£gure a faulty power distribution network in such a way as to minimize breakdown costs. Due to sensor and actuator uncertainty, the location of the faulty areas and the current network con£guration are uncertain, which leads to a trade-off between acting to resupply customers and acting (intrusively) to reduce uncertainty. (Bertoli *et al.* 2002) reports the £rst successful attack of a general purpose planner on the PSR domain: MBP was able to solve non-trivial instances of the problem of generating a contingent plan guaranteed to achieve a given supply restoration goal. By contrast, our objective was to have GPT address the full scope of the PSR benchmark, for which there is no speci£ed goal but rather the request to minimize breakdown costs. This amounts to generating *optimal* policies for a contingent planning problem, which we wish to emphasize is typically much harder.

The fact that even small instances of this optimization problem were stretching GPT's limits motivated the development of *improved, general* methods for planning under incomplete information. E.g, to avoid the costly generation of a large number of irrelevant states, we had to resort to an incremental version of RTDP. This, in turn, required us to £nd clever ways of *dynamically* computing a domain-independent admissible heuristic for contingent planning (a variant of the well known QMDP), without generating the entire state space. This results in a substantial improvement in the ability of GPT to cope with larger problem instances.

In addition to reporting an interesting case study and improved techniques for contingent planning, this paper makes a number of contributions towards the use of PSR as a benchmark for planning under uncertainty. For instance, we give a concise, PDDL style, problem-independent encoding matching the informal description of PSR in (Thiébaux & Cordier 2001), which we expect to be particularly useful for future planning competitions.

We start with an overview of PSR and of the contingent planning model it can be recast as. We next present the techniques we developed to improve GPT's performance and in particular the dynamic computation of the heuristic. We then explain how we encoded PSR in GPT's input language, and why this required to extend GPT to deal with PDDL axioms

Copyright © 2003, American Association for Arti£cial Intelligence (www.aaai.org). All rights reserved.

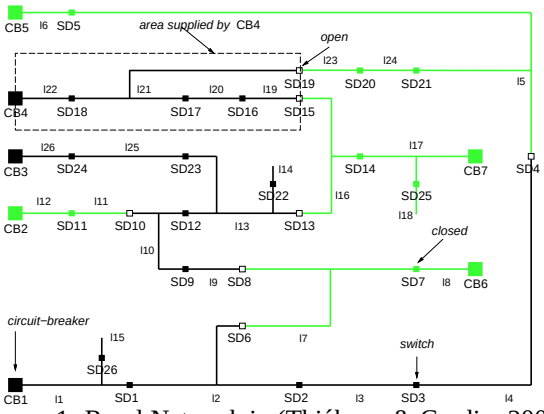

Figure 1: Rural Network in (Thiébaux & Cordier 2001)

(McDermott 1998). Finally, we present experimental results before summarizing our contribution and concluding with remarks about related and future work.

The PSR Benchmark

In (Thiébaux & Cordier 2001), the authors recast the problem of supply restoration in power distribution systems as a benchmark for planning under uncertainty. They give an informal description of the PSR benchmark which we now summarize, and issue the challenge of modeling and solving PSR with general purpose planning tools.

Network Topology For the purpose of the benchmark, a power distribution system (see Figure 1) consists of electric lines and devices of two types: circuit-breakers (large squares in the £gure) and switches (small squares). Devices are connected to at most two lines and have two possible positions: either closed or open (open devices, e.g. SD19, are white in the £gure). Circuit-breakers are viewed as power sources. When closed, they feed power into the network, and that power ¤ows through the various lines up to the point where it is stopped by an open device. The positions of the devices are initially set so that each circuit-breaker feeds a different area of the network (the area fed by CB4 is boxed in the £gure; Adjacent areas fed by different circuit-breakers are distinguished using dark and gray). The current network con£guration can be modi£ed by opening or closing devices. Closing and opening are the only available actions in PSR.

Faults, Supply Restoration Under bad weather, lines are often affected by permanent faults. When a line is faulty, the circuit-breaker feeding it opens to prevent overloads. This leads not just the faulty line but the entire area the breaker was feeding to be out of power. Supply restoration consists in recon£guring the faulty network so as to minimize breakdown costs: ideally we want to open and close devices in such a way as to isolate the faulty lines and resupply a maximum of the non-faulty lines on the lost areas. PSR would be relatively easy to solve if we knew the exact locations of the faulty lines and the current network con£guration. For instance, in case of a fault on l20 leading CB4 to open and the boxed area to be left without power, an adequate restoration plan, provided complete knowledge of the network state, would be to open SD16 and SD17 to isolate the fault, close CB4 to resupply l22 and l21, and close SD15 to have CB7 resupply l19. Unfortunately, PSR is much more complicated, because as we explain below, the sensors used to locate the faults and determine the devices' positions, as well as the actuators used to open/close devices, are unreliable.

Sensors Each device is equipped with a fault detector and a position detector, both continuously providing action-independent sensing information. The role of the position detector is to indicate the device's current position. Position detectors have two modes: "normal" in which the information they provide is correct, and "out of order" in which they do not provide any information at all. The role of the fault detectors is to help locate faults. Fault detectors have one "normal" mode and two failure modes: "out of order" and "liar". In normal mode, the fault detector of a fed device indicates whether or not it is upstream of a faulty line located on the same area – upstream is to be taken in relation to the ¤ow of current whose source is a circuit-breaker feeding the area. When not fed, a normal fault detector indicates the same information as when it was last fed. For instance, if only l20 is faulty, only the fault detectors of SD17 and SD18 should indicate that they are upstream of a fault; Then CB4 will open and the information provided by the fault detectors of the devices in the lost area should remain the same until they are fed again. In "liar" mode, fault detectors return the negation of the correct fault status, and in "out of order" mode, they do not return any information at all.

Actuators Each device is also equipped with an actuator whose role is to execute opening/closing actions and report on their execution status. Actuators also have a "normal", "out of order" and "liar" modes. In normal mode, the actuator of the prescribed device execute the requested action and sends a positive noti£cation. In "out of order" mode, the actuator fails to execute the action (the position remains unchanged) and sends a negative noti£cation. In "liar" mode, it also fails to execute the action but still sends a positive noti£cation. We take all sensor/actuator modes to be permanent across a supply restoration episode.

Minimizing Cost under Uncertainty Under sensor and actuator uncertainty, PSR takes another dimension. Many fault location and network con£guration hypotheses are consistent with the observations, and each of them corresponds to an hypothesis about the behavior modes (normal, liar) of the sensors and actuators. Since observations can only change when a device is operated, there is no non-intrusive way of gathering information to eliminate hypotheses. Often, decisive information comes at the price of an increase in breakdown costs: our best option to determine whether a line is faulty may be to resupply it via an healthy circuit-breaker, and check whether that breaker opens, yet leading a new area to be temporarily lost! Minimizing breakdown costs therefore amounts to trading off the need to act to resupply lines against that of acting to reduce uncertainty. This optimization problem is extremely challenging. To the best of our knowledge, even domain-speci£c solvers compute suboptimal solutions, see e.g. (Thiébaux *et al.* 1996).

Contingent Planning Problems

Classical planning and its generalizations can be understood in terms of a *state model* consisting of a set of states, a set of actions, and transition and observability functions. Different planning models correspond to state models with different types of transition and observability functions. In this paper, we consider *contingent planning*, whose state model combines a possibly non-deterministic transition function with a partial observation function, as this is the class into which PSR falls. Contingent planning models are characterized by:

(M1) A state space S,

(M2) an initial situation given by a state[1] $s_0 \in S$,

(M3) goal situations given by a non-empty $S_G \subseteq S$,

(M4) actions $A(s)$ applicable in each state $s \in S$,

(M5) a dynamics in which each action $a \in A(s)$ non-deterministically maps s into the set $F(s,a) \subseteq S$ of successor states,

(M6) positive costs $c(s,a)$ of performing action a in s such that $c(s,a) = 0$ for each $s \in S_G$, and

(M7) observations $o \in O(s,a)$ received when the actual state after the execution of a is s.

As is well-known, a solution for such a model is not a sequence of states but a policy which, at each decision stage, needs to consider the full history of observations and actions.

A simple characterization of such policies is achieved by considering *belief states*. In its simplest form, the term belief state refers to sets of states that the agent executing the policy deems possible at some point. Thus, the initial belief state b_0 is the singleton $\{s_0\}$, and if b denotes the belief state prior to performing an action a, the belief state b_a describing the possible states after the execution of a is

$$b_a \stackrel{\text{def}}{=} \cup\{F(s,a) : s \in b\}.$$

The actions $A(b)$ that can safely be applied in a situation described by belief state b are those that are applicable in all states compatible with b, i.e.

$$A(b) \stackrel{\text{def}}{=} \cap\{A(s) : s \in b\}.$$

The (worst-case) cost of performing action $a \in A(b)$ in belief state b and the observations that may result are

$$c(b,a) \stackrel{\text{def}}{=} \max\{c(s,a) : s \in b\},$$

$$O(b,a) \stackrel{\text{def}}{=} \cup\{O(s,a) : s \in b_a\}.$$

After performing action $a \in A(b)$ in belief state b, only one observation $o \in O(b,a)$ is received, and upon receipt of o, the next possible situations are characterized by the belief b_a^o de£ned as

$$b_a^o \stackrel{\text{def}}{=} \{s \in b_a : o \in O(s,a)\},$$

An interesting property of belief states—see e.g. (Bonet & Geffner 2000)—is that there is a straightforward equivalence between model M1–M7 and the following *non-deterministic and fully observable* state model in belief space:

(B1) A space of belief states B,

(B2) an initial belief state $b_0 \in B$,

(B3) goal belief states given by a non-empty $B_G \subseteq S$ where $b \in B_G$ iff $b \subseteq S_G$,

(B4) actions $A(b)$ applicable in each belief state $b \in B$,

(B5) a dynamics in which every action $a \in A(b)$ non-deterministically maps b into the set $F(b,a) = \{b_a^o : o \in O(b,a)\}$, and

(B6) positive costs $c(b,a)$ of performing action a in b.

A solution to state model B1–B6 is a policy mapping belief states to actions. In order to characterize valid solutions, we need to de£ne the set of π-trajectories, for policy π, as the collection of all £nite tuples $(b_0, \ldots, b_n)$ where $b_{k+1} \in F(b_k, \pi(b_k))$. Then, π is a solution to B1–B6 iff the number of π-trajectories is £nite and every π-trajectory can be extended to a π-trajectory ending in a goal belief state. This ensures that the policy reaches the goal and that its cost is well-de£ned, as

$$\text{cost}_{\text{worst}}(\pi) \stackrel{\text{def}}{=} \max\{cost(\tau) : \tau \text{ is } \pi\text{-trajectory}\}$$

where $cost(\tau)$ is the cost of trajectory τ de£ned as the sum of the costs of individual transitions within the trajectory. As usual, an (overall) optimal policy is one of minimum cost.

The above model targets the worst-case scenario. Later on, we will also be interested in minimizing the *expected* policy cost. In that case, the non-deterministic transition function and observation are enriched with transition and observation probabilities, so belief states are *probability distributions* over states rather than sets of states. The corresponding model in belief space is grounded in similar de£nitions for b_a and b_a^o which consider such probabilities, costs are expectations rather than maxima, and £niteness is not required, see (Bonet & Geffner 2000) for details.

General Planning Tool

As demonstrated in (Bonet & Geffner 2000; 2001), contingent planning problems can be solved *optimally* using algorithms based on heuristic search in belief space. In particular, (Bonet & Geffner 2001) presented a novel *off-line* version the RTDP algorithm which solves such problems in £nite time.[2] This version of RTDP is at the core of the GPT planner where it is used to solve a range of planning models.

The GPT planner is a *domain-independent* planning system which works by converting the description of a planning problem into an appropriate state model (either in state space or belief space) and then applying an heuristic search algorithm. The term domain-independent refers to the fact that GPT only uses the description of the problem in order to solve it. Thus, for example, no domain-speci£c control knowledge and/or heuristic functions are allowed.

Before starting the search, GPT performs a precompilation step in which, among other things, it generates the part of the *state* space that is *reachable* from s_0 and computes an admissible domain-independent heuristic estimate for each

[1] There is no loss of generality in assuming a single initial state: multiple initial states can be encoded as non-deterministic effects of a `start` action constrained to be the only action applicable in the initial state.

[2] The standard RTDP algorithm is a learning algorithm that only converges asymptotically.

reachable state. This heuristic, called h_{QMDP}, is used to de£ne a heuristic function over belief states for the worst-case or expected-case scenario as follows:

$$h_{\text{worst}}(b) \stackrel{\text{def}}{=} \max \{ h_{\text{QMDP}}(s) : s \in b \},$$
$$h_{\text{exp}}(b) \stackrel{\text{def}}{=} \sum_{s \in S} h_{\text{QMDP}}(s) \cdot b(s).$$

where, in the expected case, $b(s)$ is the probability of s in belief state b. It is not hard to show that $h_{\text{worst}}(b)$ and $h_{\text{exp}}(b)$ are admissible heuristics for the respective contingent planning problems. Therefore, armed with these heuristics, the search algorithm over belief states is guaranteed to return optimal solutions.

In our benchmarks, we £nd that GPT spends almost all its time in this precompilation step and even runs out of memory due to the huge number of reachable states. This limits its applicability to challenging problems such as PSR. One way to avoid this bottleneck is to generate states and compute the heuristic *incrementally*, as the search algorithm explores the belief state space. In principle, computing h_{QMDP} without necessarily generating the entire reachable state space is feasible using algorithms such as labeled RTDP or LAO* (Bonet & Geffner 2001; Hansen & Zilberstein 2001). However, in the restricted case of our contingent planning models, we are able to devise a more ef£cient approach by considering a slightly different admissible heuristics h_{DYN} and using classical heuristic search to *dynamically* compute the estimates for the states as they are required.[3]

Incremental state computation and dynamic computation of heuristic estimates are two domain-independent techniques and have been implemented in the GPT system. We empirically demonstrate their bene£ts later on in the paper.

The idea of incremental generation of the state space is easy to implement. Simply, whenever a belief state b is expanded, GPT checks if all states $s \in b$ have already been compiled (incorporated) into the model. If not, GPT expands such states and compiles the new information. The dynamic computation of the heuristic, however, is more subtle.

Dynamic Computation of the Heuristic

We de£ne $h_{\text{DYN}}(s)$ as the cost of the shortest path from state s to a goal state. It turns out that in the worst-case scenario, $h_{\text{QMDP}}(s)$ equals $h_{\text{DYN}}(s)$, and that in the expected-case scenario, $h_{\text{DYN}}(s)$ is a lower bound on $h_{\text{QMDP}}(s)$. h_{DYN} can be computed with algorithms such as Dijkstra or Uninformed cost search but in our case, where many searches will be performed for different states, we are able to do better.

The basic idea behind the algorithm is to collect the information found in previous searches for its reuse in future searches. This is achieved by using an IDA* search (Korf 1985) with transposition tables (Slate & Atkin 1977). In the following discussion, we assume a deterministic transition function; a problem with non-deterministic transitions can be mapped onto a deterministic one by adding actions for the different non-deterministic effects.

[3]The use of classical heuristic search methods as for h_{DYN} (or even of RTDP or LAO* as for h_{QMDP}) raises the question of which heuristic function to use in such a search, i.e. which heuristic to use for computing the heuristic. An interesting possibility is to extract information from the problem encoding as it is done in state-of-the-art STRIPS planners. However, we just take $h = 0$ here.

```
IDA(s : state)
begin
    t = 0
    while ¬goal_found ∧ t < ∞ do
        t = boundedDFS(t, s, 0)
    return t
end

boundedDFS(t : real, s : state, g : real)
begin
    // base cases
    f = g + TTABLE[s].v
    if TTABLE[s].solved then
        if f ≤ t then
            goal_found = true
        return f
    else if f > t then
        return f
    else if isGoal(s) then
        goal_found = true
        return g

    // expand state
    new_t = ∞
    for a ∈ A(s) do
        f = boundedDFS(t, res(s, a), g + c(s, a))
        if goal_found then
            new_t = f
            break
        new_t = min{new_t, f}

    // update transposition table
    v = new_t − g
    if TTABLE[s].v < v then
        TTABLE[s].v = v
    if goal_found then
        TTABLE[s].solved = true

    return new_t
end
```

Algorithm 1: IDA* algorithm for h_{DYN}.

The standard IDA* algorithm performs a series of cost-bounded depth £rst searches with successively increasing cost thresholds. As usual, the total value f of a node is composed of the cost g already spent in reaching that node and of the estimated cost h of reaching the goal. At each iteration, the search only expands nodes whose value is below the current threshold t, cutting off all other nodes. The threshold is initialized to the estimated cost h of the start state (0 in our case), and is increased at each iteration to the minimum path value that exceeds the previous threshold.

A transposition table TTABLE is a table indexed by states that is used to store the result of previous searches, thereby preventing IDA* from re-exploring previously visited nodes. Upon visiting state s with cost g and having completed the cost-bounded search below it with result f, the transposition table for s is updated as $\text{TTABLE}[s] = f - g$. Then, whenever s is re-visited with a cost g such that $f = g + \text{TTABLE}[s]$ exceeds the current threshold t, the algorithm can safely increase the threshold to f without further exploration. Otherwise the search below s is performed and the transposition table is updated as explained above.

It is not hard to see that $\text{TTABLE}[s]$ is always a *lower bound* on $h_{\text{DYN}}(s)$. Thus, if the goal is found when searching below s, the lower bound becomes *exact* and no future searches below s are necessary. A description of this IDA* search is given in Algorithm 1. As can be seen, the transposition table has two £elds: $\text{TTABLE}[s].v$ which contains

the current lower bound on $h_{\text{DYN}}(s)$, and TTABLE$[s].solved$ which says whether this bound is exact. The table is initialized with TTABLE$[s].v = h(s)$ (i.e. 0 in our case) and TTABLE$[s].solved = false$.

PDDL-like encodings of PSR

GPT's input language is a variant of PDDL based on the functional version of STRIPS (Geffner 2000) in which states are not merely sets of atoms but £rst-order models over £nite domains. Terms with arbitrary nesting of function symbols are allowed and actions may modify functional ¤uents. Like PDDL, the language enables a clear separation of the domain de£nition from that of the problem instances. Figure 2 shows our encoding of the full PSR domain, together with that of a problem instance with 4 faulty lines at unknown locations for the `simple` network in Figure 3. We now describe the most important elements of the encoding.

States

A domain de£nition £rst speci£es the class of planning models the domain belongs to (keyword `:model`), and declares the types, function, predicate and object symbols used.

We need types `DEVICE` and `LINE`, a type `SIDE` and objects `side1` and `side2` to distinguish between the two sides of each device to which lines can connect, as well as a type `MODE` and objects `ok`, `out`, and `liar` denoting the possible fault detector and actuator modes (for a position detector mode, a simple boolean suf£ces).

We represent a state of PSR as follows. The ¤uents `closed`, and `faulty` have the obvious readings. The functional ¤uents `ac_mode` and `fd_mode` return the modes of the actuator and fault detector of a device, while `pd_ok` tells whether its position detector behaves normally, and `fault_status` tells whether the device was upstream of a fault when last fed. Other ¤uents refer to the topology of the network: `ext` tells whether a given line is connected to a given side of a device, `breaker` tells whether a device is a circuit-breaker, and `opposite` maps `side1` to `side2` and vice versa. The initial values of functions and predicates are given in the problem instance de£nition. Some values may not be completely known. For our example in Figure 2, all values are known, except that of `faulty`: there is just a constraint that the number of faulty lines equals 4. Note that propositions not mentioned at all in the initial state are taken to be false. By inspecting the domain and problem de£nitions, GPT is able to identify and compile away those propositions with a £xed value across the entire set of states.

Axioms

One of the dif£culties of PSR is that actions have relatively complex effects: e.g. when we close a device, a faulty line may become fed and affect devices upstream of it in some way (breakers open, and switches have their fault status changed). Being upstream is a dynamic notion which depends on the current network con£guration, and so needs to be computed after each action execution. Furthermore, computing it requires an iterative or recursive traversal of the network's paths, and there is no intuitive way of doing this in the body of a PDDL-style action. Consequently, to model PSR actions while keeping the encoding independent of a particular network, we need to axiomatize upstream as a *derived* predicate. Such a predicate is one whose value is derived from the current value of other predicates and functions and cannot be directly modi£ed by actions.

We found that the best option to de£ne derived predicates such as `upstream` was to use directional recursive axioms of the form `(:predicate (name arguments) condition)`, much as in the original version of PDDL (McDermott 1998). The meaning is that when the condition is true, we should *infer* that the value of the predicate at the speci£ed arguments is true. Note that these inferences cannot be contraposed and that what cannot be inferred as true is false. This closed world assumption is a crucial strength of PDDL axioms. Were we to use £rst order logic axioms instead, we would be unable to axiomatize upstream (which is the transitive closure of a relation) since transitive closure is not £rst-order axiomatizable in general. Upstream *can* be axiomatized in £rst order logic under assumptions such as that the network is loop-free and each device is fed only by one breaker (i.e., the closure is irre¤exive and the base relation is a function), but even under these restrictions, it is not trivial as in the absence of the closed world assumption we need to specify when `upstream` does *not* hold.

In our encoding, `(upstream ?x ?sx ?y ?sy)` means that the current produced by some circuit-breaker ¤ows from side `sx` of device `x` to side `sy` of device `y`. It is necessary to speak of devices' sides rather than devices in order to keep proper track the direction of the current ¤ow. The three conjuncts respectively ensures that the current ¤ows (1) up to side `sx` of `x`, (2) through `x`, and (3) on as far as side `sy` of `y`.[4] For readability, we de£ne four other derived predicates: `con` which tells whether two devices' sides are connected via a given line, `affected` which tells whether a device is upstream of a faulty line, `fed` which tells whether a device is fed by checking that there is a side of a device upstream of one of its sides (or that the device is a closed breaker), and `fed_line` which tells whether a line is fed by checking that it is connected to a closed fed device.

If a pure PDDL encoding (without axioms) is required, it is possible to automatically compile axioms into additional context-dependent effects of existing actions or into additional actions. However, this leads to gross ine£ciency, and in the worst-case, to exponentially larger domain descriptions or exponentially longer plans (Thiébaux *et al.* 2003).

Actions, Observations

With the help of the derived predicates and GPT's rami£cation rules (keyword `:ramification`), PSR actions can be expressed very concisely. The syntax of a rami£cation rule is similar to that of an action, except that the `:observation` and `:cost` £elds do not apply. While the effects of an action become true at the next time step, those of a rami£cation rule become true immediately. These rules are useful to specify indirect action effects and domain constraints. When an action is executed, it effects are computed, then the

[4] In our experiments, we in fact use a more ef£cient but longer encoding of `upstream` and other derived predicates, which is omitted on grounds of readability and space.

```
(define (domain psr)
  (:model (:dynamics :deterministic) (:feedback :partial))
  (:types DEVICE SIDE LINE MODE)
  (:functions
    (ac_mode DEVICE MODE)
    (fd_mode DEVICE MODE)
    (opposite SIDE SIDE))
  (:predicates
    (ext LINE DEVICE SIDE)
    (breaker DEVICE)
    (closed DEVICE)
    (faulty LINE)
    (fault_status DEVICE)
    (pd_ok DEVICE))
  (:objects side1 side2 - SIDE
            ok out liar - MODE
            done - :boolean)

  (:predicate (con ?x - DEVICE ?sx - SIDE ?y - DEVICE ?sy - SIDE)
     (:and (:or (:not (= ?x ?y))
                (:not (= ?sx ?sy)))
           (:exists ?l - LINE
              (:and (ext ?l ?x ?sx) (ext ?l ?y ?sy)))))

  (:predicate (upstream ?x - DEVICE ?sx - SIDE ?y - DEVICE ?sy - SIDE)
    (:and (:or (breaker ?x)
               (:exists ?z - DEVICE
                  (:exists ?sz - SIDE
                     (:and (con ?z (opposite ?sz) ?x ?sx)
                           (upstream ?z ?sz ?x ?sx)))))
          (closed ?x)
          (:or (con ?x (opposite ?sx) ?y ?sy)
               (:exists ?z - DEVICE
                  (:exists ?sz - SIDE
                        (:and (closed ?z)
                              (con ?z (opposite ?sz) ?y ?sy)
                              (upstream ?x ?sx ?z ?sz)))))))

  (:predicate (affected ?x - DEVICE)
    (:exists ?l - LINE
       (:and (faulty ?l)
             (:exists ?y - DEVICE
                (:exists ?sy - SIDE
                   (:and (ext ?l ?y ?sy)
                         (:exists ?sx - SIDE
                            (upstream ?x ?sx ?y ?sy)))))))))

  (:predicate (fed ?x - DEVICE)
    (:or (:and (breaker ?x) (closed ?x))
         (:exists ?y - DEVICE
            (:exists ?sy - SIDE
               (:exists ?sx - SIDE
                  (upstream ?y ?sy ?x ?sx))))))

  (:predicate (fed_line ?l - LINE)
    (:exists ?x - SWITCH
      (:exists ?sx - SIDE
        (:and (ext ?l ?x ?sx) (closed ?x) (fed ?x)))))

  (:ramification status_ramification
    :parameters ?x - DEVICE
    :effect (:when (fed ?x)
                   (:set (fault_status ?x) (:formula (affected ?x)))))

  (:ramification open_ramification
    :parameters ?x - DEVICE
    :effect (:when (:and (breaker ?x) (affected ?x))
                   (:set (closed ?x) false)))

  (:action open
    :parameters ?x - DEVICE
    :effect (:when (= (ac_mode ?x) ok) (:set (closed ?x) false))
    :observation
      (= (ac_mode ?x) out)
      (:vector ?y - DEVICE
         (:if (pd_ok ?y) (:formula (closed ?y)) false))
      (:vector ?y - DEVICE
         (:if (= (fd_mode ?x) ok) (:formula (fault_status ?x))
              (:if (= (fd_mode ?x) liar)
                   (:formula (:not (fault_status ?x)))
                   false))))

  (:action close
    :parameters ?x - DEVICE
    :effect (:when (= (ac_mode ?x) ok) (:set (closed ?x) true))
   :observation
      (= (ac_mode ?x) out)
      (:vector ?y - DEVICE
         (:if (pd_ok ?y) (:formula (closed ?y)) false))
      (:vector ?y - DEVICE
         (:if (= (fd_mode ?x) ok) (:formula (fault_status ?x))
              (:if (= (fd_mode ?x) liar)
                   (:formula (:not (fault_status ?x)))
                   false))))

  (:action finish
    :effect (:set done true)
    :cost (:sum ?l - LINE
            (:if (:or (faulty ?l) (fed_line ?l)) 0 5))))

(define (problem simple)
  (:domain psr)
  (:objects l1 l2 l3 l4 l5 l6 l7 - LINE
            cb1 cb2 cb3 sd1 sd2 sd3 sd4 sd5 sd6 sd7 - DEVICE)
  (:init
     (:set (opposite side1) side2) (:set (opposite side2) side1)
     (:set done false)

     (:set (breaker cb1) true)
     (:set (breaker cb2) true)
     (:set (breaker cb3) true)

     (:set (ext l1 cb1 side2) true) (:set (ext l1 sd6 side1) true)
     (:set (ext l2 sd6 side2) true) (:set (ext l2 sd5 side1) true)
     (:set (ext l2 sd7 side2) true) (:set (ext l3 sd5 side2) true)
     (:set (ext l3 sd1 side1) true) (:set (ext l4 sd1 side2) true)
     (:set (ext l4 sd2 side2) true) (:set (ext l4 sd3 side2) true)
     (:set (ext l5 cb2 side2) true) (:set (ext l5 sd4 side1) true)
     (:set (ext l6 sd2 side1) true) (:set (ext l6 sd4 side2) true)
     (:set (ext l6 sd7 side1) true) (:set (ext l7 cb3 side2) true)
     (:set (ext l7 sd3 side1) true)

     (:foreach ?x - DEVICE
        (:set (closed ?x) true)
        (:set (fd_mode ?x) ok)
        (:set (pd_ok ?x) true)
        (:set (ac_mode ?x) ok))
     (:set (closed sd3) false)
     (:set (closed sd5) false)
     (:set (closed sd7) false)
     (:foreach ?l - LINE (:set (faulty ?l) :in { true false }))
     (= (:sum ?l - LINE (:if (faulty ?l) 1 0)) 4))

  (:goal (= done true)))
```

Figure 2: Encoding of the Full PSR & Simple Problem

rami£cation rules are applied to the resulting state in the order they appear, and only then the action's observations and cost are evaluated. Note that the rami£cation rules are also applied in the initial state, before any action takes place.

In our encoding of the full PSR, PSR actions have no precondition. Their effect is simply to change the position of the device unless the actuator is abnormal. Then the rami£cation rules take care of setting the fault status of fed devices appropriately (if a device is affected its status is set and otherwise unset), and of opening affected breakers. The observations include the noti£cation of the operated device which is positive iff the device's actuator is not out of order, the position of the devices whose position detector is normal, as well as the fault status of the devices whose fault detector is normal, or their negation for those whose fault detector lies.[5]

A pure PDDL encoding of the actions can be obtained by simulating GPT's rami£cation rules via an extra action treating fed devices and affected breakers, which we can easily constrain to interleave with the other actions. Another option is to add a parameter `?c` to the derived predicates, and make their value conditional upon device `c` being closed. This is achieved by replacing `(closed ?x)` in the present de£nitions with `(Cclosed ?c ?x)` de£ned to be true when `x` is closed or `c` = `x`. The actions' effects are then easily described using the conditional version of `affected`.

Goal

As we want to address the full scope of the benchmark for which there is no speci£ed goal but the request to minimize

[5]For space reasons the encoding in Figure 2 sloppily identi£es the absence of information with `false`. However, it is possible to create a three valued type `OBS` and return observations of that type.

breakdown costs, we formulate the problem as a pure optimization problem. We declare a proposition `done` initially false, and an action `finish` which makes `done` true and whose cost is linear in the number of unsupplied healthy lines. Note that all other actions have the default cost of 1. Setting the goal to `done` allows GPT to £nish the plan at any step by paying the price corresponding to the current breakdown costs. Therefore, any optimal policy found by GPT for the goal `done` minimizes breakdown costs.

It is important to understand that minimal breakdown costs cannot be achieved by supplying GPT with a restoration goal such as "supply all lines that can be supplied". The reason is that partial observability sometimes prevents the existence of a policy satisfying such a goal, even though actions can still be taken to reduce breakdown costs.[6]

Comparison with $\mathcal{AR}$ encodings

Our encoding of PSR differs substantially from the $\mathcal{AR}$ encodings used by MBP (Bertoli *et al.* 2002), which are propositional in nature and are automatically generated for a given network by a custom procedure. The procedure computes all minimal acyclic paths in the network and uses those to determine all the conditions on device positions and line modes under which a given device is affected by an action. Unfortunately, the number of these conditions and consequently the action description can grow exponentially large in the number of devices in the network. This together with the propositional character of $\mathcal{AR}$ leads to huge descriptions (over 8 MB for the rural network in Figure 1), which is to be contrasted with our concise network-independent encoding. On the other hand, the number of propositions in the $\mathcal{AR}$ encoding is smaller than that induced by the present one, and this positively impacts on the ef£ciency of BDD-based planners such as MBP. Indeed, we found that a propositional expansion of our encoding into $\mathcal{AR}$ caused computational dif£culties for MBP and that vice-versa the $\mathcal{AR}$ encoding generated by the procedure was unmanageable for GPT.

The goal constitutes another difference between the two encodings. Since MBP does not reason about plan cost, (Bertoli *et al.* 2002) treats PSR as the problem of £nding a contingent plan achieving the goal of supplying all suppliable lines, under various additional assumptions (D1-D3) to be discused below. As deciding which lines are suppliable in a given state is a non-trivial problem, the $\mathcal{AR}$ encoding identi£es suppliable with a stricter condition of existence of a "safe path" between a breaker and the line, that is a path consisting entirely of non-faulty lines and maneuverable devices (i.e. with normal actuators). As we argued above, GPT is able to act optimally even when no such plan exists. We wish to emphasize that our encoding is targeted at solving a typically much harder problem, requiring optimization.

[6]For a small example, consider a linear network with 3 lines, one breaker at each end and two switches. From left to right, let us call those `CB1`, `SD1`, `SD2`, `CB2`. All are initially closed, except `SD2` which is open. There is at least one fault on the area fed by `CB1`, all fault detectors are out of order, the position detector of `CB2` is out of order, and everything else is correct. We leave it as an exercise to the reader to show that no policy can decide whether it has supplied all suppliable lines, and that on the other hand opening `SD1` and closing `CB1` reduces the expected breakdown costs.

Experimental Results

Domains, Algorithms, and Optimization Criteria

Our experiments involve several variants of the full PSR given above, the comparison of several heuristics and algorithms for computing them, and two optimization criteria.

Domains We consider the following modi£cations to the full PSR domain in Figure 2.

(D1) We prevent closing actions to power loops or create areas fed by multiple breakers by setting the precondition of `(close ?x)` to `(:not (bad ?x))`, where `bad` is true for switches fed on two sides, and for breakers connected to a loopy part of the network.

(D2) We prevent GPT to open a device which is currently fed by setting the precondition of `(open ?x)` to `(:not (fed ?x))`, and favor closing over opening when breaking ties.

(D3) We give GPT the goal of supplying all lines than can be supplied, where we identify suppliable with the stricter notion of safe path existence explained above.

(D4) We make the fault detector of a unfed device return no information rather than the same information it was returning when last fed.

Modi£cations D1-D3 lead GPT to address essentially the same domain as MBP, under the same assumptions as the experiments reported in (Bertoli *et al.* 2002). This enables a somewhat fairer comparison of the strength and weaknesses of both planners, although GPT still attempts to minimize action costs while MBP does not. D1 is useful on its own, as no loops and no double feeding is a standard assumption in supply restoration. D4 enables us to address larger problem instances than would be normally possible: it obviates the need to remember fault status and make them state variables, which signi£cantly reduces the number of states.

We experiment with the following domain variations, standard (**std**): the full PSR domain with modi£cation D1, **mbp**: the full domain with modi£cations D1–D3, as well as **std*** and **mbp***, which result from applying modi£cation D4 to std and mbp, respectively.

Algorithms For each domain, we experiment with the following 3 algorithms, **org.**: the original implementation of RTDP with precompilation step and h_{QMDP} heuristic, **incr.**: RTDP with incremental state space generation and dynamic computation of h_{DYN}, and **h=0**: RTDP with incremental state space generation seeded with $h(b) = 0$ for all belief states b.

Optimization Criteria We produce policies with minimal expected (**exp**) or worst-case (**wst)** cost. For mbp and mbp* domains, where the supply restoration goal is given, the cost only includes (unit) action costs, not breakdown costs. For the exp criteria, we take all initial states in each problem instance to be equiprobable, as this avoids making up fault probabilities. In principle, however, GPT can handle any distribution. In particular, it would be straightforward to specify independent fault and mode probabilities and consider the corresponding distribution.

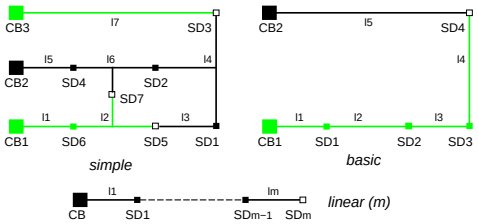

Figure 3: Small Test Networks

Networks, Problems and Results

Small Networks We first tested GPT on the small networks `basic`, `simple` and `linear` in Figure 3. For `basic`, we considered 5 problem instances `b1`–`b5`, where `b`n has *at most* n faulty lines at unknown locations. For `simple`, we considered 7 instances `s1`–`s7`, where `s`n has *exactly* n faulty lines at unknown locations (e.g. the PDDL encoding of `s4` is shown in Figure 2). Similarly, `l`m`_e`n is an instance of the linear network with m lines and exactly n faulty lines – we considered 12 such instances. All experiments with small networks were run on a standalone Ultra-10 with 300MB of memory and a clock speed of 440MHz.

Figures 4 and 5 show the results obtained by GPT with incremental state space generation and the h_{DYN} heuristic (incr. algorithm) for the various domain versions and optimization criteria. The left-hand sides of the tables refer to domains mbp and std, and the right hand-sides to the * versions mbp* and std*. The tables show the run time (sec.), the number of states generated, and the optimal policy cost. Note that the cost of mbp(*) and std(*) policies are incomparable. A dash (–) means that GPT ran out of memory.

The *original* version of GPT (org. algorithm) run out of memory during the precompilation step for basically all but the tiniest instances. As shown in the figures, the incr. algorithm is at least able to cope with all instances of the * domain versions which lack the additional exponential growth of the number of reachable states. The price to pay for the time and memory gain provided by the * versions is a reduction in policy quality. For instance, even in a very simple instance such as `l4_e2`, the cost of the optimal policy for mbp/exp (resp. std/exp) is 2.50 (resp. 9.16), and that for mbp*/exp (resp. std*/exp) is 3.00 (resp. 9.50).

Another observation is that the run times for `simple` and `linear`, for which we know the *exact* number of faults, exhibit an easy-hard-easy pattern. The same phenomenon was observed in (Bertoli *et al.* 2002), and was attributed to the ability of MBP's symbolic algorithm to exploit problem structure. Since GPT's algorithm enumerates states and is unable to exploit such structure, it appears that the real cause for the pattern is that the number of states is dictated by the number of ways of choosing n faulty lines among m, which peaks at $n = m/2$. We also note that critically constrained std worst-case instances are much easier to solve optimally than their expected-case counterpart. This is because when there are at least as many faults as breakers, in the worst case nothing is resuppliable and so the optimal policy is to do nothing, while in the expected case the policy must still prescribe what to do for other situations besides the worst.

Sticking with incremental state generation and * versions,

prob.	regular versions time	cost	states	* versions time	cost	states
	mbp/wst/incr			mbp*/wst/incr		
l4_e0	0.14	0.00	2	0.15	0.00	2
l4_e2	0.34	4.00	507	0.30	4.00	310
l4_e4	0.14	0.00	7	0.15	0.00	7
l6_e0	0.14	0.00	2	0.15	0.00	2
l6_e2	4.15	5.00	5838	2.74	5.00	3152
l6_e4	3.20	4.00	6073	1.60	4.00	2418
l6_e6	0.14	0.00	9	0.17	0.00	9
l8_e0	0.14	0.00	2	0.19	0.00	2
l8_e2	57.40	5.00	42167	30.43	5.00	20880
l8_e4	–	–	–	139.04	6.00	55383
l8_e6	26.54	4.00	30020	12.47	4.00	12255
l8_e8	0.19	0.00	11	0.19	0.00	11
	mbp/exp/incr			mbp*/exp/incr		
l4_e0	0.26	0.00	2	0.32	0.00	2
l4_e2	0.55	2.50	525	0.52	3.00	316
l4_e4	0.27	0.00	7	0.28	0.00	7
l6_e0	0.27	0.00	2	0.29	0.00	2
l6_e2	6.71	3.20	6950	4.31	3.66	3316
l6_e4	7.67	2.86	10716	4.69	3.20	3056
l6_e6	0.27	0.00	9	0.31	0.00	9
l8_e0	0.28	0.00	2	0.32	0.00	2
l8_e2	100.89	3.57	56126	52.59	4.00	24902
l8_e4	–	–	–	236.28	3.97	59376
l8_e6	278.88	2.75	136024	47.79	2.92	22334
l8_e8	0.34	0.00	11	0.32	0.00	11
	std/wst/incr			std*/wst/incr		
l4_e0	0.15	0.00	53	0.17	0.00	53
l4_e2	0.51	10.00	1399	0.30	10.00	549
l4_e4	0.14	0.00	42	0.18	0.00	42
l6_e0	0.17	0.00	86	0.16	0.00	86
l6_e2	18.22	20.00	25360	3.35	20.00	5110
l6_e4	24.26	10.00	46695	2.15	10.00	4526
l6_e6	0.16	0.00	72	0.16	0.00	72
l8_e0	0.23	0.00	123	0.22	0.00	123
l8_e2	–	–	–	60.11	30.00	37718
l8_e4	–	–	–	109.97	20.00	85452
l8_e6	–	–	–	22.41	10.00	31157
l8_e8	0.23	0.00	110	0.19	0.00	110
	std/exp/incr			std*/exp/incr		
l4_e0	0.28	0.00	53	0.30	0.00	53
l4_e2	0.97	9.16	1726	0.64	9.50	626
l4_e4	0.28	0.00	42	0.30	0.00	42
l6_e0	0.29	0.00	86	0.29	0.00	86
l6_e2	30.37	16.53	30809	8.37	17.00	6627
l6_e4	32.13	10.00	51569	3.34	10.00	5385
l6_e6	0.29	0.00	72	0.29	0.00	72
l8_e0	0.37	0.00	123	0.35	0.00	123
l8_e2	–	–	–	151.65	24.00	50538
l8_e4	–	–	–	521.30	19.42	118752
l8_e6	–	–	–	24.11	10.00	31907
l8_e8	0.37	0.00	110	0.33	0.00	110

Figure 4: Results for `linear` (incr. algorithm)

Figure 6 evaluates the benefits of the h_{DYN} heuristic (incr. algorithm) in comparison to $h = 0$. The run time improvement is dramatic, up to the point where even some std*/exp instances are not solvable with the zero heuristic. Similar results are obtained with the other domains and criteria.

Larger Networks After testing GPT on small instances, we considered the challenging `rural` network instance solved (non-optimally) by MBP. In the original instance, see (Bertoli *et al.* 2002), everything about the rural network in Figure 1 is known to be correct, except the mode of lines `l3` and `l15`, the mode of the fault detectors of `SD1`, `SD2`, `SD3`, `S26`, as well as the mode of the position detector and actuator of `SD26` which are unknown. This leads to 1944 initial states and to a myriad of reachable states ($\simeq 10^{20}$ for std).

Unfortunately, even equipped with h_{DYN}, GPT was unable to solve this instance. To identify the largest scaled down version of this problem that GPT could solve, we considered two variations of the rural network. `simplified-rural` is like `rural` except that we remove all intermediate switches on areas other than that fed by `CB1`. On

prob.	regular versions time	cost	states	* versions time	cost	states
	mbp/wst/incr			mbp*/wst/incr		
b1	0.58	5.00	1065	0.34	5.00	445
b2	2.16	5.00	3992	1.00	5.00	1341
b3	4.31	6.00	7091	1.85	6.00	2115
b4	4.77	6.00	7952	2.07	6.00	2452
b5	4.40	6.00	7862	2.05	6.00	2510
s1	46.86	5.00	28987	9.28	5.00	5552
s2	–	–	–	42.72	8.00	22720
s3	–	–	–	105.72	9.00	36689
s4	–	–	–	107.89	8.00	31489
s5	354.60	6.00	198621	26.30	6.00	14909
s6	7.51	4.00	10935	2.35	4.00	2862
s7	0.17	0.00	13	0.18	0.00	13
	mbp/exp/incr			mbp*/exp/incr		
b1	0.80	3.00	1065	0.52	3.16	466
b2	2.39	3.31	3689	2.22	3.87	1377
b3	6.15	3.38	6905	4.23	3.84	2178
b4	7.85	3.32	7825	4.16	3.74	2532
b5	8.67	3.31	8469	4.68	3.71	2599
s1	47.53	3.28	28987	8.70	3.42	5155
s2	–	–	–	42.25	5.04	22091
s3	–	–	–	103.28	5.42	36312
s4	–	–	–	247.63	5.31	32190
s5	–	–	–	115.31	4.42	16775
s6	30.14	2.42	31614	6.45	3.00	3967
s7	0.30	0.00	13	0.33	0.00	13
	std/wst/incr			std*/wst/incr		
b1	1.13	5.00	2572	0.60	5.00	1096
b2	5.41	15.00	12200	0.89	15.00	2192
b3	10.21	15.00	24005	1.30	15.00	3378
b4	10.74	15.00	26011	1.45	15.00	3862
b5	10.96	15.00	26022	1.48	15.00	3916
s1	61.58	4.00	44412	16.68	5.00	11174
s2	–	–	–	75.74	9.00	37523
s3	–	–	–	58.48	20.00	44735
s4	–	–	–	45.87	15.00	39069
s5	–	–	–	17.79	10.00	20137
s6	29.69	5.00	48444	2.45	5.00	4736
s7	0.22	0.00	123	0.18	0.00	123
	std/exp/incr			std*/exp/incr		
b1	1.12	3.00	2415	0.72	3.16	1039
b2	9.31	6.43	15894	3.17	7.00	2891
b3	26.87	7.23	33131	9.90	7.69	4522
b4	30.53	7.03	37691	7.97	7.32	5219
b5	33.48	6.90	39373	8.50	7.12	5252
s1	70.48	3.28	50222	13.14	3.42	9376
s2	–	–	–	62.86	5.28	36484
s3	–	–	–	175.24	8.85	61411
s4	–	–	–	379.12	10.68	60145
s5	–	–	–	40.96	8.09	28388
s6	29.95	4.28	48444	2.99	4.28	5288
s7	0.33	0.00	123	0.31	0.00	123

Figure 5: Results for `basic` and `simple` (incr. algorithm)

those areas, we only keep the breaker and open switches, which leaves us with 7 breakers, 11 lines, and 11 switches. `small-rural` is further reduced by removing all breakers except `CB1`, `CB5` and `CB6` as well as the area attached to them, so are left with 3 breakers, 7 lines, and 6 switches. For both networks, we considered 8 problem instances with an increasing degree of uncertainty peaking at that of the original. These instances comprise 2, 4, 8, 24, 72, 216, 648, and 1944 initial states, respectively. Figure 7 shows the run time (sec.), optimal policy cost, and number of states generated by the incr. algorithm for mbp* and std* under the two optimization criteria. We used a time-shared server with a similar processor as before and 4GB of memory, only half of which GPT was allowed to use.

The incremental state generation and dynamic computation of h_{DYN} pays off again: the org. and $h = 0$ algorithms were unable to solve any of those instances within the allocated memory. Although GPT is still not able to solve the larger instances, it is worth noting that some of those it can solve are *big*. Even the subset of states generated with h_{DYN}

prob.	* version incr.	$h = 0$
	std*/exp	
l4_e0	0.30	0.29
l4_e2	0.64	3.07
l4_e4	0.30	0.30
l6_e0	0.29	0.27
l6_e2	8.37	139.80
l6_e4	3.34	177.88
l6_e6	0.29	0.28
l8_e0	0.35	0.28
l8_e2	151.65	3683.23
l8_e4	521.30	–
l8_e6	24.11	–
l8_e8	0.33	0.31

prob.	* version incr.	$h = 0$
	std*/exp	
s1	13.14	58.34
s2	62.86	1473.63
s3	175.24	12054.37
s4	379.12	14785.79
s5	40.96	3322.15
s6	2.99	194.88
s7	0.31	0.29

Figure 6: Run Times for incr. and $h = 0$ on Small Instances

grows near to a million, which is very signi£cant for optimal planning in a partially observable domain. It is clear that we are reaching the limits of what can be achieved with algorithms based on explicit state representations, and that the use of compact representations is the key to further improvement of these results. Symbolic representations are indeed one of the reasons behind the success of the MBP planner: it solved the original rural network instance in a few seconds following a 30mn compilation of the network description into ef£cient data structures. Another obvious reason behind the discrepancy between the run-times of MBP and GPT on similar problems is that GPT always optimizes cost. E.g., on small networks, this causes the mbp* domain to be only somewhat easier for GPT than its std* counterpart.

Conclusion, Future, and Related Work

We believe that the systematic (and even competitive) confrontation of realistic benchmarks by general-purpose planners is a key to further advance the £eld of contingent planning. This paper contributes to the demonstration of this idea, explaining how the PSR benchmark motivated the development of new techniques which are shown to increase the ability of the GPT planner to cope with larger problem instances. Our technical contribution to contingent planning, namely the incremental computation of an admissible domain-independent heuristic coinciding with h_{QMDP} in pure non-deterministic domains, is applicable in conjunction with virtually any explicit heuristic search algorithm in belief space, and is particularly useful when even the reachable state space is too large to be entirely explored. For all but the smallest PSR instances, the bene£ts of incremental computation are clear as the original version of GPT could not complete the precompilation step. Yet, we plan to do a comprehensive comparison of h_{QMDP} and h_{DYN} over other domains. (Bertoli & Cimatti 2002) presents another, perhaps better informed, heuristic for contingent planning which does not require the complete generation of the state space. However, it is not admissible and is therefore better suited to a hill-climbing search than to the search for an optimal solution.

By presenting a PDDL-like encoding of PSR, this paper contributes to its future as a benchmark. In particular, except perhaps for the reintroduction of axioms into PDDL, there is now no obstacle to PSR's featuring in a planning competition. Ours is the £rst published network-independent encoding: (Thiébaux & Cordier 2001) only provides an informal description, while the propositional $\mathcal{AR}$ encoding in

		small_rural								simplified_rural				
		rsm1	rsm2	rsm3	rsm4	rsm5	rsm6	rsm7	rsm8	rsp1	rsp2	rsp3	rsp4	rsp5
mbp*/wst/incr	time	0.59	2.74	5.42	24.57	108.54	583.38	3950.55	36196.09	8.56	240.88	481.05	9339.97	–
	cost	3.00	6.00	6.00	8.00	8.00	8.00	9.00	9.00	3.00	6.00	6.00	8.00	–
	states	457	2096	4099	10082	30322	91015	273386	822475	729	20673	38646	639947	–
mbp*/exp/incr	time	0.74	2.47	4.82	19.69	73.73	388.63	3153.21	81541.57	8.53	231.79	460.43	1922.22	15384.11
	cost	2.00	3.75	3.75	4.17	4.17	4.17	4.40	5.19	2.00	3.75	3.75	4.17	4.17
	states	457	1843	3679	9491	28666	84694	266034	822026	729	20103	37429	148129	890097
std*/wst/incr	time	1.26	4.97	9.04	28.81	114.49	574.59	7076.00	–	80.86	2039.79	3794.65	–	–
	cost	3.00	6.00	6.00	11.00	11.00	11.00	12.00	–	3.00	6.00	6.00	–	–
	states	1138	4519	8225	19880	61427	183577	615771	–	6929	148631	279134	–	–
std*/exp/incr	time	1.458	3.86	7.23	33.53	132.28	647.39	6602.78	–	82.20	1836.27	3352.00	–	–
	cost	2.00	3.75	3.75	5.83	5.83	5.83	6.06	–	2.00	3.75	3.75	–	–
	states	1138	3414	6473	19587	56668	168541	528050	–	6929	131668	243421	–	–

Figure 7: Results for `small_rural` and `simplified_rural` (incr. algorithm)

(Bertoli *et al.* 2002) is generated by a special-purpose procedure based on the analysis of the paths in the network of interest. In a sense, this makes our experiments on PSR the £rst 100% domain-independent ones, as knowledge of the structure of particular networks was not even used in the encoding, let alone in solving the problem. In the future, we plan to encode the numeric aspects of the benchmark (the constraints on the capacity of breakers and lines), and study the impact on planner behavior and optimal policies. A straightforward but also interesting variation would be to assign a cost not just to the £nal states of a plan, but also to intermediate states, and trade that extra cost against the need to gain information by discounting costs along trajectories.

Finally, this paper is the £rst to report the generation of *optimal* solutions for PSR – albeit still for rather small networks and in far longer than real-time. Special-purpose PSR software such as SYDRE achieves real-time performance on large networks (hundreds of devices and lines), but returns suboptimal policies which do not attempt to gain information (Thiébaux *et al.* 1996). We plan to compare the quality of the behaviors of GPT and SYDRE. Burton (Williams & Nayak 1997) is a general-purpose planner based on the compilation of a planning problem into a real-time executive, which has been applied to problems in spacecraft engine recon£guration close to PSR. However, Burton assumes total observability and deterministic actions: as in the SyDRe system, uncertainty in the problem is handled by a mode-identi£cation module, which determines the most likely state of the system and passes it onto Burton. The symbolic MBP planner was able to obtain extremely impressive results with a simpler version of PSR for which the supply restoration goal is given and optimization is not required. This leads to the conclusion that a combination of the features of GPT and MBP i.e., admissible heuristic search in the belief space, restriction to reachable (belief) states, and use of compact BDD-like representations to obviate the need for explicit state enumeration, could be a killer for PSR.

Acknowledgements Thanks to Erik Khoo and John Slaney for their valuable input to the PDDL encoding of PSR. Blai Bonet is supported by grants from NSF, ONR, AFOSR, DoD MURI program, and by a USB/CONICIT fellowship from Venezuela.

References

Barto, A.; Bardtke, S.; and Singh, S. 1995. Learning to act using real-time dynamic programming. *Art. Int.* 72:81–138.

Bertoli, P., and Cimatti, A. 2002. Improving heuristics for planning as search in belief space. In *AIPS*, 143–152.

Bertoli, P.; Cimatti, A.; Roveri, M.; and Traverso, P. 2001. Planning in nondeterministic domains under partial observability. In *IJCAI*, 473–478.

Bertoli, P.; Cimatti, A.; Slaney, J.; and Thiébaux, S. 2002. Solving power supply restoration problems with planning via symbolic model checking. In *ECAI*, 576–580.

Bonet, B., and Geffner, H. 2000. Planning with incomplete information as heuristic search in belief space. In *AIPS*, 52–61.

Bonet, B., and Geffner, H. 2001. Gpt: A tool for planning with uncertainty and partial information. In *IJCAI Workshop on Planning under Uncertainty and Incomplete Information*, 82–87.

Cassandra, A.; Kaelbling, L.; and Kurien, J. 1996. Acting under Uncertainty: Discrete Bayesian Models for Mobile-Robot Navigation. In *IROS-96*.

Geffner, H. 2000. Functional strips: a more ¤exible language for planning and problem solving. In Minker, J., ed., *Logic-Based Arti£cial Intelligence*. Kluwer.

Hansen, E., and Feng, Z. 2000. Dynamic programming for POMPDs using a factored state representation. In *AIPS*, 130–139.

Hansen, E., and Zilberstein, S. 2001. LAO*: A heuristic search algorithm that £nds solutions with loops. *Art. Int.* 129:35–62.

Karlsson, L. 2001. Conditional progressive planning under uncertainty. In *IJCAI*, 431–436.

Korf, R. 1985. Iterative-deepening A*: An optimal admissible tree search. In *IJCAI*, 1034–1036.

Majercik, S., and Littman, M. 1999. Contingent planning under uncertainty via stochastic satis£ability. In *AAAI*, 549–556.

McDermott, D. 1998. PDDL – The Planning Domain De£nition Language, Version 1.2. Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control.

Slate, D., and Atkin, L. 1977. CHESS 4.5 – the Northwestern University chess program. In Frey, P., ed., *Chess Skill in Man and Machine*. Springer-Verlag. 82–118.

Thiébaux, S., and Cordier, M.-O. 2001. Supply restoration in power distribution systems — a benchmark for planning under uncertainty. In *ECP*, 85–95.

Thiébaux, S.; Cordier, M.-O.; Jehl, O.; and Krivine, J.-P. 1996. Supply restoration in power distribution systems — a case study in integrating model-based diagnosis and repair planning. In *UAI*, 525–532.

Thiébaux, S.; Hoffmann, J.; and Nebel, B. 2003. In defense of PDDL axioms. Technical Report TR-ARP-01-03, ANU. `http://csl.anu.edu.au/~thiebaux/paper/trarp0103.pdf`.

Williams, B., and Nayak, P. 1997. A reactive planner for a model-based executive. In *IJCAI*, 1178–1185.