Combined Task and Motion Planning for Mobile Manipulation

Jason Wolfe
Bhaskara Marthi
Stuart J. Russell

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2010-27
http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-27.html

March 8, 2010

Combined Task and Motion Planning for Mobile Manipulation

Jason Wolfe
Computer Science Division
University of California, Berkeley
Berkeley, CA 94720
jawolfe@cs.berkeley.edu

Bhaskara Marthi
Willow Garage, Inc.
Menlo Park, CA 94025
bhaskara@willowgarage.com

Stuart Russell
Computer Science Division
University of California, Berkeley
Berkeley, CA 94720
russell@cs.berkeley.edu

Abstract

We present a hierarchical planning system and its application to robotic manipulation. The novel features of the system are: 1) it finds high-quality kinematic solutions to task-level problems; 2) it takes advantage of subtask-specific irrelevance information, reusing optimal solutions to state-abstracted subproblems across the search space. We briefly describe how the system handles uncertainty during plan execution, and present results on discrete problems as well as pick-and-place tasks for a mobile robot. This is an extended version of a paper by the same name appearing in ICAPS '10.

1. Introduction

A useful household robot should be able to autonomously move around and manipulate objects in its environment. One such task is to tidy up a room by putting away objects in a set of target regions. In this paper, we assume that the initial state of the world is known (approximately) and consider the resulting decision-making problem. The robot must sequence the various pick-and-place operations, and decide on appropriate base positions, specific target locations for each object, and feasible grasps and corresponding paths through configuration space for its arms. We specifically consider *optimization* problems, where the goal is to find plans that minimize some measure of total cost (e.g., execution time).

Problems like these present a rich set of challenges. Even in simple environments, there are geometric constraints that are difficult to capture symbolically. For example, deciding where to move the base prior to placing a juice bottle on a table requires attention to obstacles on the floor and table, as well as the kinematics of the robot. These problems also have complex combinatorial structure, and are hard for A*-based planners (Helmert and Röger 2008). Finally, even specialized planners for low-level tasks such as arm movement take tens of milliseconds, strongly limiting the number of node expansions that can be performed by a forward search algorithm under real-time constraints.

Traditionally, such problems have been attacked top-down, separating high-level task planning (e.g., sequencing pick-and-place operations) from lower-level planning (e.g., finding feasible paths for the arm). Task planning is simplified by ignoring low-level details, but the resulting plans may be inefficient or even infeasible due to missed lower-level synergies and conflicts. For example, the task planner might sequence b before a, unaware that a particular way of doing a leaves the robot in an ideal configuration to follow with b, or worse, that every way of doing b renders a infeasible (e.g., by blocking the only feasible grasp for a).

Our first technical contribution is an alternative method, described in Section 3, for encoding robotic manipulation problems as vertically integrated hierarchical task networks (HTNs). At the bottom of the hierarchy, primitive actions (e.g., for arm or base movements) are modeled by calling out to external solvers such as rapidly-exploring random trees (RRTs). Continuous choices (e.g., grasp angles) are made finite via sampling. Sensing and robustness to failures are handled within the primitive actions. Given this encoding, the optimal plan allowed by the hierarchy can be found by exhaustive search. This plan will (with high probability) be a high-quality, guaranteed-feasible kinematic solution.

Our second contribution is the SAHTN algorithm, described in Section 4, which uses *subtask-specific irrelevance* to speed up this search. For example, the best way to move the arm to pick up object 23 depends only on the position of the base, arm, object 23, and nearby objects, and the results of such planning can be reused every time this subproblem occurs in the search space. Empirical results, presented in Section 5, show the effectiveness of the combined system.

2. Related Work

Several recent algorithms integrate information from the task level into a sampling-based motion planning framework. The configuration space can then be viewed as consisting of regions (one per instantiation of the discrete variables), connected by lower-dimensional submanifolds. aSyMov (Gravot, Cambon, and Alami 2003) decomposes the configuration space for manipulation into transit and transfer manifolds, taking advantage of stability constraints for free objects. Hauser and Latombe (2009) use a geometrically motivated heuristic function to focus sampling on those subtasks that are likely to be feasible. HYDICE (Plaku, Kavraki, and Vardi 2009), a hybrid system verification tool, also uses a heuristic estimate of the likely feasibility of each subproblem to focus sampling. Our algorithm differs from these in its focus on optimization, and its use of relevance information (aSyMov does this to some degree by reusing roadmaps). An advantage of the above methods is that they interleave sampling between motion planning subproblems.

Hierarchical planning also has a long history in the discrete planning literature, and hierarchical task network

`BaseAction`(p, θ)	Move the base to position p with angle θ. Executed using a separate system that does its own perception, A* planning, and obstacle-avoidance. Modeled by calling out to this system's planner with a static map, to determine feasibility and estimated cost.
`ArmJointAction`$(\boldsymbol{\theta})$	Move the arm to joint vector $\boldsymbol{\theta}$. Executed using a separate system that does its own perception, sampling-based planning, and obstacle-avoidance. Modeled by calling out to this system's planner with a rendered collision map, to determine feasibility and estimated cost.
`ArmGraspAction`(p, θ)	Grasp the object near point p from angle θ. Executed by using the laser range-finder to find the exact position of the object on a surface nearest to p, and moving the arm into a grasp position (computed by inverse kinematics) using the same system as `ArmJointAction`. Modeled by assuming that the object will be exactly at point p.
`CloseGripperAction`(o)	Close the gripper, to grasp object o. Executed by closing the gripper (using force-feedback to ensure an object was grasped), and updating the robot model to include the grasped object. Modeled by assuming constant cost, and that o becomes attached to the gripper.
`OpenGripperAction`$()$	Open the gripper. In execution, updates the robot model. Modeled by assuming constant cost, and that the held object (if any) ends up on the surface below the gripper.
`TorsoAction`(h)	Extend the torso to height h. Used when picking or placing, to work around problems the external arm planner had with contacts. Executed and modeled as expected.

Figure 1: Primitive actions for the pick-and-place domain

(HTN) planners such as SHOP2 (Nau et al. 2003) have been widely used in practice. The rich geometry of robotic configuration spaces seems difficult to express using standard discrete representations alone, though, and so these methods have tended to stay above the motion planning level. Moreover, little work has been done on optimization for HTNs.

Abstracting a problem using a subset of state variables has also been considered by many researchers. Most work we are familiar with in the planning literature uses abstraction to create a simpler approximation to the problem, whose solution is then expanded (Sacerdoti 1973) or used as a heuristic (Culberson and Schaeffer 1996; Holte, Grajkowski, and Tanner 2005; Felzenszwalb and McAllester 2007). For the related problem of learning or deriving an optimal policy, prior work in the reinforcement learning literature (Dietterich 2000; Diuk, Strehl, and Littman 2006) uses an exact abstraction to reduce the size of the value function or policy representation. Finally, cognitive architectures such as SOAR (Laird, Newell, and Rosenbloom 1987) use explanation-based learning to avoid deliberating about the same subproblem twice; however, these systems do not do optimal planning, and are not explicitly hierarchical.

3. Pick-and-Place Domain and Hierarchy

We consider a pick-and-place domain for a mobile robot with arm(s), where the environment consists of objects (juice bottles) on various surfaces. The robot's task is to move all objects to their goal regions, as quickly as possible.

States for this problem consist of a robot joint configuration, along with, for each object, a description of its geometry, current position, and relation to other objects (e.g., *bottle1* on *table1*).

The primitive actions in this domain move a specific part of the robot (base, torso, arm, or gripper) to a target joint configuration. One additional action moves the arm into a grasp configuration given the approximate location of a target object and a grasp angle (this action is represented as primitive because it involves perceptual feedback from the laser scanner, and our planner assumes full observability).

For each primitive action, we require a *transition model* that takes in a state and returns the successor state and action cost (or failure). The complex geometric constraints in the environment make it difficult to use declarative representations such as PDDL directly. Instead, the transition models are procedural, and call out to external planners. For example, `ArmJointAction` is implemented by a sampling-based motion planner (Rusu et al. 2009), using a collision map rendered from the current state. Action costs are set based on the estimated time required. For example, the cost of `ArmJointAction` is the length of the returned path, multiplied by a weighting constant. See Figure 1 for detailed descriptions of the primitive actions.

In addition to an initial state and transition model, our planning algorithm also takes a *action hierarchy* as input. This hierarchy specifies 1) a finite set of high-level actions (HLAs), including a distinguished top-level action `Act`; and 2) for each HLA and state, a set of immediate refinements into finite sequences of (possibly high-level) actions.

Such a hierarchy both structures and potentially restricts the space of solutions. In particular, rather than searching directly over primitive action sequences, an agent can begin with a plan consisting of just `Act`, and repeatedly replace the first non-primitive action in this plan with an immediate refinement until a primitive solution is found. We assume WLOG that the hierarchy does not generate primitive non-solutions. This can always be achieved, e.g., by ending each plan with a `Goal` HLA that has zero refinements from non-goal states and an empty refinement from goal states.

Concretely, we use the following hierarchy. `Act` has recursive refinements `MoveToGoal`(o), `Act` ranging over all objects o that are not yet in their goal positions, or just the empty refinement if this set is empty. `MoveToGoal`(o) refines in turn to `GoPick`(o), `GoPlace`(o, p), ranging over positions p in the goal region of o. Both of these HLAs refine to `ArmTuck`, `BaseRgn`(r) if needed (where r is a region around the target), followed by `Pick`(o) or `Place`(o, p). `Pick` and `Place` further refine to choose a grasp/drop angle, and then generate the appropriate sequence of arm, torso, and gripper primitives to effect the appropriate pick or place operation. See Figure 2 for a more detailed description of the hierarchy.

As with the primitive transition models, we allow the set of refinements of an HLA to be generated by arbitrary code. This allows, e.g., `GoPick` to compute a candidate base region for the grasp, and `Place` to use inverse kinematics to generate valid arm joint configurations for the drop. One complication that arises in hybrid domains is that the set of refinements of a high level action may be (uncountably) in-

HLA	Refinements
`Act`	[`MoveToGoal`(o), `Act`] | o not at goal [] | all objects at goals
`MoveToGoal`(o)	[`GoPick`(o), `GoPlace`(o, p)] | p in goal region of o
`GoPick`(o)	[`Pick`(o)] | o in range [`ArmTuck`,`BaseRgn`(r),`Pick`(o)] | r is candidate base region
`Pick`(o)	[`ArmGraspAction`($pos(o), \theta$), `CloseGripperAction`(o), `TorsoAction`(up)] | $\theta \in [-1, 1]$ rad
`GoPlace`(o, p)	[`Place`(o, p)] | p in range [`ArmTuck`,`BaseRgn`(r),`Place`(o, p)] | r is candidate base region
`Place`(o, p)	[`ArmJointAction`($\boldsymbol{\theta}_1$), `TorsoAction`($down$), `OpenGripperAction`, `ArmJointAction`($\boldsymbol{\theta}_2$)] | $\boldsymbol{\theta}$ are candidate joint configs
`ArmTuck`	[`ArmJointAction`($tucked$)]
`BaseRgn`(r)	[`BaseAction`(p, θ)] | $p, \theta \in r$

Figure 2: High-level actions (HLAs) for the pick-and-place domain

finite. For example, the `BaseRgn`(r) action has one refinement `BaseAction`(p) for each point p in region r. We make the search space finite by having each such HLA generate a random sample of pre-specified size from its infinite set of refinements. The same random bits are used across invocations of an HLA, which produces a "graphy" reachable state space that provides more opportunities for caching. For instance, each time we consider `Place`ing a particular object we will generate the same candidate target positions.

Note that hierarchical constraints may rule out all optimal solutions; for instance, our hierarchy does not allow putting an object down in an intermediate location. A plan is called *hierarchically optimal* if has minimal cost among all plans generated by the hierarchy. Moreover, let N be the minimum number of samples used when discretizing the refinements of an HLA (or calling a sampling-based primitive planner). A planning algorithm is *hierarchically resolution-optimal* if, as $N \to \infty$, the cost of the returned plan converges almost surely to the hierarchically optimal cost. Finally, define the *reachable state space* under a hierarchy as the set of all states visited by any primitive refinement of `Act`. In this paper, we assume that the reachable state space is always finite, and restrict our attention to hierarchically (resolution-) optimal planning algorithms.

4. SAHTN Algorithm

This section presents the State-Abstracted Hierarchical Task Network (SAHTN) algorithm. It takes in a domain description, action hierarchy, and a RELEVANT-VARS function specifying which state variables matter for doing an action from some state. The output is a hierarchically optimal solution.

As a stepping stone towards describing SAHTN, we first present a simple exhaustive, hierarchically optimal HTN algorithm (see Figure 1). The top-level SOLVE function takes an initial state, and returns a hierarchically optimal solution (or *failure*). RESULTS-A and RESULTS-P take an initial state and action or plan, and return a map from each state reachable by (some refinement of) the given action or plan to a tuple containing an optimal primitive refinement that

Algorithm 1 : Optimal HTN algorithm for acyclic problems

```
/* s is a state, a is an action, p is a plan (action sequence), and
 * c is its corresponding cost. The MERGE function merges maps,
 * retaining the minimum-cost entry for each reachable state.  */

function SOLVE($s$)
    return the plan associated with the min-cost state
          in RESULTS-A($s$, Act), or failure if empty

function RESULTS-A($s, a$)
    if $a$ is high-level then
        return MERGE({RESULTS-P($s, ref$) |
                        $ref \in$ REFINEMENTS($a, s$)})
    else if ¬APPLICABLE($s, a$) then return { }
    else return {SUCCESSOR($s, a$) : [COST($s, a$), [$a$]]}

function RESULTS-P($s, p$)
    output ← {$s$ : [0, [ ]]}
    for each $a'$ in $p$ do
        output ← MERGE({BOOKKEEP-A($s', a', c', p'$) |
                        $s'$ : [$c', p'$] ∈ output})
    return output

function BOOKKEEP-A($s, a, c, p$)
    output ← {}
    for each $s'$ : [$c', p'$] ∈ RESULTS-A($s, a$) do
        output[$s'$] ← [$c + c', p + p'$]
    return output
```

Algorithm 2 : Changes to Algorithm 1 for SAHTN algorithm

```
function BOOKKEEP-A($s, a, c, p$)
    relevant ← RELEVANT-VARS($s, a$)
    if cache has no entry for [$s_{relevant}, a$] then
        cache[$s_{relevant}, a$] ← RESULTS-A($s, a$)
    output ← {}
    for each $s'$ : [$c', p'$] ∈ cache[$s_{relevant}, a$] do
        output[$s_{\overline{relevant}} + s'_{relevant}$] ← [$c + c', p + p'$]
    return output
```

reaches this state and its cost (cf. the exact valuations of Marthi, Russell, and Wolfe (2008)). In particular, RESULTS-A returns the single outcome state for a primitive action, or the best cost and plan to each reachable state (over all refinements) for a high-level action. RESULTS-P computes the result for an action sequence by progressing through each action in turn. Finally, BOOKKEEP-A simply calls RESULTS-A, with some extra bookkeeping to sum optimal costs and concatenate optimal plans along action sequences.

SAHTN (see Algorithm 2) makes this into a dynamic programming algorithm by adding state-abstracted caching to the BOOKKEEP-A function. We assume a global *cache* of the results of each call to RESULTS-A(s, a), keyed by a and the values of state variables in s that are *relevant* to a from s. Then, in a later call to RESULTS-A(s', a) where s' has the same relevant variable values as s, we simply look up the cached result and combine it with the irrelevant variables of s' to produce the output mapping. For this output to be correct, it is crucial that RELEVANT-VARS(s, a) truly captures all aspects of state s that are relevant to doing action a:

Definition 1. A state variable v is *relevant* to action a from state s, iff (1) the set of primitive refinements of a from s depends on v, or (2) any primitive refinement of a from s conditions on, sets, or has cost dependent on the value of v.

For example, the only variables relevant to

Algorithm 3 : Changes to Algorithm 2 for cyclic hierarchies

```
function RESULTS-A($s, a$)
    if $a$ is high-level then
        if CYCLE-ID($s, a$) $\neq$ $null$ then return DIJKSTRA($s, a$)
        else return MERGE($\{$RESULTS-P($s, ref$) |
                                $ref \in$ REFINEMENTS($a, s$)$\}$)
    else if $\neg$APPLICABLE($s, a$) then return $\{\ \}$
    else return $\{$SUCCESSOR($s, a$) : [COST($s, a$), $[a]$]$\}$

function DIJKSTRA($s, a$)
    $q \leftarrow$ a priority queue on $(state, plan)$ pairs, ordered by
        cost to $state$, which ignores re-adds with $\geq$ cost,
        initially containing only $(s, [a])$.
    $output \leftarrow \{\}$
    while $q$ is not empty do
        $(s', p') \leftarrow$REMOVE-MIN($q$)
        if $p'$ is empty then
            $output[s'] \leftarrow [c, p]$, where $p$ is best plan to $s'$
                                from $s$, and $c$ is its cost
        else
            $p'_{first}, p'_{rest} \leftarrow$ first, remaining actions in $p'$
            if CYCLE-ID($s', p'_{first}$) = CYCLE-ID($s, a$) then
                for each $ref$ in REFINEMENTS($p'_{first}, s'$) do
                    add $(s', ref + p'_{rest})$ to $q$
            else
                for each $s''$ : $[c, p] \in$ RESULTS-A($s', p'_{first}$) do
                    add $(s'', p'_{rest})$ to $q$
    return $output$
```

BaseAction(p) are the the current base position, and the positions of objects that are potential obstacles.[1]

Before stating the correctness of SAHTN, there is one more issue that must be made concrete. As described, the algorithm will loop forever in *cyclic* hierarchies; we first define this condition, and later discuss how it can be relaxed.

Definition 2. Call pair (s', a') a *subproblem* of (s, a) iff there exists a refinement $\mathbf{r}$ of a from s and integer i s.t. $s' = \text{SUCCESSOR}(s, r_{1:i-1})$ and $a' = r_i$. A hierarchy is *cyclic* from initial state s iff there exists any subproblem of $(s, \texttt{Act})$ that has itself as a subproblem.

Equivalently, a hierarchy is cyclic if SAHTN ever recursively calls RESULTS-A(s, a) when RESULTS-A(s, a) is already on the call stack. Note that an acyclic hierarchy may include recursive HLAs. For instance, while the Act HLA for our pick-and-place domain is recursive, the hierarchy is *not* cyclic because in every recursive application of Act, one more object will have been placed in its goal position.

Theorem 1. Given an acyclic hierarchy, correct RELEVANT-VARS function, and finite reachable state space, SAHTN will always return a hierarchically optimal solution.[2]

The acyclic requirement can rule out some natural hierarchies. For instance, consider a Nav(x, y) HLA in a grid-world domain, which refines to the empty sequence iff at (x, y), and otherwise to a primitive move action followed by a recursive Nav(x, y). In this case, the refinement Left, Right, Nav(x, y) leads to a cycle in the above sense. Fortunately, a simple change to SAHTN can extend Theorem 1 to such cases. The basic idea is to add a case to RESULTS-A: if (s, a) potentially leads to a cycle, instead of recursively decomposing run Dijkstra's algorithm locally over the space of potential cycles, switching back to recursive computation for (s', a') pairs that can not cycle back to (s, a).

Algorithm 3 shows changes to SAHTN that implement this extension. We assume that the function CYCLE-ID(s, a) returns the same (arbitrary) identifier for every (s, a) pair involved in a given cycle, or $null$ if (s, a) cannot cycle. In the above example, it might return "Nav" if a is a Nav and $null$ otherwise. Now, Algorithm 3 modifies RESULTS-A(s, a) so that whenever it encounters an (s, a) pair with non-$null$ cycle ID, it calls the new function DIJKSTRA(s, a) rather than decomposing as usual. This new function does a Dijkstra search over $(state, plan)$ pairs, where $state$ is the result of doing a (primitive) prefix of a refinement of a from s, and $plan$ is the remaining (possibly high-level) actions in this refinement. To compute the successors of a pair, we either (1) progress $state$ through the first action in $plan$ using RESULTS-A, if this computation is not part of the current cycle, or otherwise (2) refine the first action in $plan$. (Note that primitive actions cannot lead to cycles, and will always fall in the first category). Finally, when $plan$ is empty we have found an optimal primitive refinement of a to a new state, and we add this to the $output$. (Bookkeeping code that keeps track of the best plan to each state is not shown.)

Because each $(state, plan)$ pair can be dequeued at most once, DIJKSTRA(s, a) is guaranteed to terminate (with the correct results) as long as the set of potential $plan$s is finite. This is always true as long as the hierarchy obeys certain natural conditions (e.g., in recursive refinements, the recursive HLA is always the last action). The cost of using DIJKSTRA(s, a) to avoid cycles is that some intermediate computations within the cycle are not cached. For instance, when computing the results of Nav$(2, 2)$ from $(1, 1)$, these results are cached but the results of Nav$(2, 2)$ from $(1, 2)$ (an implicit intermediate computation) are not.

5. Results

We now present comparisons of SAHTN with three other search algorithms. Uniform-cost search (UCS) searches forward from the initial state, without using a hierarchy. Hierarchical UCS (H-UCS) searches in a space where each node consists of the state arising from the primitive prefix of a plan, together with the remaining actions in the plan. N-SAHTN is the SAHTN algorithm with no state abstraction.

We first present results on a version of the *taxi domain*, a benchmark commonly used in the reinforcement learning literature (Dietterich 2000). Taxi problems consist of a taxi in a grid world, and passengers with randomly chosen source and destination squares. The taxi must pick up and drop off the passengers, one at a time. On such problems (see Figure 3), SAHTN (with the Dijkstra modification mentioned above) clearly scales much better than other algorithms as the number of passengers increases, due to decoupling of navigation decisions from high-level sequencing of passengers. All algorithms are primitive-optimal in this domain.

[1]With appropriate changes to the algorithm, more general notions of relevance can be considered as well. For instance, in a multi-robot problem one could share results between MoveToGoal(*robot1*, o) and MoveToGoal(*robot2*, o) for interchangeable robots with identical starting configurations.

[2]See Appendix A for proof.

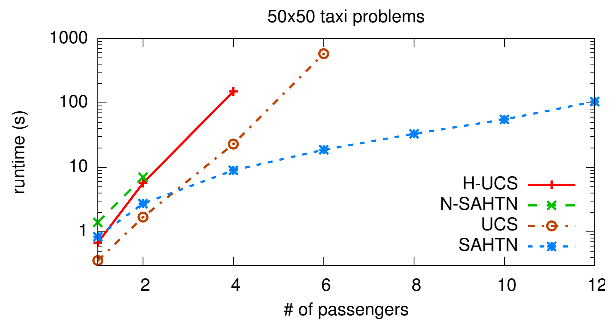

Figure 3: Runtimes for algorithms on 50x50 taxi problems, as a function of the number of passengers. Larger problems were not solvable by any algorithm but SAHTN within 512MB of memory.

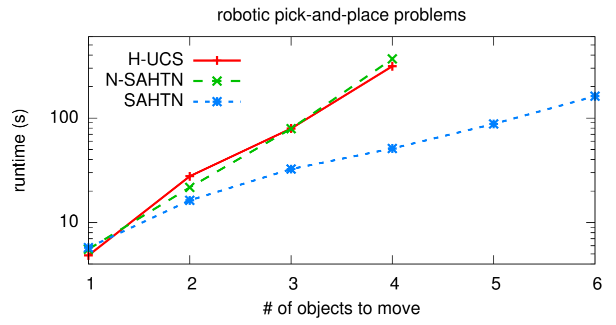

Figure 4: Runtimes (averaged over three runs) for three algorithms on pick-and-place tasks, as a function of the number of objects to be moved. Larger problems were not solvable by any algorithm except SAHTN within 10 min. Two runs where unlucky sampling led to no solutions being found were discarded and rerun.

We next evaluated our algorithms on the full pick-and-place domain, using a prototype PR2 robot constructed by Willow Garage, Inc (Wyrobek et al. 2008). The robot has a wheeled base and two 7-dof arms (including 1-dof gripper). Figure 4 shows results on single-arm pick-and-place tasks with two tables and randomly placed objects and goal regions, for increasing numbers of objects. There is no obvious way to apply uniform-cost search to these problems, so we compare the three other algorithms (all of which take advantage of our hierarchical problem formulation), using the same sampling settings. We again see significant improvement in runtime from state abstraction in SAHTN, increasing as the number of objects grows. SAHTN's performance is initially dominated by the polynomially many primitive action applications it must make, although eventually the exponential cost of the top-level traveling salesman problem will take over.

We also implemented, for each primitive action, a function to execute it on the PR2. The execution primitives were responsible for implementing perceptual feedback and returning a success flag, which was used to implement a simple executive that retried failed actions. A video of the PR2 executing a 4-object plan is available online.[3]

[3]Supplemental materials are available online at `http://www.ros.org/wiki/Papers/ICAPS2010_Wolfe`

6. Conclusion

We view this work as a proof-of-concept that task-level planning can be successfully extended all the way to the kinematic level, to generate robust and high-quality plans. To help reduce computation times for larger tasks, future work will examine extensions of SAHTN that can use available heuristic information, and perhaps approximate models for the HLAs and primitives (Marthi, Russell, and Wolfe 2008), to guide search and reduce calls to external solvers. Another important improvement will be incremental, adaptive search algorithms that better manage the tradeoff between computational cost and plan quality.

References

Culberson, J. C., and Schaeffer, J. 1996. Searching with Pattern Databases. In *Advances in Artificial Intelligence (Lecture Notes in Artificial Intelligence 1081*, 402–416.

Dietterich, T. G. 2000. Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition. *JAIR* 13:227–303.

Diuk, C.; Strehl, A. L.; and Littman, M. L. 2006. A Hierarchical Approach to Efficient Reinforcement Learning in Deterministic Domains. In *AAMAS*, 313–319.

Felzenszwalb, P. F., and McAllester, D. 2007. The Generalized A* Architecture. *J. Artif. Int. Res.* 29(1):153–190.

Gravot, F.; Cambon, S.; and Alami, R. 2003. aSyMov: A Planner That Deals with Intricate Symbolic and Geometric Problems. In *ISRR*, 100–110.

Hauser, K., and Latombe, J. C. 2009. Integrating Task and PRM Motion Planning. In *Workshop on Bridging the Gap between Task and Motion Planning, ICAPS 2009*.

Helmert, M., and Röger, G. 2008. How Good is Almost Perfect? In *AAAI*, 944–949.

Holte, R. C.; Grajkowski, J.; and Tanner, B. 2005. Hierarchical Heuristic Search Revisited. In *SARA*, 121–133.

Laird, J. E.; Newell, A.; and Rosenbloom, P. S. 1987. SOAR: an Architecture for General Intelligence. *Artif. Intell.* 33(1):1–64.

Marthi, B.; Russell, S. J.; and Wolfe, J. 2008. Angelic Hierarchical Planning: Optimal and Online Algorithms. In *ICAPS*.

Nau, D.; Au, T. C.; Ilghami, O.; Kuter, U.; Murdock, W. J.; Wu, D.; and Yaman, F. 2003. SHOP2: An HTN planning system. *JAIR* 20:379–404.

Plaku, E.; Kavraki, L. E.; and Vardi, M. Y. 2009. Hybrid systems: from verification to falsification by combining motion planning and discrete search. *Formal Methods in System Design* 34(2):157–182.

Rusu, R. B.; Şucan, I. A.; Gerkey, B.; Chitta, S.; Beetz, M.; and Kavraki, L. E. 2009. Real-time perception guided motion planning for a personal robot. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 4245–4252.

Sacerdoti, E. D. 1973. Planning in a Hierarchy of Abstraction Spaces. In *IJCAI*, 412–422.

Wyrobek, K. A.; Berger, E. H.; der Loos, H. F. M. V.; and Salisbury, J. K. 2008. Towards a personal robotics development platform: Rationale and design of an intrinsically safe personal robot. In *ICRA*, 2165–2170.

Appendix A. Proof of Theorem 1

We will first prove the correctness of SAHTN without state abstraction, i.e., when RELEVANT-VARS(s,a) always returns all variables in s, and then show how adding correct state abstraction does not affect the proof.

Lemma 1. Given an acyclic hierarchy, RELEVANT-VARS function that always returns the set of all state variables, and finite reachable state space, SAHTN will always return a hierarchically optimal solution.

Proof. First, note that RESULTS-A(s, a) can be called at most once for each (s, a) pair. After the first call to RESULTS-A(s, a) returns, BOOKKEEP-A will cache the results and never call RESULTS-A(s, a) again. Thus, the only way RESULTS-A(s, a) could be called twice would be if the second call was made recursively inside the first call; however, this situation is precluded by the requirement that the hierarchy is acyclic.

Now, since RESULTS-A(s, a) can be called at most once for each (s, a) pair, and the set of reachable states s and possible HLAs a are both finite, RESULTS-A can only be called a finite number of times, and SAHTN must terminate.

It remains to show that when SAHTN does terminate, it always returns a hierarchically optimal solution (when one exists). We do this by proving by induction that RESULTS-A(s, a) is correct (e.g., always returns a map whose keys are those states reachable by some primitive refinement of a from s, and whose values are the associated optimal costs and corresponding primitive refinements.) In particular, we use structural induction over recursive calls to RESULTS-A(s, a) (or cached results thereof), which must form a finite partial order as argued above.

The base case is when a is primitive. In this case a is its own sole primitive refinement, and RESULTS-A(s, a) correctly outputs the empty map if a is not applicable to s, and otherwise a map containing the single successor of s on a, with cost equal to the cost of this transition and optimal primitive refinement $[a]$.

For the inductive case where a is high-level, we assume that all immediate recursive calls (including cached calls) made to RESULTS-A(s', a') within a call to RESULTS-A(s, a) terminate with the correct result. First, note that this entails that RESULTS-P(s, p) returns the correct result for each immediate refinement of a from s. This follows from the observation that each optimal primitive refinement of a sequence of actions is a concatenation of optimal primitive refinements of its component actions. Now, since the set of primitive refinements of a from s is precisely the union of the sets of primitive refinements of the immediate refinements of a from s, the call to RESULTS-A(s, a) is correct.

Thus, by induction, the top-level call to RESULTS-A$(s, \texttt{Act})$ is correct, and so SOLVE(s) will always return an optimal solution. □

Theorem 1 (reprise). Given an acyclic hierarchy, correct RELEVANT-VARS function, and finite reachable state space, SAHTN will always return a hierarchically optimal solution.

Proof. Add to the induction of Lemma 1, showing that the output of BOOKKEEP-A(s, a, c, p) is unchanged by adding state abstraction, i.e., when RELEVANT-VARS(s, a) is an arbitrary function that respects Definition 1. Note that the recursive structure for the induction may change, but will remain a valid partial order.

In a particular call to BOOKKEEP-A(s, a, c, p), there are two cases. First, if *cache* had no entry for $[s_{relevant}, a]$, everything is the same as with no state abstraction as long as RESULTS-A(s, a) does not modify any variables outside of RELEVANT-VARS(s, a), which it cannot by Definition 1. Second, if *cache* does have an entry, it contains RESULTS-A(s', a) from a previous call to BOOKKEEP-A(s', a, c', p') where s and s' share the same relevant variables and values. We require that these results are identical to RESULTS-A(s, a), except for values outside of RELEVANT-VARS(s, a). To show this, we simply note that by Definition 1 the sets of primitive refinements of a from s and s' must be identical, each such refinement cannot test or set the irrelevant variables of s or s', and thus each such refinement must have the same effects on the (identical) relevant variables of s and s'. □