Extending an Estimated-Regression Planner for Multi-Agent Planning*

Drew V. McDermott
Yale University
Computer Science Department
drew.mcdermott@yale.edu

Mark H. Burstein
BBN Technologies
burstein@bbn.com

Abstract

We examine the issues that arise in extending an estimated-regression planner to find plans for *multiagent teams*, cooperating agents that take orders but do no planning themselves. An estimated-regression system is a classical planner that searches situation space, using as a heuristic numbers derived from a backward search through a simplified space, summarized in the *regression-match graph*. Extending the planner to work with multiagent teams requires it to cope with autonomous processes, and objective functions that go beyond the traditional step count. Although regressing through process descriptions is no more difficult than regressing through standard action descriptions, figuring out how good an action recommended by the regression-match graph really is requires projecting the subtree suggested by the action. We are in the process of implementing the algorithm.

Introduction

One important type of multi-agent planning is concerned with the development of activity plans for collections of agents, either automatically using a planning algorithm or interactively through a dialog between planning agent(s) and user(s). Our long-term interest is in the latter, that is, mixed-initiative approaches to agent team tasking, but we start with a purely autonomous model in which the planning algorithm does all the work.

This paper focuses on the extensions to a classical planner which are required to support generation of plans for coordinated teams of agents. A *classical* planner is one based on these assumptions:

1. Nothing happens except the actions the planning agent takes. We use the term *planning agent* to mean the agent for which the plan is being constructed.[1]
2. The planning agent takes just one action at a time.
3. The planner's model of the world is complete and accurate, so that it knows exactly what is true before any action is taken, and knows all the consequences of any sequence of actions the planning agent might take.

A classical *problem* is an initial situation plus a proposition (the "goal") to be made true by a series of actions. If we care about the cost of alternative plans, that cost is measured by adding up the costs of the steps in a plan (often just taken to be 1).[2]

Under these assumptions, it doesn't matter how long each action takes; even if the time is counted as part of the step cost, it's just a number, not an interval over which anything interesting can happen.

We will be examining the problem of constructing plans for multiagent teams. We will assume that the agents are cooperating with each other completely. Under these circumstances, the planner can construct a plan in which the individual agents are treated as extensions ("effectors") of the planning agent.

Planning for multi-agent teams requires several extensions to the classical framework:

- Actions of team members can go on in parallel and finish at different times. From the point of view of the planner, the actions of team members look like *autonomous processes*, that is, sequences of events that go on without further intervention by the planning agent.
- Preconditions on plan steps for one team meember may be established by actions not under its control, so agents may be required to *wait* for processes they or someone else initiated to complete before acting themselves.
- Plans must be developed and executed with incomplete knowledge. In agent plans, there are typically messages to other agents to request or provide information, or to directly synchronize activities (a special case of information provision). Messages and their replies can be viewed as single actions that *produce values*, and lead to particular agents *knowing values*. Although we are hopeful that the mechanisms of (McDermott 2002) can be extended to this domain, we will not discuss how that might be done, except for some speculations at the end of this paper.
- It is no longer reasonable to measure the utility of a plan by counting the number of steps. A typical step is "Start

*This work was supported by DARPA/CoABS under contract number F30602-98-0168.

[1] A more accurate term might be "execution agent" or "plan beneficiary," but it's simpler and easier on the ear to pretend that there exists an agent making plans for itself.

[2] Exactly how the step costs are counted is a subtle issue that needn't concern us (Blum & Furst 1997).

agent 86 going to Malta." However we measure its cost (the cost of sending a radio message to agent 86?), the real cost is the resources expended in getting to Malta, plus whatever costs are incurred on other agents' behavior by sending agent 86 to Malta instead of agent 99. We can measure plan utility, by the total time to execute the plan, or the total resources expended, but in general utility can be an arbitrary function of its execution trace.

We will assume that concurrency is *interleaved* (Chandy & Misra 1988; Roscoe 1998), in that no two actions occur at exactly the same time. If several agents are ordered to Malta at the beginning of the plan, we model that by assigning an arbitrary sequence to the agents' actions and treating consecutive actions as separated by an infinitesimal slice of time. Remember that this applies only to the issuing of the orders; the trips to Malta occur in parallel, and take different times for different agents.

We produce a multiagent plan by extending an *estimated-regression* (ER) planner, which is a situation-space planner guided by a heuristic estimator obtained by doing a simplified search back from the *end goal* to the current situation. The result of the search is a graph of goals and subgoals, whose *reduction trees* represent sketches of ways to complete the plan. Planners in this family include Unpop (McDermott 1996), Optop (McDermott 2002) and the HSP variants (Bonet, Loerincs, & Geffner 1997; Geffner 1998; Bonet & Geffner 2001). Here we will focus on the Unpop-Optop family, using the term "Unpop" as a generic term to cover all of them. Unpop differs from many recent planners in that it does not replace first-order formulas with all their possible instances before planning begins. As a result, at every step it must reason about how to substitute for the variables. This feature makes it slower than other algorithms in finite domains, but makes it able (in principle) to handle domains involving infinite domains such as numbers. This ability is crucial in modeling autonomous processes. In the rest of this paper we will explain how ER planners work, and how to extend them to the multiagent-team context.

Estimated-Regression Planning

Unpop is a "state-space" planner. More precisely, the space it searches is a space of *plan prefixes*, each of which is just a sequence of steps, feasible starting in the initial situation, that might be extendable to a solution plan. In the context of considering a particular plan prefix, the term *current situation* will be used to refer to the situation that would obtain if these steps were to be executed starting in the initial situation. Each search state consists of:

1. a plan prefix P
2. the resulting current situation
3. a score, equal to the cost of the steps in P + the *estimated completion effort* for P, which is an estimate of the cost of the further steps that will be needed to get from the current situation to a situation in which the goal is true.

The estimated completion effort for P is obtained by constructing a *regression-match* graph, which can be considered a "subgoal tree with loops." The nodes of the tree are divided into *goal nodes* and *reduction nodes*. A goal node G is a literal to be made true. If G is not true in the current situation, then below it are zero or more reduction nodes, each corresponding to an action that might achieve it. A reduction node is a triple $\langle G, M, \{P_1, \ldots, P_k\}\rangle$, where G is a goal node, M is an action term, and each P_i is a literal, called a *subgoal*. It is usually the case that goal nodes and subgoals have no variables, and that in a reduction node the conjunction of the P_i are sufficient to make M feasible and to make G one of M's effects. There is a *reduction edge* from any goal node G to every reduction node with G as its first component, and a *link edge* between any reduction node whose third component includes P_i and the goal node P_i.

The structure is not a tree because there can be multiple paths from one goal node to another through different reductions, and there can be a path from a goal node to itself. What we are interested in are cycle-free subgraphs, called "reduction trees," because they correspond to (apparently) feasible ways of achieving goals. The size of the subgraph below a goal node gives an estimate of the difficulty of achieving it. The estimate neglects step ordering, and destructive and constructive interactions among steps, but taking those phenomena into account is an exponential process, whereas the regression-match graph tends to be of polynomial size (as a function of the size of the problem and the size of the solution).

Unpop avoids variables in goal nodes by its treatment of conjunctive goals. Given a conjunctive goal $H_1 \wedge \ldots \wedge H_n$, Unpop finds maximal matches between it and the current situation, defined as substitutions that, roughly, eliminate all the variables while making as many of the H_i true in the current situation as possible (McDermott 1999). However, in domains with numbers, an unsatisfied H_i may have to be left with a variable. In that case, the exact meaning of a reduction node becomes slightly harder to state. (McDermott 2002)

Unpop builds the regression-match graph by creating an artificial goal node `top` as the root of the "tree," then maximally matching the end goal to the current situation, producing a set of reduction nodes, each of the form $\langle \texttt{top}, \texttt{done}, \{P_i\}\rangle$. It then examines every P_i in every reduction node, finds actions that would achieve it, and maximally matches their preconditions to produce reduction nodes for P_i. The process normally just keeps running until no more goal nodes have been generated, although there is a depth limit on the graph to avoid runaway recursions in perverse domains.

Having generated the graph, its "leaves" are reduction nodes all of whose subgoals are true in the current situation. We call these *feasible* reduction nodes. The action of such a reduction node, called a *recommended action*, is feasible in the current situation, and is therefore a candidate for extending the current plan prefix.

We assign *completion effort estimates* to every node of the graph by assigning effort 0 to feasible reduction nodes, assigning effort ∞ to all other nodes, then recursively updating the efforts according to the rules:

$$\mathit{eff}(G) = \begin{cases} 0 \\ \quad \text{if } G \text{ is true} \\ \quad \text{in the current situation} \\ \min_{R \in \mathit{reductions}(G)} \mathit{eff}(R) \\ \quad \text{otherwise} \end{cases}$$

$$\mathit{eff}(\langle G, M, \{P_i\}\rangle) = 1 + \sum_i \mathit{eff}(P_i)$$

until all the values have stabilized.

We are interested in extracting subgraphs *reduction trees* from the regression-match graph. A reduction tree is a rooted tree, whose root, which coincides with the artifical `top` node, we will call $E_{\mathtt{top}}$. Each node E of the tree has a goal $G(E)$ and a reduction $R(E)$. If $G(E)$ is true in the current situation, then $R(E) = \phi$; otherwise, for some reduction node $\langle G, M, \{P_1, \ldots, P_k\}\rangle$ in the regression-match graph, $R(E) = \langle G, M, \{E_1, \ldots, E_k\}\rangle$, where $G(E_i) = P_i$. The completion effort estimate for a reduction tree is defined by

$$\mathit{eff}(E) = \begin{cases} 0 \\ \quad \text{if } G(E) \text{ is true} \\ \quad \text{in the current situation} \\ \mathit{eff}(R(E)) \\ \quad \text{otherwise} \end{cases}$$

$$\mathit{eff}(\langle G, M, \{E_i\}\rangle) = 1 + \sum_i \mathit{eff}(E_i)$$

An *effective reduction tree* of the regression-match graph is a reduction tree obtained by choosing the reduction node for $G(E)$ that achieves the minimum estimated completion effort. What we are really interested in are the reduction trees for recommended actions. A *reduction tree for* recommended action A is a reduction tree that includes A in a feasible node. A *minimal reduction tree for* A is a reduction tree for A that has minimal completion effort estimate.

As an example, consider a simple blocks world with one action `(move` x y`)`. The initial situation has `(on C A)`, `(on A table)`, and `(on B table)`. The goal is `(and (on A B) (on B C))`. The reduction nodes of the regression-match graph are:

```
⟨top, done, {(on A B), (on B C)}⟩[2]
⟨(on A B), (move A B), {(clear A)}⟩[2]
⟨(clear A), (move C table), {}⟩[1]
⟨(on B C), (move B C), {}⟩[1]
```

The numbers in brackets are the completion-effort estimates. The recommended actions are `(move B C)` and `(move C table)`. The following tree is a reduction tree for both actions:

```
(on A B)
    Do (move A B)
          (clear A)
             Do (move C table)
(on B C)
    Do (move B C)
```

Unpop will try both recommended actions, and will realize that `(move B C)` is a mistake as soon as it recomputes the regression-match graph on the next iteration of its outer search loop.

Processes and Objective Functions

We now describe the extensions to Unpop to handle autonomous processes and objective functions. Both rely on the concept of *fluent*, a term whose value varies from situation to situation. Fluents have been in PDDL from the beginning, but are just now beginning to be noticed, notably in the AIPS02 competition (Fox & Long 2001). The value of a fluent can be of any type, but we assume from here on that they all have numerical values, some integer (the *discrete* fluents), some floating-point (the *continuous* fluents).

We formalize a *process* as an action-like entity which has a `:condition` field instead of a `:precondition`; whenever the condition field is true, the process is *active*. The `:effect` field is altered with a new kind of effect:

`(derivative` q d`)`

which means that the derivative of continuous fluent q is d. Although "derivative" sounds like we might be able to reason about arbitrary differential equations, for now we will assume that d is always a constant, i.e., that all fluent changes are linear.

An *objective function* is a measurement of the cost of a plan, a value to be minimized. We will take this to be the value of some given fluent in the situation when the end goal is achieved. It turns out that the extensions to Unpop to handle objective functions and those to handle autonomous processes are closely related. Both extensions require the planner to abandon "step count" as a measure of the cost of a plan. We can still use it as a crude heuristic for managing the regression-match graph, but once the planner has found a feasible action, it must extract from the graph an "expected continuation" of the plan, and use that to generate a more precise estimate of the value of the action.

Let's look at an example. (See figure 1, which omits some details of the actual domain specifications.) Our agent teams consist of cats whose job is to find mice by putting a set of rooms under surveillance. Each cat can monitor a room by standing on a platform of the correct height for that room. The platforms may have to be built up from smaller platforms to reach the correct height. height. The floor of a room is a platform of height 0. Cats can climb onto a platform if they are on a platform adjacent to it and the height difference between the platforms is not too great (0.5 meter). Otherwise, they need to find or build a "step" next to the platform to get up onto it. Some cats are "builders", who can raise the height of a platform one meter every 100 seconds. To get from room to room, a cat must "travel" at the rate of one meter per second. The objective function is to minimize the time required to get a set of rooms under surveillance.

Clearly, a solution to a problem in this domain will consist of issuing a set of orders to available cats. Builder cats will

```
(define (domain agent-teams)
   (:types Agent Order)
   (:predicates
       (told ?a - Agent ?r - Order))
   (:action
        (tell ?a - Agent ?r - Order)
    :effect
       (and (forall (?r-old - Order)
                (when (told ?a ?r-old)
                   (not (told ?a ?r-old))))
            (told ?a ?r)))

(define (domain cat-and-mouse)
   (:extends agent-teams)
   (:types Room Platform - Object
           Cat - Agent
           Floor - Platform)

   (:variables
       ;; max height a cat can build
       (build-increment 0.5) ;; m
       (build-rate 0.01) ;; m/sec
       (cat-speed 1) ;; m/sec
       - Number)

   (:predicates
       (floor-of ?f - Floor ?r - Room)
       (connected ?rm1 ?rm2 - Room)
       (in ?a - Animal ?rm - Room)
       (on ?a - Object ?p - Platform)
       (nextto ?b1 ?b2 - Platform)
       (builder ?a - Cat)
       (surveillance-height ?rm - Room)
       (under-surveillance ?rm - Room))

   (:functions (height ?p - Platform)
               - (Fluent Number)

               (to-build ?p - Platform
                         ?height - Number)
               (to-go ?rm1 ?rm2 - Room)
                 - Order)
   (:fact
      (<- (under-surveillance ?rm)
          (exists (c - Cat h - Number
                   p - Platform)
             (and (surveillance-height ?rm h)
                  (in c ?rm)
                  (on c p)
                  (>= (height p) h)))))

(:process (building ?a - Cat
                    ?p - Platform)
    (:vars ?current-loc - Platform
           ?h - Number)
    (:condition
        (and (builder ?a)
             (told ?a (to-build ?p ?h))
             (on ?a ?current-loc)
             (or (on ?p ?current-loc)
                 (nextto ?p ?current-loc))
             (=< (height ?p)
             (+ (height ?current-loc)
                build-increment))))
    (:effect (and (under-construction ?p)
                  (derivative (height ?p)
                              build-rate))))

(:process (traveling ?a - Cat
                     ?rm1 ?rm2 - Room)
   (:condition
       (and (connected ?rm1 ?rm2)
            (told ?a (to-go ?rm1 ?rm2))
            (>= (dist ?a ?rm2) 0)
            (or (in ?a ?rm1)
                (in ?a ?rm2))))
   (:effect
      (and (derivative (dist ?a ?rm2)
                       (- cat-speed))
           (derivative (dist ?a ?rm1)
                       cat-speed)
           (forall (pl - Platform)
              (when (and (in pl ?rm)
                         (not (floor-of
                                pl ?rm)))
                 (not (on ?a pl))))
           (when (< (dist ?a ?rm2)
                    (dist ?a ?rm1))
              (and (in ?a ?rm2)
                   (not (in ?a rm1)))))))

(:action (move ?a - Cat ?p - Platform)
   (:vars ?current-loc - Platform)
   (:precondition
       (and (on ?a ?current-loc)
            (not (under-construction ?p))
            (or (nextto ?current-loc ?p)
                (on ?p ?current-loc)))
            (>= (height ?current-loc)
                (- (height ?p)
                   build-increment)))
   (:effect (and (not (on ?a ?current-loc))
                 (on ?a ?p)))))
```

Figure 1: Cat-and-mouse domain

be told to build platforms of appropriate sizes, and available cats will be sent to climb those platforms, so that all rooms are under surveillance. If builder cats are in short supply, they should be used for building, not surveillance.

The first step in getting an ER planner to handle processes is to get it to regress through process descriptions. Given a

goal of the form `(>= (height p13) 3)`, it sees that the process instance `(building a p13)` would change the derivative of `(height p13)`, where a is as yet unchosen. To make this instance active requires that all of the following be achieved;

```
* (builder a)
*(told a (to-build p13 h))
*(on a l)
*(or (on p13 l) (nextto p13 l))
*(=< (height p13)
     (+ (height l) build-increment))
```

Unpop, as usual, tries to find an a, l, and h that make as many of these conjuncts true as possible. Typically, it will consider, for every builder cat, issuing an order for it to build `p13` higher. This will generate subgoals that involve moving a builder to platforms adjacent to `p13`. If there is no adjacent location high enough, Unpop will regress through the `building` definition again, looking for a spot that could be used to build up an adjacent spot.

To take a specific example, suppose we have two cats, `cat1` and `cat2`. `cat2` is a builder; `cat1` is not. They are both initially in `room1`, whose surveillance height is 1m; the goal is to get both `room1` and `room2` under surveillance. `room2`'s surveillance height is 0m. There are three platforms, the floors `fl1` and `fl2` of the two rooms plus a buildable site `pl1` in `Room1` with an initial height of 0m.

Glossing over many details, we arrive at a regression-match graph containing the following reduction tree (The recommended actions are tagged with a star.)

```
(under-surveillance room1)
    Do (move cat2 pl1)
          (surveillance-height room1 1)
          (in cat2 room1)
          (on cat2 floor1)
          (not (under-construction pl1))
          (on pl1 floor1)
          (>= (height floor1)
              (- (height pl1) 1))
          (>= (height pl1) 1)
              Do Wait until (>= (height pl1) 1)
                    (builder cat2)
                    (told cat2 (to-build pl1 1))
                        *Do (tell cat2
                                (to-build pl1 1))
                    (on cat2 floor1)
          (on pl1 floor1)
          (=< (height pl1) 1)
(under-surveillance room2)
    Do Wait until (< (dist cat2 room2)
                     (dist cat2 room1))
          (surveillance-height room2 0)
          (connected room1 room2)
          (told cat1 (to-go room1 room2))
              *Do (tell cat1
                        (to-go room1 room2))
          (>= (dist cat2 room2) 0)
          (in cat2 room1)
```

The step count for this subgraph is the number of *Do* nodes, or 5, but that doesn't tell us what we want to know, especially since two of them are *Wait*s. We must go one step further, and, for each recommended action derive a *plausible projection* from a minimal reduction tree containing it. A plausible projection is obtained by "projecting" (simulating the execution of) the reduction tree bottom up, starting with the recommended action. At any point in this process there is a set of *fringe actions*, meaning those that have not been projected yet, but whose predecessors have been. The planner chooses any non-*Wait* action to project; if there aren't any, it simulates the passage of time until something "interesting" occurs, then checks to see if any of the events being waited for is one of the things that happened. Whether a *Wait* or non-*Wait* is projected, the hoped-for result is to change the set of fringe actions. The process repeats until the top of the reduction tree is reached.

The resulting projection can then be used to estimate the cost of executing the recommended action next. It gives us a possible final situation, and the objective function is some arbitrary fluent, so we evaluate it and see what number we get out.

Obviously, we have glossed over several difficulties. One is easy to deal with: What does the planner do if a fringe action is redundant, that is, if the goal it is meant to achieve is already true at the point where it is to be projected? The answer is: it doesn't bother to project the step at all.

The second is a bit harder: What does the planner do if a fringe action is not feasible? An arbitrary linearization of the remaining steps of a plan is all too likely to contain infeasible steps, but the planner still has to be able to project the plan sequel in a "reasonable" way. Suppose the purpose of fringe action S is to make a subgoal G true. G must be false in the current situation, or else the regression-match graph would have guessed it cost 0 to make true. We now discover that the step intended to make it true won't do that, at least not if executed in the semi-random way we are examining. But it's reasonable to go ahead and assert the usual effects of S, including G, so that the feasibility of later steps isn't unduly affected by the failure of S. We have to decide how to penalize a projection that contains an infeasible step. Although there are various possibilities, we opt for charging a simple percentage penalty for every infeasible step.

The most interesting issue is how the planner simulates the passage of time. In every situation, there is a set of *active process instances*, those whose `:conditions` are true. The *trigger point* for this set of processes is the next point in time at which either the active-process set will change, or some effect of an active process will occur. (The importance of the idea of next trigger point has been emphasized by (Atkin, Westbrook, & Cohen 1999), albeit in a more specialized context.) These are all the changes that can occur because of a change in the truth value of an inequality in a process definition, either inequalities in the `:condition` of a process instance, or inequalities in the `:effect` of an active process instance. It is easy to project the passage of time until the next trigger point, because the only changes are smooth changes in numerical quantities, and impossible to project beyond it without stopping to add effects of in-

equalities changing truth value, and recomputing the active process set.

Hence what the planner must do, when every fringe action is a *Wait*, is simulate the passage of time until one of the waited-for propositions becomes true, or until it runs out of patience. It should run out of patience if no progress is being made toward any of its subgoals, but we don't have the space to discuss this issue in detail here.

"Delinearizing" a Plan

Like all classical planners, Unpop, even with the enhancements described, produces a linear sequence of actions as a solution to a planning problem. For many purposes, it is clearer to produce a set of plans for each agent. This is especially useful if the agents really are autonomous actors (say, submarine commanders) who need to understand their part in the overall plan and exactly what sequence of operations is expected of them.

"Delinearizing" a plan is not terribly difficult. Every non-*Wait* step must mention an agent as actor, as in `(move` *agent* p_1 p_2`)` or `(tell` *agent* c`)`. Extracting the plan for agent a means finding all the steps that mention a as actor. Some of the steps have preconditions that are achieved by *Waits* or by actions taken by other agents. These are incorporated into the agent's plan as notes about exactly what to wait for.

For instance, if there is just one builder cat, and three observer cats dependent on its services, the builder's plan might look like: "Move to `pl21`; begin building `pl62`; wait for it be complete, at time 100; begin traveling to `room23`; wait until you are on `pl23f` (the floor of `room23`), at time 110; move to `pl103`; begin building ...." An observer's plan might look like "Begin traveling to `room23`; wait until you are in `room23`, at time 20; wait until `builder1` has built `pl103` a meter high, at time 240; move to `pl103`; ...."

Things get much more interesting if plans include information-gathering steps, an issue we discuss in the next section.

Status and Conclusions

This is clearly a work in progress. At this time, we are nearly done with a first implementation. Only then we will know whether some of the heuristics we have suggested will actually work. Possible pitfalls include an explosion in the number of inequalities produced by regression, and inefficiency or inaccuracy in projecting reduction subtrees with *Waits*.

A key issue for future work is getting the devices described here to work with the mechanisms for reasoning about information gathering described in (McDermott 2002). We assumed at the outset that the agents' actions are entirely at the service of the planner's goals, which means that they are in a sense "effectors" of the planning agent, no more autonomous than one's hands or eyes. But in fact, the planning agent is not a real entity, in the sense that once each agent has been given its marching orders, there is nothing the planning agent does separate from what they do. In particular, there is no notion of the planning agent knowing or learning a fact. If an action has a knowledge precondition, then that precondition will have to be associated with the particular effector agents that actually need the knowledge. If a cat needs to know that a door is open before it can go through it, there might be two ways for it to find out: by going to the door, or by asking another cat that knows and getting a useful response. The planner might choose the latter course, choosing a specific cat whose route takes it past the door and then past the cat with the knowledge gap.

"Delinearizing" a plan in which effector agents communicate presents a few extra challenges. In the original plan, we'll have a step like `(tell cat23 cat17 open-status door24)`, followed by conditional steps such as `(if (open door24) (go cat17 ...) ...)`. We have to make sure that "agency" is correctly allocated to the various pieces of the plan, so that `cat23`'s plan comes out thus:

```
...; (go room5);
     (tell cat17 open-status door24); ...
```

and `cat17`'s plan comes out thus:

```
...; (wait-event
         (tell cat23 cat17
               open-status door24));
     (if (open door24)
         ...
         ...)
```

References

Atkin, M.; Westbrook, D.; and Cohen, P. R. 1999. Capture the Flag: Military simulation meets computer games. In *Proceedings of AAAI Spring Symposium Series on AI and Computer Games*, 1–5.

Blum, A. L., and Furst, M. L. 1997. Fast planning through planning graph analysis. *Artificial Intelligence 1–2* 90:279–298.

Bonet, B., and Geffner, H. 2001. Planning as heuristic search. *Artificial Intelligence* 129(1-2).

Bonet, B.; Loerincs, G.; and Geffner, H. 1997. A fast and robust action selection mechanism for planning. In *Proc. AAAI-97*.

Chandy, K. M., and Misra, J. 1988. *Parallel program design : a foundation*. Addison-Wesley.

Fox, M., and Long, D. 2001. Pddl. 2.1: An Extension to PDDL for Expressing Temporal Planning Domains available at `http://www.dur.ac.uk/d.p.long/pddl2.ps.gz`.

Geffner, H. 1998. Classical, probabilistic, and contingent planning: three models, one algorithm. Workshop on Planning as Combinatorial Search: Propositional, Graph-Based, and Disjunctive Planning Methods.

McDermott, D. 1996. A Heuristic Estimator for Means-ends Analysis in Planning. In *Proc. International Conference on AI Planning Systems*, 142–149.

McDermott, D. 1999. Using Regression-match Graphs to Control Search in Planning. *Artificial Intelligence* 109(1–2):111–159.

McDermott, D. 2002. Estimated-regression planning for interactions with web services. In *Proc. AI Planning Systems Conference 2002*. To appear.

Roscoe, A. 1998. *The theory and practice of concurrency*. Prentice Hall.