Solving POMDPs online through HTN Planning and Monte Carlo Tree Search

Robert P. Goldman
SIFT, LLC
319 1st Ave N., Suite 400,
Minneapolis, MN 55401, USA
rpgoldman@sift.net

Abstract

This paper describes our SHOPPINGSPREE HTN algorithm for online planning in Partially Observable Markov Decision Processes (POMDPs). SHOPPINGSPREE combines the HTN planning algorithm from SHOP3, extensions to SHOP3's representation to handle partial observability, and Monte Carlo Tree Search for efficient sampling in the problem space. This paper presents only the algorithm and initial notes on the implementation: this is work in progress.

1 Introduction

In this paper we describe SHOPPINGSPREE (named after a US TV game show and SHOP3), a technique for solving Partially Observable Markov Decision Processes (POMDPs) in an online fashion – that is, interleaving planning and execution – based on Monte Carlo Tree Search (MCTS), currently under development, building on the Hierarchical Task Network (HTN) planner, SHOP3. We describe a complete algorithm, but the implementation is still in very early stages: its implementation is still messy, made up primarily of patches to the existing version of SHOP3.However, the algorithm is promising because it enables us to combine HTN planning with sequential decision problems.

The key tenets of our approach are as follows: 1. We approach POMDPs as *games against nature*, where the system's objective is to execute an (approximately) optimal strategy against its environment. 2. The planning agent's "turns" in this game are the set of actions taken between observations. 3. The planning agent plans turn-by-turn ("online"), allowing us to avoid computing full policies, which can be exponentially large. 4. If myopic, turn-by-turn planning can be severely suboptimal. We use MCTS to provide lookahead and avoid myopic decision making. 5. To apply MCTS to POMDP planning, we combine planning in belief space with sampling in the base state space.

In this paper we briefly introduce POMDPs and the textbook "oil wildcatter" sequential decision problem. Again briefly, we summarize MCTS. Then we explain the extensions to SHOP3's Knowledge Representation (KR), and how to use it to represent a POMDP. Finally, we present our method for integrating HTN planning and MCTS, and conclude with notes on some limitations, mention of related work, and future directions.

A quick definition before we begin:

Definition 1 (POMDP) *A POMDP, $\mathcal{P} = \langle \mathcal{S}, s, \mathcal{A}, \mathcal{T}, \mathcal{R}, \Omega, \mathcal{O} \rangle$ where 1. $\mathcal{S}$ is a finite set of states, $s \in \mathcal{S}$ is the initial state;$\mathcal{A}$ is a finite set of actions; 2. $\mathcal{T} : \mathcal{S} \times \mathcal{A} \rightarrow \Pi(\mathcal{S})$ is the state-transition function, assigning a probability distribution over successor states when an action, a is executed in state s; 3. $\mathcal{R}$ is the reward function, which may be defined over $\mathcal{S} \times \mathcal{A}$, and defines the reward, a real number reward received when a is executed in s. The reward of a finite trace is the sum of the rewards at each step in the trace. Equivalently, we use a cost function, in the work described here. 4. Ω is a set of observations that the agent may make; 5. $\mathcal{O} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow \Pi(\Omega)$ is the observation function, which gives a probability distribution over the set of observations that may be received when the agent takes action a in state s_0 and the successor state (dictated by $\mathcal{T}$) is s_1 (Kaelbling, Littman, and Cassandra 1998, adapted).*

There are important subtypes of POMDP varying by time horizon. We address **only** *finite horizon* POMDPs.

A POMDP's solution is a *policy*: a mapping from belief states to action choices. We define an *optimal* policy as a policy that provides maximum expected utility. Note that the policy need not be explicitly computed. One way to think about POMDPs is as games against "nature," a stochastic opponent. That is the perspective we take here. Rather than computing a policy off-line and executing it, SHOPPINGSPREE computes its policy implicitly and on-line.

As a running example, we use the textbook "oil-wildcatter" decision problem (Raiffa 1968): The oil wildcatter needs to decide whether to drill or not. They don't know if their hole is **dry**, **wet**, or **soaking**. Their payoffs are given in Table 1. All payoffs are *net*: profit less \$70,000 for drilling, if the wildcatter chooses to. For \$10,000, the wildcatter can test the site's geological structure: no structure (NS), open (OS), or closed structure (CS). The structure is correlated with the likelihood that oil is present (Table 2). This problem may be formulated as a POMDP where $\mathcal{S} = \{\text{Dry}, \text{Wet}, \text{Soaking}\}$; $\Omega = \{\text{NS}, \text{OS}, \text{CS}\}$, and $\mathcal{A} = \{\text{test}, \text{drill}\}$, and $\mathcal{T}$ and $\mathcal{R}$ are from Tables 1 and 2. Expected rewards for some policies are given in Table 3.

2 Monte Carlo Tree Search (MCTS)

MCTS is a high-performance search method originally developed for game playing for games with extremely large

	Act	
State	a_1	a_2
Dry (θ_1)	- \$70,000	0
Wet (θ_2)	\$50,000	0
Soaking (θ_3)	\$200,000	0

Table 1: Monetary Payoffs for Oil Wildcatter problem (Raiffa 1968, pp. 35).

	Seismic Outcome			Marginal Probability
State	NS	OS	CS	of state
Dry (θ_1)	.300	.150	.050	.500
Wet (θ_2)	.090	.120	.090	.300
Soaking (θ_3)	.020	.080	.100	.200
Marginal probability of seismic outcome	.410	.350	.240	1.000

Table 2: Joint and marginal probabilities for Oil Wildcatter problem (Raiffa 1968, pp. 35).

state spaces, such as Go. Our discussion here is heavily indebted to the excellent survey by Browne et al.(2012). Briefly, the job of MCTS is to estimate the value of each node in the top of the search tree, so that an agent can choose its actions by greedily taking the highest value choice. To do this, the algorithm must search the tree so that it can accurately estimate the value of early decisions on the eventual outcome (*e.g.*, estimate the value of an opening move in terms of its effect on the eventual outcome of a game). To search the tree, the algorithm must have a way of choosing a child at every node of the tree. MCTS splits its search into two phases: first by *tree policy*, then by *default policy*.

In large search spaces, states near the root of the search tree will be explored according to the tree policy, but since the search space prohibits complete exploration, when search reaches some depth threshold, the system will switch to the default policy. The tree policy is generally a choice that weights known estimates of child nodes – causing the system to *exploit* its knowledge of where value is to be found – against a term that weights parts of the tree that have not been visited often – causing the system to *explore* new parts of the search space. We use the popular UCT (Upper Confidence Bounds for Trees) (Kocsis and Szepesvári 2006) rule:

$$\arg\min_{a \in A} \frac{V(a)}{N(a)} + k\sqrt{\frac{2 \ln N}{N(a)}} \tag{1}$$

where $V(a)$ is the mean value of node a, $N(a)$ is the visit count, and N is the visit count of the parent of a, and k is a constant. Currently, we use the original $k \equiv 1/\sqrt(2)$ (Kocsis and Szepesvári 2006). Depending on problem, other policy rules may be better than UCT and, in UCT, different values of k may be better. We leave this for future work.

The default policy will be some extremely cheap, largely random choice rule. At present, since we are working with small problems, ShoppingSpree has no default policy.

Algorithm 1 General MCTS approach, reproduced from (Browne et al. 2012)

```
1: function MCTSSEARCH($s_0$)
2:     create root node $v_0$ with state $s_0$
3:     while within computational budget do
4:         $v_l \leftarrow$ TREEPOLICY($v_0$)
5:         $\Delta \leftarrow$ DEFAULTPOLICY($s(v_l)$)
6:         BACKUP($v_l, \Delta$)
7:     end while
8:     return $a$(BESTCHILD($v_0$))
9: end function
```

When a "rollout" has been completed by reaching a leaf node of the search tree, the value encountered at the leaf will be *backpropagated* to update the estimates of the value of nodes nearer the root of the tree, and their visit counts.

Finally, when some resource threshold is reached, the system will choose an action (or set of actions) to take, based on the highest expected value. In a game – including a game against nature like our oil wildcatter problem – thee the MCTS system will wait to receive the opponent's move (*e.g.*, the results of the seismic test), and then repeat the process. The process is summarized in Algorithm 1.

3 Knowledge Representation (KR)

The current version of ShoppingSpree makesonly minimal extensions to the Shop3 KR for domains and problems: hidden propositions, and uncontrollable actions and methods. Other than this, we use standard Shop3 (Goldman and Kuter 2019) notation, which we review below.

Shop3 primitive operators have a **head**, comprising an **operator name** and **parameters**. They also have **preconditions**, **add-list**, **delete-list**, and **cost function**. The add-list and delete-list are lists of literals, potentially containing parameter variables. The preconditions are more expressive than Strips or Pddl: supporting logical operators, finite quantification, and the invocation of arbitrary functions. We will make use of this expressive power below. The cost function can be an arbitrary function of the parameters and any variables bound in its preconditions. An example operator from the oil-wildcatter domain:

```
(:op (!do-drill)
  :precond (not (drilled))
  :add ((drilled))
  :delete ()
  :cost 70)
```

Shop3 method definitions have a **head**, and **preconditions** as above and a **task network**. Note: variables in the head *and* in the preconditions are scoped over the task network. Task networks are made up of tasks and `:ordered` and `:unordered` networks. For example:

```
(:method (make-decisions)
 () ; empty preconditions
 (:ordered (decide-test)
           (decide-drill)
           (profit)))
```

Policy	Details	Expected Value	Rescaled EV
Don't test, don't drill	1×0	0	0.286
Don't test, drill	$0.5 \times -70 + 0.3 \times 50 + 0.2 * 200$	20	0.357
Test, drill iff CS	$-10 + (.09 \times 50) + (.1 \times 200)$	14.5	0.338
Test, drill if OS or CS	$-10 + (.09 \times 50) + (.1 \times 200) + (.12 \times 50) + (.08 \times 200)$	**36.5**	**0.416**

Table 3: Rewards for some policies, in thousands of dollars, and rescaled to between 0 and 1.

The above definition states that the `(make-decisions)` task can be rewritten into the task network, unconditionally, since its preconditions are empty.

SHOP3 domains may also have **axioms**, Prolog-style Horn clauses that are used when checking preconditions. The following axiom is used to compute the profit (`?p`) from drilling when the oil condition is `?o`:

```
(:- (drill-profit ?o ?p)
   ((= ?o dry) (= ?p 0))
   ((= ?o wet) (= ?p 120))
   ((= ?o soaking) (= ?p 270)))
```

SHOP3 permits a compressed representation allowing multiple Horn clauses with the same head (here `(drill-profit ?o ?p)`) to be combined.

Problems contain an initial state and task network:

```
(defproblem wildcatter ; problem name
  wildcatter ; planning domain name
  () ; initial state (empty)
  (oil-wildcatter)) ; initial task network
```

A SHOP3 planning problem is *solved* when its task network has been fully rewritten into a sequence of primitive actions that is executable (each action's preconditions are satisfied when it is executed). We will see, though, that generating plans is only a *subproblem* for SHOPPINGSPREE.

We have made minimal extensions to SHOP3's KR to model finite-duration POMDPs: 1. We allow propositions to be marked as `hidden` 2. We mark some methods as **stochastic**: these methods are allowed to read `hidden` propositions and invoke `random` in their preconditions. 3. We mark some operators as **uncontrolled**; these may read `hidden` propositions in their preconditions, and do not appear in the agent's history (see Section 4).

Here is an example of how these extensions are used in the oil wildcatter problem:

```
(:stochastic-method (init-oil~)
 (and (assign ?r (random 1.0d0))
      (oil-outcome ?r ?o))
 (:ordered (!init-oil~ ?o)))

(:op (!init-oil~ ?o)
 :add ((hidden (oil ?o)))
 :cost 0)

(:- (oil-outcome ?r ?o)
   ((< ?r 0.5) (= ?o dry))
   ((> ?r 0.5) (< ?r 0.8) (= ?o wet))
   ((> ?r 0.8) (= ?o soaking)))
```

`init-oil~` randomly samples from a categorical to find what the oil conditions then uses the uncontrolled operator `!init-oil~` to record the hidden literal. All uncontrolled tasks have the ~ character as suffix.

We represent the oil wildcatter problem as follows (full domain: https://pastebin.com/pUqLt2H6):

The top-level task is `oil-wildcatter`, which decomposes to `prepare-problem` and `make-decisions`.

`prepare-problem` decomposes to `init-oil~` followed by `init-test~`. `init-oil~`, is as described above.`init-test~` samples from a categorical (conditioned on the `oil`) and adds `(test-result` *s*`)`, for *s* = `ns`, `os`, or `cs`. These are only revealed to the agent if it tests.

`make-decisions` expands to `test-decision` followed by `drill-decision`. Each of these expands to a decision to either perform or not perform the corresponding action. `!test` incurs the testing cost and reveals the value of *o* in `(hidden (test-result` *o*`))` by adding `(test-outcome` *o*`)`. `drill` incurs the cost and accrues some income depending on *o* in `(hidden (oil` *o*`))`.

The way SHOPPINGSPREE handles partial observability is, in a way, the inverse of the one in Definition 1: state components default to being visible, and must be explicitly hidden, rather than needing to be explicitly observed.

4 Planning Algorithm

In this section, we describe how we have fused MCTS with SHOP3's HTN planning algorithm. As with a conventional planning problem, the input to SHOPPINGSPREE is a planning domain and problem (Section 3), but the notion of solution is quite different. The "solution" is a sequence of actions generated in an attempt to maximize the agent's utility (as defined in the domain and problem): see Alg. 1. At the moment, we have a very simple simulator, also based on the planning domain and state. In a real application one would connect the planning agent to effectors, and incorporate sensory inputs from the environment.

A simplified version of SHOP3's planning algorithm is given in Algorithms 2, and 3. At the top level, the "Find plans" procedure of Algorithm 2 is invoked with the initial state, the top-level task from the planning problem, and the empty set of bindings. For reasons of space, we do not discuass the basic SHOP3 algorithm in detail here. However, these pseudocode procedures were taken from our previous paper on SHOP3 (Goldman and Kuter 2019), interested readers can consult that paper for further details.

The first modification to the planning algorithm is to use the MCTS method for choosing options at nondeterministic choice points. These choice points occur at lines (5 in Alg. 2), (2 in Alg. 3), and (8 in Alg. 3). In each case, it is straightforward to adapt the MCTS bandit method to choose one of the alternatives from the finite set.

There is a complication, however: the scores for the MCTS algorithm must be tabulated at states that are *equiv-*

Algorithm 2 Simplified planning search algorithm.

```
 1: procedure FIND PLANS($S, T, B$)                    ▷ state, tasklist, bindings
 2:     if $T = \emptyset$ then
 3:         return ()        ▷ No tasks: return empty action sequence.
 4:     end if
 5:     choose $t \in T$ with no predecessors
 6:     if $t$ is primitive then
 7:         $o \leftarrow$ operator for $t$
 8:         if $o$ is applicable in $S$ then
 9:             $S' \leftarrow \mathrm{result}(o, S)$
10:             $T' \leftarrow T - t$
11:             $P \leftarrow$ FIND PLANS($S', T', B$)
12:             return $\mathrm{cons}(o, P)$
13:         else
14:             return FAIL
15:         end if
16:     else                                             ▷ $t$ is a complex task
17:         $< b, R' >= \mathrm{reduction}(t, S)$
18:         if $b$ is $FAIL$ then
19:             return FAIL
20:         else
21:             $B' = \mathrm{apply}(b, B)$
22:             if $B'$ is $FAIL$ then
23:                 ▷ Merge new bindings with incoming.
24:                 return FAIL
25:             end if        ▷ Replace $t$ with its expansion $R'$ in $T$
26:             $T' \leftarrow \mathrm{replace}(t, R', T)$
27:             return FIND PLANS($S, T', B'$)
28:         end if
29:     end if
30: end procedure
```

alent with respect to knowledge, rather than at complete states. See line 3 of Alg. 4. This is necessary because the agent can only choose actions based on the state it observes, rather than based on complete knowledge. The oil wildcatter can only choose to drill or not based on the results of the test (if they have done it), not based on the full state of the world, including the `oil` predicate. On the other hand, when doing a rollout, the planner must take into account the full state in order to project the outcome and its value. Put differently, the *agent* can *choose* only based on the visible part of the state, and the value estimates it collects through MCTS. But the *MCTS sampling algorithm* must *sample* (project) based on the full state, whose evolution it simulates.

In SHOPPINGSPREE, "equivalent with respect to knowledge" is implemented by projecting SHOP3's state onto the set of visible propositions. The set of visible propositions are those propositions that are not `hidden` (see Section 3). In addition to this, the state index (b in Alg. 4) includes the *visible history* of the state. The visible history of the state is the path of actions from the initial state leading to this state, omitting the uncontrolled actions.

The visible history is an essential part of the table indexing because the POMDP policy that we are computing is not memoryless; MDPs have optimal policies that are memoryless (Puterman 1994), but POMDPs do not unless the state is taken to include the agent's belief state, as well as the world state. The tables that MCTS computes are approximations of the value function, not the agent's belief state.

Algorithm 3 Task reduction procedure

```
 1: procedure REDUCTION($t, S$)                              ▷ task, State
 2:     choose $m$ a method for $\mathrm{name}(t)$
 3:     ▷ List of bindings from precondition query.
 4:     $b^* = \mathrm{query}(\mathrm{pre}(m), S)$
 5:     if $b^* = \emptyset$ then
 6:         return FAIL
 7:     else
 8:         choose $b \in b^*$               ▷ bindings from preconditions
 9:         $R \leftarrow$ task-net($m$)
10:         $R' \leftarrow \mathrm{apply}(b, R)$
11:         return b, R'
12:     end if
13: end procedure
```

Where Algs. 2 or 3 require a choice, the choice is made according to Alg. 4, which either initializes the choice table for the current choice and chooses an arbitrary alternative, or chooses the best decision, based on the current statistics, according to the MCTS rule chosen. For this reason, an MCTS rollout corresponds to the generation of a full plan, fitted to a particular outcome of the chance variables: in the case of the oil wildcatter problem, the `oil-state` and the outcome of a test (if the planner chooses to make one), given the `oil-state`. Because the plan is completed in a specific context in terms of the chance variables, it can be scored.

The score of each plan/rollout is backpropagated through the tree nodes of the search tree, and onto the corresponding statistics tables constructed as part of Algorithm 4. Per Equation 1, we increment the count of the action chosen at each table entry, and increment the cost entry with the cost of the full plan. If the search reaches a dead end – the current partial plan cannot be completed– we back-propagate $+\infty$.

The rollouts and backpropagation are the way that MCTS avoids the problems of myopic decision making we mentioned in the introduction. In the oil wildcatter problem, the planner can "see" that testing will provide information that will later be of use, rather than rejecting it because of the up-front cost. This kind of lookahead is more expensive and difficult under uncertainty than in classical planning.

The second modification is that this "planning algorithm" is not used as a conventional planner. Instead, it is essentially used as a way to populate the MCTS decision tables through planning and sampling. When it is time for plan execution, the system repeats the planning process, but this time, instead of using the tables to randomly generate a chosen action, the system takes the action dictated by the table: the one with the highest expected value (breaking ties randomly). Execution continues in this way until the system makes an observation, at which time the entire process repeats.

5 Conclusions

In this paper we have described an algorithm for HTN-based POMDP planning that combines the HTN planner, SHOP3, with Monte Carlo Tree Search. POMDP planning requires a combination of expressive power and sequential decision making in which observations and actions are interleaved, but where myopic planning and deterministic relaxations are

Algorithm 4 Choice method (Browne et al. 2012, Alg. 2, adapted)

```
1: $C \leftarrow$ MCTSTable()
2: function CHOOSE($S, T, A$)        ▷ state, choice type, alternatives
3:     $b \leftarrow$ belief state for$S$
4:     if $\langle b, T\rangle \neg \in C$ then        ▷ new choice table entry
5:         $C[\langle b, T\rangle] \leftarrow$ new table for$\langle b, T\rangle$
6:         return random member of A
7:     end if
8:     if $\exists c \in$ alternatives $\mid c \neg \in C[\langle b, T\rangle]$ then
9:         return $c$        ▷ try untried option
10:    else
11:        return best element of $C[\langle b, T\rangle]$
12:    end if
13: end function
```

insufficient to provide acceptable behaviors. Our work is still at an extremely early stage where we have developed the algorithm, but are still working to identify the best modeling techniques and to refine our method to achieve the best performance in the classes of problem that interest us.

Related work There is a great deal of work on applying MCTS to various sorts of planning. An obvious parallel to our work is the the application of MCTS to HTN by Wichlacz et al. (2020). The key difference here is that their work applies to classical HTN planning, not POMDPs.

SHOPPINGSPREE will not be competitive with other planning systems based on MCTS that aim to handle large POMDPs of standard structure (Silver and Veness 2010, *e.g.*,). Where we hope that SHOPPINGSPREE will shine is in problems with complex structures to the action space that will take advantage of HTN capabilities, where the sequential decision making is relatively simple, but bushy, and where we can profit from SHOP3's expressive power, allowing us to handle real-world complexities including object creation, numerical attributes, *etc.*.

Limitations A prominent limitation of this work is that it cannot handle situations where the HTN planner can get "off track" in the course of execution and not be able to complete a plan. Consider a case where the planner has chosen a method $M = T \rightarrow T_1, T_2, T_3$, to expand task T. Now imagine something goes wrong in the course of executing the subtasks of T_1. Since the planner has committed to M, it can no longer back up and consider alternatives, either alternatives to M, or alternatives to the current expansion of T_1. There are two ways to address this. The first is to ensure that the planning domain does not feature such dead ends. In our oil wildcatter example, as long as the discretization of the test results is complete, there are no dead ends.

An alternative to the no dead ends property is to support *plan repair*: the search space of the planner is expanded to consider repair operations when the execution of the current plan fails. We have developed an approach to plan repair for SHOP3 (Robert P. Goldman, Ugur Kuter, and Richard G. Freedman 2020) that we will incorporate in future work.

Acknowledgments This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory under Contract No. FA8750-17-C-0184. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not reflect the views of DARPA, the Dept. of Defense, or the U.S. Government.

References

Browne, C. B.; Powley, E.; Whitehouse, D.; Lucas, S. M.; Cowling, P. I.; Rohlfshagen, P.; Tavener, S.; Perez, D.; Samothrakis, S.; and Colton, S. 2012. A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games* 4(1): 1–43. ISSN 1943-068X, 1943-0698. doi:10.1109/TCIAIG.2012.2186810. URL http://ieeexplore.ieee.org/document/6145622/.

Goldman, R. P.; and Kuter, U. 2019. Hierarchical Task Network Planning in Common Lisp: The Case of SHOP3. In *Proceedings of the 12th European Lisp Symposium*. Genova, Italy.

Kaelbling, L. P.; Littman, M. L.; and Cassandra, A. R. 1998. Planning and Acting in Partially Observable Stochastic Domains. *Artificial Intelligence* 101(1-2): 99–134. ISSN 00043702. doi:10.1016/S0004-3702(98)00023-X. URL https://linkinghub.elsevier.com/retrieve/pii/S000437029800023X.

Kocsis, L.; and Szepesvári, C. 2006. Bandit Based Monte-Carlo Planning. In Hutchison, D.; Kanade, T.; Kittler, J.; Kleinberg, J. M.; Mattern, F.; Mitchell, J. C.; Naor, M.; Nierstrasz, O.; Pandu Rangan, C.; Steffen, B.; Sudan, M.; Terzopoulos, D.; Tygar, D.; Vardi, M. Y.; Weikum, G.; Fürnkranz, J.; Scheffer, T.; and Spiliopoulou, M., eds., *Machine Learning: ECML 2006*, volume 4212, 282–293. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN 978-3-540-45375-8 978-3-540-46056-5. doi:10.1007/11871842_29. URL http://link.springer.com/10.1007/11871842_29.

Puterman, M. L. 1994. *Markov Decision Processes—Discrete Stochastic Dynamic Programming*. New York, NY: John Wiley & Sons, Inc.

Raiffa, H. 1968. *Decision Analysis: Introductory Lectures on Choices under Uncertainty*. Behavioral Science: Quantitative Methods. New York: Random House.

Robert P. Goldman; Ugur Kuter; and Richard G. Freedman. 2020. Stable Plan Repair for State-Space HTN Planning. In *Hierarchical Planning Workshop*. Nancy, France. URL https://hierarchical-task.net/HPlan/HPlan2020-paper4.pdf.

Silver, D.; and Veness, J. 2010. Monte-Carlo Planning in Large POMDPs. In Lafferty, J.; Williams, C.; Shawe-Taylor, J.; Zemel, R.; and Culotta, A., eds., *Advances in Neural Information Processing Systems*, volume 23. Curran Associates, Inc. URL https://proceedings.neurips.cc/paper/2010/file/edfbe1afcf9246bb0d40eb4d8027d90f-Paper.pdf.

Wichlacz, J.; Höller, D.; Torralba, A.; and Hoffmann, J. 2020. Applying Monte-Carlo Tree Search in HTN Planning. In *Proceedings SoCS*.