HTN Planning with Semantic Attachments

Maurício Cecílio Magnaguagno, Felipe Meneguzzi

School of Computer Science (FACIN)
Pontifical Catholic University of Rio Grande do Sul (PUCRS)
Porto Alegre - RS, Brazil
mauricio.magnaguagno@acad.pucrs.br
felipe.meneguzzi@pucrs.br

Abstract

Hierarchical Task Networks (HTN) generate plans using a decomposition process guided by extra domain knowledge to guide search towards a planning task. While many HTN planners can make calls to external processes (e.g. to a simulator interface) during the decomposition process, this is a computationally expensive process, so planner implementations often use such calls in an ad-hoc way using very specialized domain knowledge to limit the number of calls. Conversely, the few classical planners that are capable of using external calls (often called *semantic attachments*) during planning do so in much more limited ways by generating a fixed number of ground operators at problem grounding time. In this paper we develop the notion of *semantic attachments* for HTN planning using semi co-routines, allowing such procedurally defined predicates to link the planning process to custom unifications outside of the planner. The resulting planner can then use such co-routines as part of its backtracking mechanism to search through parallel dimensions of the state-space (e.g. through numeric variables). We show empirically that our planner outperforms the state-of-the-art numeric planners in a number of domains using minimal extra domain knowledge.

Introduction

Planning in domains that require numerical variables, for example, to drive robots in the physical world, must represent and search through a space defined by real-valued functions with a potentially infinite domain, range, or both. This type of numeric planning problem poses challenges in two ways. First, the description formalisms (Fox and Long 2003) might not make it easy to express the numeric functions and its variables, resulting in a description process that is time consuming and error-prone for real-world domains (Strobel and Kirsch 2014). Second, the planners that try to solve such numeric problems must find efficient strategies to find solutions through this type of state-space. Previous work on formalisms for domains with numeric values developed the Semantic Attachment (SA) construct (Dornhege et al. 2009) in classical planning. Semantic attachments were coined by (Weyhrauch 1981) to describe the attachment of an interpretation to a predicate symbol using an external procedure. Such construct allows the planner to reason about fluents where numeric values come from externally defined functions. In this paper, we extend the basic notion of semantic attachment for HTN planning by defining the semantics of the functions used as semantic attachments in a way that allows the HTN search and backtracking mechanism to be substantially more efficient. Our current approach focused on depth-first search HTN implementation without heuristic guidance, with free variables expected to be fully-ground before task decomposition continues.

Most planners are limited to purely symbolic operations, lacking structures to optimize usage of continuous resources involving numeric values (Gerevini, Saetti, and Serina 2008). Floating point numeric values, unlike discrete logical symbols, have an infinite domain and are harder to compare as one must consider rounding errors. One could overcome such errors with delta comparisons, but this solution becomes cumbersome as objects are represented by several numeric values which must be handled and compared as one, such as points or polygons. Planning descriptions usually simplify such complex objects to symbolic values (e.g. *p25* or *poly2*) that are easier to compare. Detailed numeric values are ignored during planning or left to be decided later, which may force replanning (Şucan and Kavraki 2011). Instead of simplifying the description or doing multiple comparisons in the description itself, our goal is to exploit external formalisms orthogonal to the symbolic description. To achieve that we build a mapping from symbols to objects generated as we query semantic attachments. Semantic attachments have already been used in classical planning (Dornhege et al. 2009) to unify values just like predicates, and their main advantage is that new users do not need to discern between them and common predicates. Thus, we extend classical HTN planning algorithms and their formalism to support semantic attachment queries. While external function calls map to functions defined outside the HTN description, we implement SAs as semi co-routines (Dahl, Dijkstra, and Hoare 1972), subroutines that suspend and resume their state, to iterate across zero or more values provided one at a time by an external implementation, mitigating the potentially infinite range of the external function.

Our contributions are threefold. First, we introduce SAs for HTN planning as a mechanism to describe and evaluate external predicates at execution time. Second, we introduce a symbol-object table to improve the readability of symbolic

descriptions and the plans generated, while making it easier to handle external objects and structures. Finally, we empirically compare the resulting HTN planner with a modern classical planner (Ilghami and Nau 2003) in a number of mixed symbolic/numeric domains showing substantial gains in speed with minimal domain knowledge.

Background

Classical Planning

Classical planning algorithms must find plans that transform properties of the world from an initial configuration to a goal configuration. Each property is a logical predicate, a tuple with a name and terms that relate to objects of the world. A world configuration is a set of such tuples, which is called a state. To modify a state one must apply an operator, which must fulfill certain predicates at the current state, as preconditions, to add and remove predicates, the effects. Each operator applied creates a new intermediate state. The set of predicates and operators represent the domain, while each group of objects, initial and goal states represent a problem within this domain. In order to achieve the goal state the operators are used as rules to determine in which order they can be applied based on their preconditions and effects. To generalize the operators and simplify description one can use free variables to be replaced by objects available, a process called grounding. Once a state that satisfies the goal is reached, the sequence of ground operators is the plan (Nebel 2000). A plan is optimal, iff it achieves the best possible quality in some criteria, such as number of operators, time or effort to execute; or satisficing if it reaches the goal without optimizing any metrics. PDDL (McDermott et al. 1998) is the standard description language to describe domains and problems, with features added through requirements that must be supported by the planner. Among such features are numeric-valued fluents to express numeric assignments and updates to the domain, as well as events and processes to express effects that occur in parallel with the operators in a single instant or during a time interval.

Hierarchical Task Networks

Hierarchical planning shifts the focus from goal states to tasks to exploit human knowledge about problem decomposition using a hierarchy of domain knowledge recipes as part of the domain description (Nau et al. 1999). This hierarchy is composed of primitive tasks that map to operators and non-primitive tasks, which are further refined into sub-tasks using *methods*. The decomposition process is repeated until only primitive-tasks mapping to operators remain, which results in the plan itself. The goal is implicitly achieved by the plan obtained from the decomposition process. If no decomposition is possible, the task fails and a new expansion is considered one level up in the hierarchy, until there are no more possible expansions for the root task, only then a task decomposition is considered unachievable. Unlike classical planning, hierarchical planning only considers tasks obtained from the decomposition process to solve the problem, which both limits the ability to solve problems and improves execution time by evaluating a smaller number of operators. The HTN planning description is more complex than equivalent classical planning descriptions, since it includes domain knowledge with potentially recursive tasks, being able to solve more problems than classical planning.

Symbolic-Geometric Planning

Classical planners with heuristic functions can solve problems mixing symbolic and numeric values efficiently using a process of discretization. A discretization process converts continuous values into sets of discrete symbols at often predefined granularity levels that vary between different domains. However, if the discretization process is not possible, one must use a planner that also supports numeric features, which requires another heuristic function, description language and usually more computing power due to the number of states generated by numeric features. Numeric features are especially important in domains where one cannot discretize the representation, they usually appear in geometric or physics subproblems of a domain and cannot be avoided during planning. Unlike symbolic approaches where literals are compared for equality during precondition evaluation, numeric value comparison is non-trivial. To avoid doing such comparison for every numeric value the user is left responsible for explicitly defining when one must consider rounding errors, which impacts description time and complexity. For complex object instances (in the object-oriented programming sense), such as polygons that are made of point instances, comparison details in the description are error-prone. Details such as the order of polygon points and floating point errors in their coordinates are usually irrelevant for the planner and the domain designer and should not be part of the domain description as they are part of a low-level specification.

Such low-level specifications can be implemented by external function calls to improve what can be expressed and computed by a HTN planner. Such functions come with disadvantages, as they are not expected to keep an external state, returning a single value solely based on the provided parameters. While HTN planners can abstract away the numeric details via external function calls, there are limitations to this approach if a particular function is used in a decomposition tree where it is expected to backtrack and try new values from the function call (i.e. if the function is meant to be used to generate multiple terms as part of the search strategy). An external function must return a list of values to account for all possible decompositions so the planner tries one at a time until one succeeds. Generating a complete list is too costly when compared to computing a single value, as the first value could be enough to find a feasible plan. A semantic attachment, on the other hand, acts as an external predicate that unifies with one possible set of values at a time, rather than storing a complete list of possible sets of values to be stored in the state structure. This implementation saves time and memory during planning, as only backtracking causes the external co-routine to resume generating new unifications until a plan (or a certain amount of plans) is found. Each SA acts as a black box that simulates part of the environment encoding the results in state variables that are often orthogonal to other predicates (Francès et al. 2017).
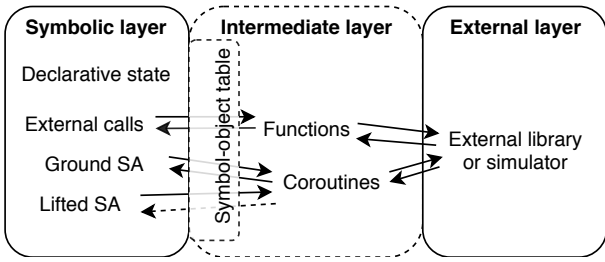
Figure 1: Symbolic and external layers share information through an intermediate layer that maps representations and calls between them.

While common predicates are stored in a state structure, SAs are computed at execution time by co-routines. With a state that is not only declarative, with parts being procedurally computed, it is possible to minimize memory usage and delegate complex state-based operations to external methods otherwise incompatible or too complex for current planning description languages and planners that require grounding.

We abstract away the numeric parts of the planning process encoded through SAs in a layer between the symbolic planner and external libraries. We leverage the abstract architecture of Figure 1 with three layers inspired by the work of de Silva and Meneguzzi (2015). In the symbolic layer we manipulate an anchor symbol as a term, such as *polygon1*, while in the external layer we manipulate a *Polygon instance with N points* as a geometric object based on what the selected external library specifies. With this approach we avoid complex representations in the symbolic layer. Instances created by the external layer that must be exposed to the symbolic layer are compared with stored object instances to reuse a previously defined symbol or create a new one, i.e. always represent position $\langle 2,5 \rangle$ with *p1*. This process makes symbol comparison work in the planning layer even for symbols related to complex external objects. The symbol-object table is also used to transform symbols into usable object instances by external function calls and SAs. Such table is global and consistent during the planning process, as each unique symbol will map the same internal object, even if such symbol is discarded in one decomposition branch. Once operations are finished in the external layer the process happens in reverse order, objects are transformed back into symbols that are exposed by free variables. The intermediate layer acts as the foreign function interface between the two layers, and can be modified to accommodate another external library without modifications to the symbolic description.

SAs can work as interpreted predicates (Mohr et al. 2018), evaluating the truth value of a predicate procedurally, and also grounding free variables. SAs are currently limited to be used as method preconditions, which must not contain disjunctions. As only conjunctions and negations are allowed, one can reorder the preconditions during the compilation phase to improve execution time, removing the burden of the domain designer to optimize a mostly declarative description by hand, based on how free variables are used as SA

Listing 1: Abstract method with SAs among preconditions.

```
(:attachments (sa1 ?a ?b) (sa2 ?a ?b))
(:method (m ?t1 ?t2)
  label
  (; preconditions
    (call != ?t1 ?t2) ; no dependencies
    (call != ?fv1 ?fv2) ; ?fv1 and ?fv2 dependencies
    (sa1 ?t1 ?fv1)   ; no dependencies, ground ?fv1
    (pre1 ?t1 ?t2)   ; no dependencies
    (sa2 ?fv1 ?fv2)  ; ?fv1 dependency, ground ?fv2
    (pre2 ?fv3 ?fv1) ; ?fv1 dependency, ground ?fv3
  )
  (; subtasks
    (subtask ?t1 ?t2 ?fv1 ?fv2)
  )
)
```

terms. Each free variable creates a dependency between the first predicate or SA that contains such variable as a term and the next predicates or SAs that contain the same term. The first predicate or SA is responsible for grounding such variable while the next predicates or SAs only verify if the previously ground value matches with the current state. Predicates have priority over SAs to ground free variables, as the possible values are obtained from the current (finite) state, while SAs may cover a possibly infinite number of values. Consider the abstract method example of Listing 1, with two SAs among preconditions, *sa1* and *sa2*. The compiled output shown in Algorithm 1 has both SAs evaluated after common predicates, while function calls happen before or after each SA, based on which variables are ground at that point. In Line 4 the free variables *fv1* and *fv3* have a ground value that can only be read and not modified by other predicates or SAs. In Line 7 every variable is ground and the second function call can be evaluated.

Algorithm 1 Compilation phase may reorder preconditions to optimize execution time.

1: **function** M($t1, t2$)
2:    **if** t1 $\neq$ t2
3:       **for** each fv1, fv3; state $\subset \{\langle$pre1,t1,t2$\rangle, \langle$pre2,fv3,fv1$\rangle\}$ **do**
4:          **for** each sa1(t1, fv1) **do**
5:             free variable fv2
6:             **for** each sa2(fv1, fv2) **do**
7:                **if** fv1 $\neq$ fv2
8:                   decompose([$\langle$subtask, t1, t2, fv1, fv2$\rangle$])

The other limitation of current SA co-routines is that they must unify with a valid value within their internal iterations or have a stop condition, otherwise the HTN process will keep backtracking and evaluating the SA seeking new values and never returning failure. Due to the implementation support of arbitrary-precision arithmetic and accessing data from real-world streams of data/events (which are always new and potentially infinite) a valid value may never be found, and we expect the domain designer to implement mechanisms to limit the maximum number of times a SA might try to evaluate a call (i.e. to have finite stop conditions). This maximum number of tries can be implemented

as a counter in the internal state of a SA, which is mostly used to mark values provided to the HTN to avoid repetition, but may achieve side-effects in external structures. The amount of side-effects in both external functions calls and SAs increase the complexity of correctness proofs and the ability to inspect and debug domain descriptions.

Examples

Discrete distance between objects

A common problem when moving in dynamic and continuous environments is to check for object collisions, as agents and objects do not move across tiles in a grid. One solution is to calculate the distance between both objects centroid positions and verify if this value is in a safe margin before considering which action to take. To avoid the many geometric elements involved in this process we can map centroid position symbols to coordinate instances and only check the symbol returned from the symbol-object table, ignoring specific numeric details and comparing a symbol to verify if objects are near enough to collide. This process is illustrated in Figure 2, in which p_0 and p_1 are centroid position symbols that match symbols S_0 and S_1 in the symbol-object table, which maps their value to point objects O_0 and O_1. Such internal objects are used to compute *distance* and return a symbolic distance in situations where the actual numeric value is unnecessary.
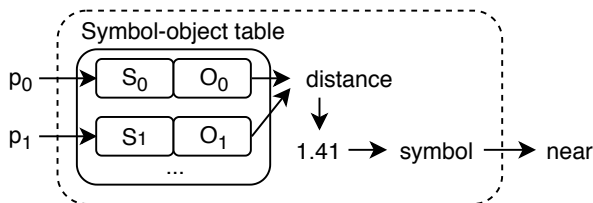

Figure 2: The symbol to object table maps symbols to object-oriented programming instances to hide procedural logic from the symbolic layer.

An iterator for HTN

In order to find a correct number to match a spatial or temporal constraint one may want to describe the relevant interval and precision to limit the amount of possibilities without having to discretely add each value to the state. Planning descriptions usually do not contain information about numeric intervals and precision, and if there is a way to add such information it is through the planner itself, as global definitions applied to all numeric functions, i.e. timestep, mantissa and exponent digits of DiNo (Piotrowski et al. 2016). The STEP SA described in Algorithm 2 addresses this problem, unifying t with one number at time inside the given interval with an ϵ step.

Lazy adjacency evaluation

To avoid having complex effects in the move operators one must not update adjacencies between planning objects during the planning process. Instead one must update only the

Algorithm 2 The STEP SA replaces the pointer of t with a numeric symbol before resuming control to the HTN.

```
1: function STEP(t, min = 0, max = ∞, ε = 1)
2:     for i ← min to max step ε do
3:         t ← symbol(i)
4:         yield                              ▷ Resume HTN
```

object position, deleting from the old position and adding the new position. Such positions come from a partitioned space, previously defined by the user. The positions and their adjacencies are either used to generate and store ground operators or stored as part of the state. To avoid both one could implement adjacency as a co-routine while hiding numeric properties of objects, such as position. Algorithm 3 shows the main two cases that appear in planning descriptions. In the first case both symbols are ground, and the co-routine resumes when both objects are adjacent, doing nothing otherwise, failing the precondition. In the second case $s2$, the second symbol, is free to be unified using $s1$, the first symbol, and a set of directions D to yield new positions to replace $s2$ pointer with a valid position, one at a time. In other terms, this co-routine either checks whether $s2$ is adjacent to $s1$ or tries to find a value adjacent to $s1$ binding it to $s2$ if such value exists.

Algorithm 3 This ADJACENT SA implementation can either check if two symbols map to adjacent positions or generate new positions and their symbols to unify $s2$.

```
1: D ← {(-1,-1),(0,-1),(1,-1),(-1,0),(1,0),(-1,1),(0,1),(1,1)}
2: function ADJACENT(s1, s2)
3:     s1 ← object(s1)
4:     if s2 is ground
5:         s2 ← object(s2)
6:         if |x(s1) - x(s2)| ≤ 1 ∧ | y(s1) - y(s2)| ≤ 1
7:             yield
8:     else if s2 is free
9:         for each (x, y) ∈ D do
10:            nx ← x + x(s1); ny ← y + y(s1)
11:            if 0 ≤ nx < WIDTH ∧ 0 ≤ ny < HEIGHT
12:                s2 ← symbol(⟨nx, ny⟩)
13:                yield
```

Domains and Experiments

We conducted empririrical tests in a machine with Dual 6-core Xeon CPUs @2GHz / 48GB memory, repeating experiments three times to obtain an average. The results show a substantial speedup over the original classical description from ENHSP (Scala et al. 2016) with more complex descriptions. Our HTN implementation is available at `github.com/Maumagnaguagno/HyperTensioN_U`.

Plant Watering / Gardening

In the Plant Watering domain (Frances and Geffner 2015) one or more agents move in a 2D grid-based scenario to reach taps to obtain certain amounts of water and pour water in plants spread across the grid. Each agent can carry up to a certain amount of water and each plant requires a certain amount of water to be poured. Many state variables can be represented as numeric fluents, such as the

Listing 2: Excerpt of the Plant Watering HTN domain used as input to our implementation, the ADJACENT SA is described separately.

```
(:attachments (adjacent ?x ?y ?nx ?ny ?gx ?gy))
(:method (travel ?a ?gx ?gy)
  base
  (; preconditions
    (call = (call function (x ?a)) ?gx)
    (call = (call function (y ?a)) ?gy)
  )
  () ; empty subtasks
  keep_moving
  (; preconditions
    (adjacent (call function (x ?a))
      (call function (y ?a)) ?nx ?ny ?gx ?gy)
  )
  (; subtasks
    (!move ?a ?nx ?ny)
    (travel ?a ?gx ?gy)
  )
)
```

coordinates of each agent, tap and plant, the amount of water to be poured and being carried by each agent, and the limits of how much water can be carried and the size of the grid. There are two common problems in this scenario, the first is to travel to either a tap or a plant, the second is the top level strategy. To avoid considering multiple paths in the decomposition process one can try to move straight to the goal first, and only to the goal in scenarios without obstacles, which simplifies the travel method. To achieve this straightforward movement we modify the ADJACENT SA to consider the goal position also, using an implementation of Algorithm 4. The top level strategy may consider which plant is closer to a tap or closer to an agent, how much water an agent can carry and so on. The simpler top level strategy is to verify how much water must be poured to a plant, travel to a tap, load water, travel to the previously selected plant and pour all the water loaded. Repeating this process until every plant has enough water poured. The travel method description using our modified JSHOP input language is shown in Listing 2 and compiled to Algorithm 5. We compare with the fastest satisficing configurations of ENHSP (sat and c_sat) in Figure 3, which shows that our approach is faster with execution times constantly below 0.01s, with both planners obtaining non-step-optimal plans.

Algorithm 4 In this goal-driven ADJACENT SA the positions are coordinate pairs, and two variables must be unified to a closer to the goal position in an obstacle-free scenario.

1: **function** ADJACENT(x, y, nx, ny, gx, gy)
2:     $x \leftarrow$ numeric(x); $y \leftarrow$ numeric(y)
3:     $gx \leftarrow$ numeric(gx); $gy \leftarrow$ numeric(gy)
4:     ▷ compare returns -1, 0, 1 for $<, =, >$, respectively
5:     $nx \leftarrow$ symbol($x +$ compare(gx, x))
6:     $ny \leftarrow$ symbol($y +$ compare(gy, y))
7:     **yield**

Car Linear

In the Car Linear domain (Bryce et al. 2015) the goal is to control the acceleration of a car, which has a minimum and maximum speed, without external forces applied, only moving through one axis to reach its destination, and requiring a small speed to safely

---

Algorithm 5 Compiled output of the Plant Watering HTN domain excerpt from Listing 2.

1: **function** TRAVEL(a, gx, gy)
2:     **if** x(a) = gx ∧ y(a) = gy
3:         decompose([])
4:     free variables nx, ny
5:     **for** each adjacent(x(a), y(a), nx, ny, gx, gy) **do**
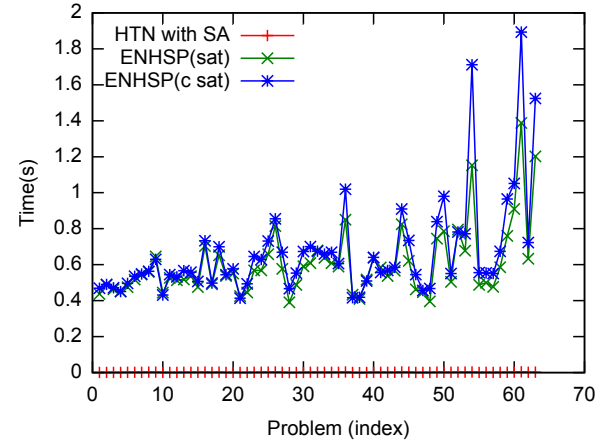6:   decompose([⟨move, a, nx, ny⟩, ⟨travel, a, gx, gy⟩])

Figure 3: Time in seconds to solve Plant Watering problems.

stop. The idea is to propagate process effects to state functions, in this case acceleration to speed and speed to position, while being constrained to an acceptable speed and acceleration. The planner must decide when and for how long to increase or decrease acceleration, therefore becoming a temporal planning problem. We use a STEP SA to iterate over the time variable and propagate temporal effects and constraints, i.e. speed at time t. We compare the execution time of our approach with ENHSP with aibr, ENHSP main configuration for planning with autonomous processes, in Table 1. There is no comparison with a native HTN approach, as one would have to add a discrete finite set of time predicates (e.g. ⟨*time 0*⟩) to the initial state description to be selected as time points during planning.

Problem	1	2	3	4	5	6	7	8	9
ENHSP (aibr)	0.461	0.462	0.427	0.461	0.475	0.474	0.443	0.466	58.256
HTN with SA	0.015	0.015	0.015	0.015	0.015	0.015	0.015	0.015	03.920

Table 1: Time in seconds to solve Car Linear problems.

Bitangent movement

For an agent to move in a continuous space it is common practice to simplify the environment to simpler geometric shapes for faster collision evaluation. One possible simplification is to find a circle or sphere that contains each obstacle and use this new shape to evaluate paths. In this context the best path is the one with the shortest lines between initial position and goal, considering bitangent lines between each simplified obstacle plus the amount of arc traversed on their borders, also know as Dubins path (Dubins 1957). One possible approach for a satisficing plan is to move straight to the goal or to the closest obstacle to the goal and repeat the process. A precondition to such movement is to have a visible target, without any other obstacle between the current and target positions. A

second consideration is the entrance direction, as clock or counter-clockwise, to avoid cusped edges. Cusped edges are not part of optimal realistic paths, as the moving agent would have to turn around over a single point instead of changing its direction a few degrees to either side. For the problem defined in Figure 4 the possible paths from point *i* to *g* are ACG, ADH, BEG, BFH.
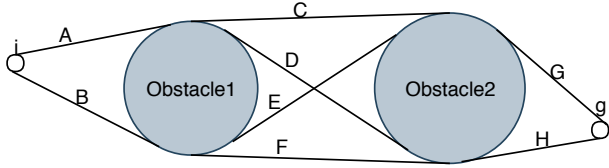

Figure 4: Possible bitangent paths from *i* to *g* with two circular obstacles.

Two possible approaches can be taken to solve the search over circular obstacles using bitangents. One is to rely on an external solver to compute the entire path, a motion planner, which could happen during or after HTN decomposition has taken place. When done during HTN decomposition, as seen in Listing 3, one must call the SEARCH-CIRCULAR function and consume the resulting steps of the plan stored in the intermediate layer, not knowing about how close to the goal it could reach in case of failure. When done after HTN decomposition, one must replace certain dummy operators of the HTN plan and replan in case of failure. The second approach is to rely on parts of the external search, namely the VISIBLE function and CLOSEST SA, to describe continuous search to the HTN planner. The VISIBLE function returns true if from a point on a circle one can see the goal, false otherwise. The CLOSEST SA generates unifications from a circle with an entrance direction to a point in another circle with an exit direction, new points closer to the goal are generated first. Differently from external search, one can deal with failure at any moment, while being able to modify behavior with the same external parts, such as the initial direction the search starts with. Another advantage over the original solution is the ability to ask for N plans, which forces the HTN to backtrack after each plan is found and explore a different path until the amount of plans found equals N or the HTN planner fails to backtrack. A description of such approach is show in Listing 4. The execution time variance between the solutions is not as important as their different approaches to obtain a result, from an external greedy best-first search to a HTN depth-first search. The external search also computes bitangents on demand, as bitangent precomputation takes a significant amount of time for many obstacles.

Conclusion

We developed a notion of semantic attachments for HTN planners that not only allows a domain expert to easily define external numerical functions for real-world domains, but also provides substantial improvements on planning speed over comparable classical planning approaches. The use of semantic attachments improves the planning speed as one can express a potentially infinite state representation with procedures that can be exploited by a strategy described as HTN tasks. As only semantic attachments present in the path decomposed during planning are evaluated, a smaller amount of time is required when compared with approaches that precompute every possible value during operator grounding. Our description language is arguably more readable than the commonly used strategy of developing a domain specific planner with customized heuristics. Specifically, we allow designers to easily define external functions in a way that is readable within the domain

Listing 3: Search over circular obstacles using bitangents is done entirely by external function and resulting plan steps stored in intermediate layer are consumed by the HTN.

```
(:method (forward ?agent ?goal)
  base
  ((at ?agent ?goal)) ; preconditions
  () ; empty subtasks
  search
  (; preconditions
    (at ?agent ?start)
    (call search-circular ?agent ?start ?goal)
  )
  ; subtasks
  ((apply-plan ?agent ?start 0 (call plan-size)))
)
(:method (apply-plan ?agent ?from ?index ?size)
  index-equals-size
  ((call = ?index ?size)) ; preconditions
  () ; empty subtasks
  get-next-action
  ; preconditions
  ((assign ?to (call plan-position ?index)))
  (; subtasks
    (!move ?agent ?from ?to)
    (apply-plan ?agent ?to (call + ?index 1) ?size)
  )
)
```

Listing 4: Search over circular obstacles using bitangents is done by the HTN using CLOSEST SA to generate each step.

```
(:attachments (closest ?circle ?to ?outcircle
  ?indir ?outdir ?goal))
(:method (forward-attachments ?agent ?goal)
  clockwise
  ((at ?agent ?start)) ; preconditions
  (; subtasks
    (loop ?agent ?start ?start clock ?goal)
  )
  counter-clockwise
  ((at ?agent ?start)) ; preconditions
  (; subtasks
    (loop ?agent ?start ?start counter ?goal)
  )
)
(:method (loop ?agent ?from ?circle ?indir ?goal)
  base
  ((call visible ?from ?circle ?goal)) ; preconditions
  ((!move ?agent ?from ?goal)) ; subtasks
  recursion
  (; preconditions
    (closest ?circle ?to ?outcircle
      ?indir ?outdir ?goal)
    (not (visited ?agent ?to))
  )
  (; subtasks
    (!move ?agent ?from ?to)
    (!!visit ?agent ?from)
    (loop ?agent ?to ?outcircle ?outdir ?goal)
    (!!unvisit ?agent ?from)
  )
)
```

knowledge encoded in HTN methods at design time, and also dynamically generate symbolic representations of external values at planning time, which makes generated plans easier to understand.

Our work is the first attempt at defining the syntax and operation of semantic attachments for HTNs, allowing further research on search in SA-enabled domains within HTN planners. Future work includes implementing a cache to reuse previous values from external procedures applied to similar previous states (Dornhege, Hertle, and Nebel 2013) and a generic construction to access such values in the symbolic layer, to obtain data from explored branches outside the state structure, i.e. to hold mutually exclusive predicate information. We plan to develop more domains, with varying levels of domain knowledge and SA usage, to obtain better comparison with other planners and their resulting plan quality. The advantage of being able to exploit external implementations conflicts with the ability to incorporate such domain knowledge into heuristic functions, as such knowledge is outside the description. Further work is required to expose possible metrics from a SA to heuristic functions.

References

Bryce, D.; Gao, S.; Musliner, D. J.; and Goldman, R. P. 2015. SMT-Based Nonlinear PDDL+ Planning. In *AAAI*, 3247–3253.

Dahl, O.-J.; Dijkstra, E. W.; and Hoare, C. A. R. 1972. *Structured programming*. Academic Press Ltd.

de Silva, L., and Meneguzzi, F. 2015. On the design of symbolic-geometric online planning systems. In *2015 Workshop on Hybrid Reasoning (HR 2015)*, 1–8.

Dornhege, C.; Gissler, M.; Teschner, M.; and Nebel, B. 2009. Integrating symbolic and geometric planning for mobile manipulation. In *Safety, Security & Rescue Robotics (SSRR), 2009 IEEE International Workshop on*, 1–6. IEEE.

Dornhege, C.; Hertle, A.; and Nebel, B. 2013. Lazy evaluation and subsumption caching for search-based integrated task and motion planning. In *IROS workshop on AI-based robotics*.

Dubins, L. E. 1957. On curves of minimal length with a constraint on average curvature, and with prescribed initial and terminal positions and tangents. *American Journal of mathematics* 79(3):497–516.

Fox, M., and Long, D. 2003. PDDL 2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research (JAIR)*.

Frances, G., and Geffner, H. 2015. Modeling and computation in planning: Better heuristics from more expressive languages. In *ICAPS*, 70–78.

Francès, G.; Ramírez, M.; Lipovetzky, N.; and Geffner, H. 2017. Purely declarative action descriptions are overrated: Classical planning with simulators. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, 4294–4301.

Gerevini, A. E.; Saetti, A.; and Serina, I. 2008. An approach to efficient planning with numerical fluents and multi-criteria plan quality. *Artificial Intelligence* 172(8-9):899–944.

Ilghami, O., and Nau, D. S. 2003. A general approach to synthesize problem-specific planners. Technical report, DTIC Document.

McDermott, D.; Ghallab, M.; Howe, A.; Knoblock, C.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. PDDL-the planning domain definition language. Technical report, Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control.

Mohr, F.; Lettmann, T.; Hüllermeier, E.; and Wever, M. 2018. Programmatic Task Network Planning. In *Proceedings of the 1st ICAPS Workshop on Hierarchical Planning*, 31–39.

Nau, D.; Cao, Y.; Lotem, A.; and Muñoz-Avila, H. 1999. SHOP: Simple hierarchical ordered planner. In *Proceedings of the 16th international joint conference on Artificial Intelligence-Volume 2*, 968–973. Morgan Kaufmann Publishers Inc.

Nebel, B. 2000. On the Compilability and Expressive Power of Propositional Planning Formalisms. *Journal of Artificial Intelligence Research* 12:271–315.

Piotrowski, W. M.; Fox, M.; Long, D.; Magazzeni, D.; and Mercorio, F. 2016. Heuristic Planning for PDDL+ Domains. In *AAAI Workshop: Planning for Hybrid Systems*.

Scala, E.; Haslum, P.; Thiébaux, S.; and Ramirez, M. 2016. Interval-based relaxation for general numeric planning. In *ECAI*, 655–663.

Strobel, V., and Kirsch, A. 2014. Planning in the wild: modeling tools for PDDL. In *Joint German/Austrian Conference on Artificial Intelligence (Künstliche Intelligenz)*, 273–284. Springer.

Şucan, I. A., and Kavraki, L. E. 2011. Mobile manipulation: Encoding motion planning options using task motion multigraphs. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, 5492–5498. IEEE.

Weyhrauch, R. W. 1981. Prolegomena to a theory of mechanized formal reasoning. In *Readings in Artificial Intelligence*. Elsevier. 173–191.