Urban Modeling via Hierarchical Task Network Planning

Michael Staud

StaudSoft UG, Ravensburg , Germany
michael.staud@staudsoft.com

Abstract

In this paper we present a new method for city modeling based on hierarchical task network planning. The planner creates actions that are executed in a city simulation. These actions generate step by step a city model within the simulation.

The advantage of this approach is that it takes into account that real cities are not designed on a drawing board, but have a history of development. By simulating this development, economic aspects can be taken into account. The result is a much more realistic urban model.

An urban simulation is an extremely complex planning domain for a planner. Therefore, we have developed a new domain-independent hierarchical task network planning algorithm that divides the planning problem into smaller planning problems. Our planning algorithm is sound and complete. We give preliminary results on its performance.

1 Introduction

Cities play a very important role in our daily lives. Most of our life takes place in a city, cities have shaped our social development. Therefore, it is important to be able to create models of cities quickly and efficiently. This field is called urban modeling. It involves either creating virtual cities based on existing cities or creating virtual cities based on parameters (Kelly and McCabe 2007). These can then be used in movies, games or artificial intelligence (Formichella, Lucien 2020; Glockner, Cyrill 2018).

Virtual cities today are generated through *procedural generation* which is used to mimic the work of artists (Freiknecht and Effelsberg 2017; Van Der Linden, Lopes, and Bidarra 2013). Procedural generation itself is not limited to generating cities, but can also be used to create textures, sounds, models or even stories (Togelius et al. 2013). A popular approach is to use a *modified L-system*(Rozenberg and Salomaa 1980) to create the road network of an urban environment. Buildings, also generated using an L-system, are then placed between the roads (Parish and Müller 2001). Other approaches include agent-based simulations (Lechner, Watson, and Wilensky 2003), template-based generation (Sun et al. 2002) or grid layouts (Kelly and McCabe 2006).

The problem with all of the approaches mentioned so far is that they only attempt to mimic the structure of a city. They do not create the structure of a city, nor do they deal with growth and development over time. However, a city grows depending on its economic structure. Businesses or factories grow only where they are profitable. Residential areas grow only when jobs are available. In addition, our approach generates a lot of other information that is useful for a credible simulation (building type, traffic routes, ...).

1.1 Our Approach

We therefore propose a new way to model cities by formulating the complex interactions in a city as a planning domain. To create a city, a planning problem is defined that contains information about the goal (e.g., to create a city of a certain size) and the environment. The environment consists of the terrain, which is defined as a graph and contains information about whether a node is buildable, whether there is water, and what resources are available (see Section 5).

One problem we faced in our approach was that such a planning problem is too large to be handled by current planners. To solve this problem, we use a new type of domain-independent hierarchical task network planning (HTN planning) (Ghallab, Nau, and Traverso 2004). We defined a new type of abstract tasks that are defined by the domain designer. We call them *separable abstract tasks* (see Section 3). When our planning algorithm encounters such a task, it generates a new independent planning problem from it and tries to solve it independently of the main problem. This process can be recursive. We call this a planning process (see Section 4.2).

To make this work, we use what we call a *hierarchical world state*. Several abstraction layers are defined in the domain. The number of layers is stored in the variable n. Each layer contains the same information, but in a more abstract form. Layer one is the most abstracted. The layer n contains only non-abstracted information (see Section 3).

Our contribution is:

- A new sound and complete domain-independent HTN planning algorithm (see Section 4).
- The application of HTN planning to urban modeling (see Section 5).

In the next section, we will first describe HTN planning in detail and then our new approach. Then we will describe the domain we used to create an urban simulation. After that, we will present some preliminary results.

2 Hierarchical Task Planning

We define the set of all constants as C and the set of all variables as V. A literal is an atom or its negation. The set of all atoms is denoted by A. An atom is a predicate applied to a tuple of terms. A term can be a constant or a variable.

The following definitions are adapted from Bercher, Keen, and Biundo (2014). We use total-order hierarchical task network planning (Ghallab, Nau, and Traverso 2004, 238). A hierarchical planning domain is a tuple $D = (T_a, T_p, M)$ containing 3 finite sets. T_a is the set of abstract tasks, T_p is the set of primitive tasks, and M is the set of methods. Primitive and abstract tasks are also tuples of the form $t(\overline{\tau}) = \langle prec_t(\overline{\tau}), \mathit{eff}_t(\overline{\tau})\rangle$. Each task has a precondition $prec_t(\overline{\tau})$ and an effect $\mathit{eff}_t(\overline{\tau})$. Moreover, each task has a set of parameters $\overline{\tau}$. If all parameters of a primitive task are equal to atoms, the task is grounded and is called an action. A method is a tuple $m = \langle t_a(\overline{\tau}_m), P_m\rangle$, where $t_a(\overline{\tau}_m)$ is the abstract task it can replace, P_m is a set of plan steps and $\overline{\tau}_m$ are the parameters of the method. A plan step is a uniquely labeled task $l : t(\overline{\tau})$. A plan is also a sequence of plan steps. A solution is a plan in which each task is an action and its preconditions are satisfied at each step. In addition, the solution transforms the initial state into the goal state. A task can be executed in a particular world state only if its preconditions are satisfied. The preconditions, which are a set of literals, are satisfied if every positive literal is in the current world state and if every negative literal is not in the current world state. The effects of a task are also a set of literals. Positive literals add atoms to the world state when the task is "executed". Negative literals remove an atom from the world state. The function $\sigma : \mathcal{P} \times T_p \to \mathcal{P}$ applies the effects of a primitive task to a world state.

A plan step $l : t(\overline{\tau})$ associated with an abstract task can be decomposed by a method $m = \langle t_a(\overline{\tau}_m), P_m\rangle$, where the plan step l in the current plan is replaced by the plan steps in P_m. A world state w is a set of atoms. A *derived predicate* (Edelkamp and Hoffmann 2004) is defined by a rule containing a logical formula with variables. If the rule of a predicate evaluates to true, it is added to the world state. Otherwise, it is removed. The logical formula may contain quantifiers.

A problem is a tuple $P = \langle \mathit{init}_P, \mathit{goal}_P, PS_P\rangle$, where init_P is the initial state consisting of atoms, the goal goal_P is a set of literals. The initial plan is stored in PS_P.

3 Hierarchical World State

The world state is divided into n layers. Each predicate and task has an associated *hierarchical layer* l. The layer 1 has the highest level of abstraction (see Section 5 for an example). The layer with the highest index n stores the actual, non-abstracted facts of the simulation. A task can only change predicates on the same layer. To jump from layer l to layer $l + 1$ during the planning process, we introduce a new type of abstract tasks, the *separable abstract tasks*. They are not directly decomposed into a method if they do not appear in the initial plan init_P. Instead, they give the planner a hint that this abstract task needs to be decomposed in another planning process that operates at layer $l + 1$. It should be noted that a separable abstract task can occur in the initial state of one planning process and by decomposition in another planning process. This is the reason why we do not simply use normal abstract tasks in the initial state. How the planning processes operates in detail is described in Section 4.2.

3.1 Information Flow to the Layer $l + 1$

The following rules enforce the separation of layers:

- A task of layer i can only use predicates in its precondition of layer $l \leq i$.
- A task can only have predicates in its effects of the same layer as itself.
- Primitive tasks are allowed only in the highest layer n.
- Normal abstract tasks cannot have effects.
- Separable abstract tasks may only occur in the layers $l < n$.
- Normal abstract tasks of layer l can only be decomposed into plan steps with tasks of layer l.
- Separable abstract tasks of layer l can only be decomposed into plan steps with tasks of layer $l + 1$.

Therefore, the initial tasks belong to layer 1 and it must be ensured by the domain developer that they can be decomposed into the primitive tasks across the n layers.

3.2 Information Flow to the Layer $l - 1$

To pass information from a layer l to a layer $l - 1$, all predicates on the layers $l < n$ must be derived predicates unless they never occur in an effect of a task. A derived predicate of layer l can only use predicates in its formula that come from layer $l+1$. Derived predicates of layer l' are only updated in the world state of the main system or in a planning instance of layer l if $l' < l$ holds (see Section 4). If the predicates have the same layer as the planning instance, they are not treated as derived predicates. Thus, effects that change derived predicates do not cause inconsistencies in a planning process of layer l, since a task of layer l can only change predicates of layer l. And there is also no inconsistency in the world state of the main system, since only primitive tasks are applied to it and they cannot change derived predicates.

4 Planning Algorithm

The general idea is to split the planning process into many smaller processes. This can increase the performance by an exponential factor (Korf 1987). The system consists of 2 main modules (see Figure 1):

- **Main System**: stores the *main plan* that is created during the planning process. It also stores the world state w, which is initialized by the initial state init_P of the problem. The main system always contains the entire world state (see Algorithm 1).
  The main plan is a series of plan steps of primitive tasks. When the planning algorithm is finished, it will contain the plan which will transform the initial state into the goal state, if this is possible.
- **Planning Processes**: These are small planning problems that are solved during the planning process and operate on a particular layer l of the world state (but they can contain predicates of a level $l' < l$). At each step of the

main system, all planning processes are invoked. When not suspended, a process takes the current world state of the main system as input to provide the next action to be added to the main plan. However, it does not use the entire world state for planning, only the horizon (see Section 4.1). Similar to adversarial search, no complete plan is created, only the next best action is returned (though the other possible actions are stored to allow backtracking).

If it is not possible to generate a next action, the entire system (main system and planning processes) will backtrack. This means that the action added to the main plan is removed, planning processes can be removed or deleted. This process continues until a step is found where another choice point can be selected in the search tree of a planning process. In our practical experiments, backtracking was never used because each planning process always found a solution that was valid in the world state of the main system.

4.1 Horizon

The horizon is a set containing all atoms that can be used in a planning problem. By "used" it is meant that they may appear in a precondition or effect of a task during the planning process. Formally, this means that the set contains exactly those atoms that occur in each task of the task decomposition graph of a planning process (Bercher, Keen, and Biundo 2014). When constructing the task decomposition graph, separable abstract tasks are treated in the same way as in the planning process. They are not decomposed unless they occur in the initial plan of the planning problem. When derived predicates occur in the graph, the predicates on which they depend are added to the horizon only if their level l is equal to or less than the level of the planning process (see Section 4.2). All this makes the horizon much smaller than the original problem and increases planning performance (both in terms of time and memory).

4.2 Planning Processes

When a planning process plans, it treats separable abstract tasks as primitive tasks unless they occur in the initial plan of the planning problem. If they do occur in the initial plan, they are decomposed like a normal abstract task. Similar to primitive tasks, separable abstract tasks are passed to the main system. However, they are then handled differently. Instead of adding them to the main plan, a new subplanning process is started by the main system and added to the list of planning processes. The planning process that tried to execute the separable abstract task is suspended until the new subplanning process finishes its task. The subordinate planning process can in turn create new subordinate processes. This is illustrated in Figure 1.

- **Primary Planning Process**: This process directly tries to solve the given problem. This means that its *initial state* and *goal* match with the one defined in the problem. However, it does so only within its horizon (see Section 4.1) using derived tasks and separable abstract tasks. It is initially generated by the main system and not by a separable task. Layer 1 is assigned to it. Therefore, the goal $goal_P$ can only contain predicates associated with layer 1.

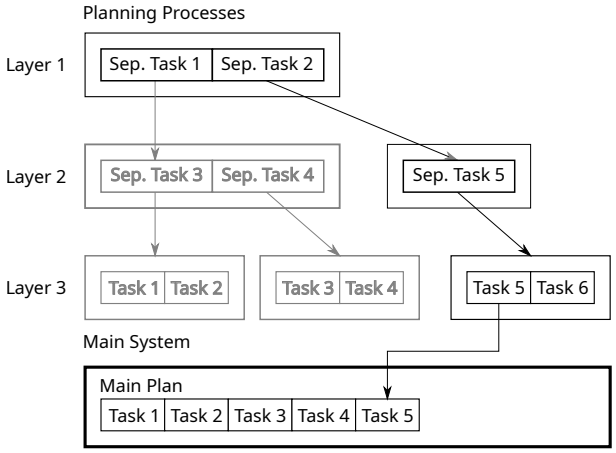

Figure 1: Example of a planning process in a domain with 3 layers. The grey processes are already completed.

- **Instanced Planning Process**: This is created whenever a separable abstract task of layer l is returned by a planning process as the next task. The *goal* of the planning process is equal to the effects of the separable abstract task. The *initial plan* contains only the separable abstract task. It takes the world state of the main system at each step and then determines the task that will be returned to the main system. So the *initial state* is equal to the current world state of the main system. The layer $l+1$ is assigned to it.

A single planning process is much easier to solve than the full planning problem because it plans only within its own horizon (see Section 4.1).

The algorithm of the main system can be seen in Algorithm 1. The initial problem is the tuple $P = \langle init_P, goal_P, PS_P \rangle$. A planning process is a tuple $p_p = (H_s, G, P_T, S, l)$, where H_s is the current task stack containing which primitive tasks to execute and which abstract tasks to decompose. G stores the goal of the planning process. P_T is the parent planning process that produced this process. For the primary planning process, it holds $P_T = \bot$. Whether the process is suspended because it is waiting until another planning process is finished is stored in $S \in \{\top, \bot\}$. The layer of the planning process is stored in l. The $getNextTask$ function triggers the planning algorithm of a process. It returns $\bot$ if the process was suspended or if failed. Otherwise, it returns a separable abstract task or a primitive task. Abstract tasks are never returned. The derived predicates are updated in $update\_hierarchical\_state(w)$.

Our planning system uses the Monte Carlo tree search algorithm (Kocsis and Szepesvári 2006) in the planning processes to determine the next action to add to the main system. It uses forward decomposition (Ghallab, Nau, and Traverso 2004, 238). We use the H0 heuristic (Ghallab, Nau, and Traverso 2004, 201) to estimate the distance to the target in the playouts.

Algorithm 1: The algorithm in the main system. The main plan is stored in the sequence m.

$P_l = \{(PS_P, G, \bot, \bot, 1)\}$
$w = init_P$
$m = ()$
repeat
  for $p_p = (H_p, G_p, P_p, S_p, l_p) \in P_l$ **do**
    $t = getNextTask(w, p_p)$
    if $t = \bot$ **then**
      continue or backtrack if p_p failed
    end if
    if $t \in T_p$ **then**
      $w = \sigma(w, t), m = (m_1, \ldots, m_{|m|}, t)$
      $w = update\_hierarchical\_state(w)$
      if $finished(p_p)$ **then**
        $P_l = P_l \setminus \{p_p\}$
        $unsuspend(P_p)$
      end if
    else
      $p_{new} = ((t), eff_t, p_p, \bot, l_p + 1)$
      $P_l = P_l \cup \{p_{new}\}$
      $suspend(p_p)$
    end if
  end for
until $isGoal(w, G)$

4.3 Theoretical Properties

The algorithm is sound because it only performs valid actions in the domain. Otherwise, it will backtrack. The algorithm finds a solution if the problem does not allow infinite decompositions of abstract tasks. In this case, the algorithm is also complete since the search space is finite. It should be noted that an HTN planning algorithm can be made complete on more unrestricted domains by performing additional checks (Nau et al. 2001).

5 Urban Simulation Domain

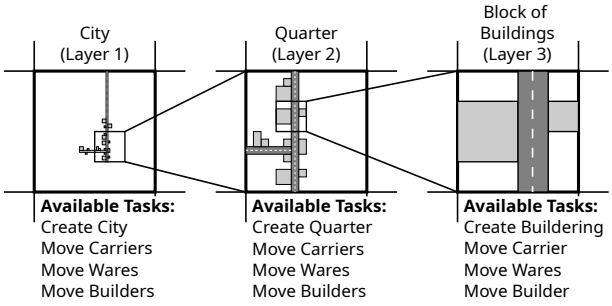

Figure 2: Hierarchical structure in the urban simulation domain.

Our planning algorithm is used for planning in an urban simulation. The rules for the simulation are declared in HDDL (Höller et al. 2020). Units and buildings are placed on a grid. Units can move on the grid. Buildings can be built at specific positions and cannot move. Each unit and building has unique capabilities. The builder unit can construct buildings. Roads are special buildings. The carrier unit can move goods between buildings. The main building can create new builder or carrier units. The woodcutter building can cut down trees and create wood. We tested an environment with a grid size of 64×64.

The domain contains separable abstract tasks for performing certain actions at different layers of the hierarchical world state. For example, creating a city, creating a neighborhood, or moving a building unit (see Figure 2). The source code of the domain is available on the Internet (Staud 2022).

6 Results

We measure how many states our algorithm (Separable Hierarchical Task Network Algorithm, SHTN for short) needs to solve multiple given goals in our urban simulation domain when using separable abstract tasks and compare it with to Forward Decomposition (Ghallab, Nau, and Traverso 2004, 238) Monte Carlo tree search (FDMCTS). This algorithm differs from ours only in that separable abstract tasks are decomposed like normal abstract tasks and thus no additional planning processes are generated. There is only the main planning instance and the hierarchical world state rules (see Section 3) are not applied.

The results can be seen in Table 1. For both of the Monte Carlo algorithms, we used a fixed number of iterations in each call to $getNextTask(w, p_p)$. In the playouts, we explored the search space in the playouts until all normal abstract tasks were randomly decomposed. Our algorithm then requires much fewer states because it does not need to explore the decomposition of separable abstract tasks. In addition, we measured the size of the horizon of the different planning instances and compared it to the size of the original planning problem. These results can be found in Table 2.

Goal	SHTN	FDMCTS
Create City	6925	6542100
Create Carrier	10716	8060040
Builder Move	38441	33479127

Table 1: Performance comparisons: Total number of states visited in the search space to reach the specified goal.

7 Conclusions

We have presented a novel HTN planning algorithm that is capable of planning in an urban simulation. The proof-of-concept shows that this approach is successful. And our tests show that it can also solve larger and more problems, as it performed very well in our experiments. Our next goal is to use Deep Learning to better predict the actual effects of separable abstract tasks. And then we want to find out how well this algorithm can perform in an adversarial environment. An opposing player in this case can be, for example, the crime rate, an enemy power in a war, or environmental influences can be modeled as opponents. These can be

Primary	Initial Atoms
Full Problem	120325
Primary Problem	338
Create City (Layer 2)	935
Create Quarter (Layer 3)	2803
Create Carriers (Layer 2)	936
Create Carriers (Layer 3)	2804
Move Builders (Layer 2)	1353
Move Builders (Layer 3)	4567

Table 2: Size of the horizons of various planning processes created by separable abstract tasks. The primary problem is the amount of atoms in the primary planning process. The full problem contains all the atoms of the problem without a horizon. The size of each horizons is on average 100 times smaller than the original planning problem.

floods, fires, or heavy rains. By modeling these as adversaries rather than random effects, the algorithm is forced to consider these effects in every part of the urban environment.

References

Bercher, P.; Keen, S.; and Biundo, S. 2014. Hybrid Planning Heuristics Based on Task Decomposition Graphs. In *SoCS 2014*, 35–43. AAAI Press.

Edelkamp, S.; and Hoffmann, J. 2004. PDDL 2.2: The Language for the Classical Part of IPC-4. In *Int. Planning Competition*.

Formichella, Lucien. 2020. Fourteen Groundbreaking Movies That Took Special Effects to New Levels. https://www.insider.com/most-groundbreaking-cgi-movies-ever-created-2020-1. Accessed: 2022-02-16.

Freiknecht, J.; and Effelsberg, W. 2017. A Survey on the Procedural Generation of Virtual Worlds. *Multimodal Technologies and Interaction*, 1(4): 27.

Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated Planning: Theory and Practice*. Elsevier.

Glockner, Cyrill. 2018. Simulators: The Key Training Environment for Applied Deep Reinforcement Learning. https://towardsdatascience.com/simulators-the-key-training-environment-for-applied-deep-reinforcement-learning-9a54353f494f. Accessed: 2022-02-16.

Höller, D.; Behnke, G.; Bercher, P.; Biundo, S.; Fiorino, H.; Pellier, D.; and Alford, R. 2020. HDDL: An Extension to PDDL for Expressing Hierarchical Planning Problems. In *Proc. of the AAAI Conference on AI*, volume 34, 9883–9891.

Kelly, G.; and McCabe, H. 2006. A Survey of Procedural Techniques for City Generation. *ITB Journal*, 14(3): 342–351.

Kelly, G.; and McCabe, H. 2007. Citygen: An Interactive System for Procedural City Generation. In *5th Int. Conf. on Game Design and Technology*, 8–16.

Kocsis, L.; and Szepesvári, C. 2006. Bandit Based Monte-Carlo Planning. In *ECML*, 282–293. Springer.

Korf, R. E. 1987. Planning as Search: A Quantitative Approach. *AI 87*, 33(1): 65–88.

Lechner, T.; Watson, B.; and Wilensky, U. 2003. Procedural City Modeling. In *1st Midwestern Graphics Conf.*

Nau, D.; Munoz-Avila, H.; Cao, Y.; Lotem, A.; and Mitchell, S. 2001. Total-Order Planning with Partially Ordered Subtasks. In *IJCAI*, volume 1, 425–430.

Parish, Y. I.; and Müller, P. 2001. Procedural Modeling of Cities. In *Proc. of SIGGRAPH-01*, 301–308.

Rozenberg, G.; and Salomaa, A. 1980. *The Mathematical Theory of L-Systems*. Academic Press.

Staud, M. 2022. Urban Simulation HDDL Domain. https://www.staudsoft.com/urbansimulation.html. Accessed: 2022-05-29.

Sun, J.; Yu, X.; Baciu, G.; and Green, M. 2002. Template-Based Generation of Road Networks for Virtual City Modeling. In *Proc. of VRST-01*, 33–40.

Togelius, J.; Champandard, A. J.; Lanzi, P. L.; Mateas, M.; Paiva, A.; Preuss, M.; and Stanley, K. O. 2013. Procedural Content Generation: Goals, Challenges and Actionable Steps. In *Artificial and Computational Intelligence in Games*, 61–75. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

Van Der Linden, R.; Lopes, R.; and Bidarra, R. 2013. Procedural Generation of Dungeons. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(1): 78–89.