Recursive Decomposition of Numeric Goals, Exemplified with Automated Construction Agents in 3D MineCraft Worlds

***Abstract*—We describe a novel approach to incorporating navigation in a general planning process. Five planning control strategies are articulated for combining "primitive rules" (which directly solve problems) with "recursive rules" (which decompose problems into combinations of problems of the same form). These are then applied to carry out navigation and generalized planning together, utilizing the observation that a "planning-aided navigation" problem can be automatically decomposed into smaller planning-aided navigation problems, and hence addressed by vertical recursive rules. The approach is illustrated in the MineCraft construction domain, via demonstrating an agent that is able to build blocks to create a path to reach an initially unreachable target, e.g. pathfinding tasks in which parts of the path must be constructed in the course of navigation/planning, such as building a bridge to fill a gap on the way to the target location.**

Keywords*— *Automatic recursive goal decomposition; recursive rules; planning-aided navigation; MineCraft agents; vertically decomposing of numeric goals;

I. INTRODUCTION

This template, modified in MS Word 2007 and saved as a "Word 97-2003 Document" for the PC, provides authors with most of the formatting specifications needed for preparing electronic versions of their papers. All standard paper components have been specified for three reasons: (1) ease of use when formatting individual papers, (2) automatic compliance to electronic requirements that facilitate the concurrent or later production of electronic products, and (3) conformity of style throughout a conference proceedings. Margins, column widths, line spacing, and type styles are built-in; examples of the type styles are provided throughout this document and are identified in italic type, within parentheses, following the example. Some components, such as multi-leveled equations, graphics, and tables are not prescribed, although the various table text styles are provided. The formatter will need to create these components, incorporating the applicable criteria that follow.

Conceptually, navigation is a kind of planning; but in practice, navigation algorithms are implemented separately from general-purpose planning frameworks. When navigation needs to be done in the context of planning, navigation is either reduced to a very high level action (so that the planner can ignore the specifics of movement toward target locations), or a separate pathfinder algorithm is called in the plan-execution phase after planning is complete. In practical applications, however, this encapsulation of navigation away from broader planning is not generally feasible. The desired target of a potential movement may be inaccessible or accessible only at large cost. It may be necessary to carry out planned interactions with the environment in order to create a path to access a desired target (e.g., to fix a broken bulb, one may need to move a ladder to the right location before climbing it), which requires unifying individual navigation steps with broader planning logic.

In the framework presented here, navigation is accomplished as a component of a more general planning process, via judicious use of propositional reasoning within a general-purpose planner. While this sort of unification has many uses, our focus has been on situations where an AI-controlled agent must construct a path in order to reach its target. We have tested our framework via deploying our navigation-incorporating planner in a 3D MineCraft-like game world implemented in the Unity3D game engine – thus exploring the domain of "MineCraft construction," in which an agent is able to build blocks to create a path to reach an initially unreachable target. However, the algorithms and ideas outlined are actually much more general and intended for application beyond this domain.

The way prior work has handled the relation between planning and navigation typifies the strain between generality and specificity that has long existed in the planning domain As Jörg Hoffmann pointed out, "What characterizes planning research is the attempt to create one planning solver that will perform sufficiently well on all possible domains"[1]. However, much of the literature actually concerns small-scale "toy problems." And when application architects encounter realistic problems that are difficult to solve beyond the "toy problem" scale with existing general planners, the standard approach is to separate the tricky aspects of the problem from the main planning process via introducing additional specialized algorithms in case-specific ways. Navigation exemplifies this pattern. In most planning domains that involve movement, navigation is excluded from the overall planning logic via assuming that the agent is able to access any location through a "move" action, (e.g. the classical Gripper task domain [2]), by or developing a special path planning algorithm optimized for navigation, e.g. ([3,4]). Obviously, both of these approaches have significant practical limitations.

There are some more advanced architectures, Zhang, Sridharan & Washington [5] use a path planner in conjunction with a global planner. In the global planning phase, whenever navigation is needed, it will call the path planner to return a path for use within global planning, so that when the path planner fails to find a path to access an object (e.g., when a door is closed or obstacles block the way), the global planner can try to access another object. This architecture enables

navigation planning to work within the main planning phase, instead of operating entirely separately or in the execution phase, while still using a special planning algorithm which is not truly encoding every step of navigation planning within the global planner.

This represents a step beyond prior work, yet still has important limitations. For instance, in this approach, even when there are actions with a path clearing effect, such as open(doorX) or move_object(obstacleX, locationY), the global planner is still unable to create a path. Since the pathfinder is a separate module, the global planner cannot know the nature of the problem that caused the pathfinding to fail. Even if the system could learn from experience, it would still be hard for the global planner to decide which location to move the obstacles to.

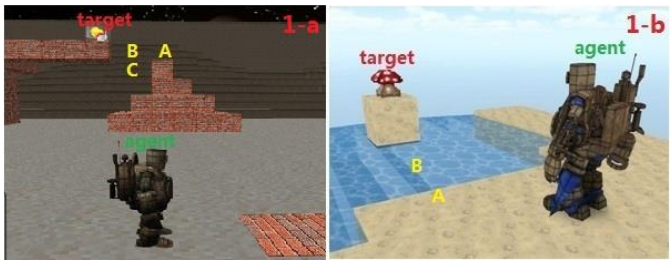

Fig. 1. Unreachable targets in a MineCraft world

Figure 1 illustrates two example path-finding problems which cannot be solved by the previously mentioned planning architectures. In Figure 1-a, a character is seeking a target which is on a beam that cannot be climbed to using existing structures; while in Figure 1-b, the target is in the middle of the water. A pathfinder will only return its failure to find a path to access the targets, but cannot tell why. So the critical problem here is to figure out that the gap between A and the target is the reason that pathfinding failed (which is very easy for human but very hard for computer), and then to come up with a solution to fill blocks at B or C to complete the path. Such problems require vertical decomposition of the goal until the exact subgoal which is the main cause of the problem is found.

II. Planning Problem: Automatic Vertical Decomposition of Numeric Goals

A. Vertical Decomposition of Numeric Goals

A STRIPS-like [[6]] numeric planning problem is a 6-tuple $\Pi = (S, S_{sen}, R, A, V, I, G)$, where S is a finite set of states; S_{sen} is a set of states returned by planning time sensors (see section Planning-time Sensing); R is a finite set of rules, each rule $r \in R$ define a state transition model, which contains an action $a \in A$, $Pre(a) \subseteq S + S_{sen}$ is the precondition set of a in this rule r, and $Eff(a) \subseteq S + S_{sen}$ is the effect set of a in this rule r; V is a set of variables used in R, with all the specified ranges for each variable $v \subseteq V$; $I, G \subseteq S + S_{sen}$, I is the initial state, G is the goal. To clarify the notions in this paper, consider a space of goal-functions G to be optimized, and a space of parametrized goal-achieving strategies, from which a certain subset f(G) may be identified as "applicable" candidates for a given goal G.

Next, suppose one has an initial goal G, which has been decomposed into subgoals $G = \{g_1, g_2,..., g_n\}$. Existing research on goal decomposition includes both manual decomposition as in HTN planners (e.g., [[7],[8]]), and automatic decomposition. Most studies of the latter focus on what we call horizontal decomposition, where one has goals that can be decomposed into different subgoals with independent solutions (e.g., [[9], [10], [11]]). In such cases, the main issue is to sort out the positive and negative interactions among solutions for subgoals so as to combine them in a correct order - for example, to boil an egg, one needs to place an egg in a pan, fill the pan with water, put the pan on the cooking hub, turn on the fire … and so on. However, the core contribution of the our approach lies in what we call vertical decomposition of numeric goals:

- ***Def. 1: Vertical Decomposition*** -- A goal decomposition $G = \{g1, g2,..., gn\}$is vertical if the applicable candidate strategies are the same for G and for every subgoal gi in G.

Intuitively, this means that every subgoal g_i is the same problem as G at a smaller scale. For example, in Fig.1-a, the goal to go to the target location (25,65,100) can be decomposed into a series of subgoals: go to A (25, 67,100), go to B (25, 66,100), go to target location. These subgoals are all in the same form as the initial goal, in smaller scales. Of course achieving the subgoal "go to B", requires to build a block at C to make the agent able to stand at B (stand on C), but this is not the main difficulty. The main difficulty is to automatically find out that B is the gap that need to be filled. In general, the main difficulty for vertical decomposition of numeric goals is to figure out which parts / sections are the main cause of the current unsatisfaction of the initial numeric goal.

As explained by Yu (Yu et al. 2004), recursive algorithms are the most efficient approach to vertical decomposition. However, Yu's approach involves manual recursive decomposition, whereas we have we encoded the automatic recursive decomposition process in a general planning framework, and developed 5 strategies for efficient decomposition in this context. Many real-world problems require vertical decomposition of this sort.

B. Definitions of Recursive Rules and Primitive Rules

In our approach, Recursive Rules and Primitive Rules are defined as the base of vertical decomposition of numeric goals:

- ***Def. 2: Recursive Rules*** -- For a rule R, with a precondition set $P = P1 \cup P2,... \cup Pn$ and an effect set E, if any Pi , $i \in \{1,2...n\}$ has $Pi \subset E$, then rule R is a recursive rule.

Functionally, a recursive rule breaks a goal into multiple subgoals which describe the same problem as the initial goal but in a smaller scale.

Many rules involve the carrying out of actions. However, another kind of rules is a propositional rule, which infers state values from other known states and requires no action, similar to "derived predicates" in PDDL2.2. Recursive rules are usually propositional rules.

- ***Def. 3: Primitive Rules*** -- Rules that are not Recursive Rules are Primitive Rules.
- ***Def. 4: Corresponding Recursive and Primitive Rules*** -- Assume one has a rule set Ω, in which each rule $Rr \in \Omega$ is a Recursive Rule; and a rule set Θ, in which each rule $Rp \in \Theta$ is a Primitive Rule. Suppose that each Rr and Rp have the same goal effect. Then we may say any $Rp \in \Theta$ is a corresponding Primitive Rule

of any Rr $\in\Omega$; and any Rr $\in\Omega$ is a corresponding Recursive Rule of any Rr $\in\Omega$.

C. Five Planning Control Strategies for Vertical Decomposition

Next, we present a set of control rules, which are necessary and sufficient for guiding a planning process handling vertical decomposition via combining Recursive and Primitive rules.

Criterion 1: For a selected goal, a Recursive Rule takes precedence over corresponding Primitive Rule.

This embodies the assumption that the complexity of applying a primitive rule to a certain case, generally increases more than linearly with the size of the case. Based on this assumption, it's generally more efficient to decompose a problem into parts when possible.

Criterion 2: Meeting the essential preconditions of a Primitive Rule is the stop condition for halting application of the corresponding Recursive Rule.

Once the planning process has actually applied a primitive rule to achieve a particular goal, there is no need for ongoing recursive decomposition related to this goal.

Criterion 3: Any Propositional Rule will inherit the cost evaluation function from its direct forward step in the planning network. For a Recursive Rule, the cost evaluation function will be the sum of all the inherited cost evaluation functions of its preconditions.

A propositional rule is assumed to have no direct cost (reflecting the assumption that the computational cost involved in executing the rule can be ignored). However, for the purpose of selecting proper values for the grounding of variables in a propositional rule, cost functions are still required as heuristics. Since its direct forward step gives it its goal, such a rule can usually inherit the cost evaluation function from its forward step. However, this criterion is not the only criterion for grounding Recursive Rules.

Criterion 4: The numeric values selected to ground a Recursive Rule should always satisfy at least half of the preconditions of this rule, unless it fails to select values by itself, in which case it requires application of Criterion 5.

Given a Recursive Rule R_r and its precondition set $P = \{p_1,p_2,p_3,...p_n\}$, the set of values V the planner selects to ground R_r should be able to satisfy at least half of the preconditions $\Omega = \{p_i,p_j,...p_m\}$, where $i,j...m\in\{1,2...n\}$, $i < j <...< m$, $\Omega \subseteq P$, $m \geq n*0.5$. This criterion is based on the philosophy that decomposing numeric goals should aim to reduce the problem rather than create additional problems. The value 0.5 used here could perhaps be tuned adaptively, but in practice we have found 0.5 to be adequate so far.

Criterion 5: When direct application of Criterion 4 fails to find adequate variable grounding values for a given Recursive Rule, an alternative is for the rule to borrow preconditions from one of its corresponding Primitive Rules. This will create the possibility to apply this Primitive Rule in the next step.

In Criterion 2, we explained that a Recursive Rule only breaks a bigger problem into instances of the same problem at a smaller scale, while its corresponding Primitive Rule is the rule that actually solves the problem. This means that the preconditions defined in a Primitive Rule are more essential for solving the problem. Therefore when a recursive rule fails to select proper variables through Criterion 4, logically there must be some unsatisfied preconditions in its corresponding Primitive Rule causing the failure, implying that borrowing some of the preconditions from its corresponding Primitive Rule may help to select suboptimal values. When there are multiple preconditions in this Primitive Rule, the preconditions which have no rules to achieve should be borrowed preferentially. It is impossible to borrow all of the preconditions, because if all the preconditions can be satisfied then the planner should have found a set of optimal values to ground this Recursive Rule according to Criterion 3 and 4.

III. DOMAIN DEFINITION

Now we introduce the MineCraft construction domain, in which we have tested our planning approach. The key distinguishing aspect of this domain is blocks-building. In addition to moving around, the agent is able to use blocks to build stairs, bridges or similar functional structures to construct a path to the target.

Our approach to the Minecraft domain centers on location variables, considered as a 3D vectors (x, y, z) , to be grounded during planning, defining positions for blocks as well as agents. An agent should be capable of determining the starting position to begin block construction, as well as where to build the next block and why, with an understanding of the spatial relationships between blocks, and between itself and blocks.

We construct our domain problem with the following low-level rules. Note that we define the domain in common STRIPS-like format. However, similar modeling can also be done with any proper version of PDDL [13] or SAT [14], or other comparable frameworks.

TABLE I. DOMAIN DEFINITION

Domain objects: target T; Domain agents: robot
Domain variable range: each location is a 3D vector v(x,y,z), where x, y, z are int and x, y, z $\in$ [0,100] .
Domain goal: distance(robot, T) < access_distance
Domain rule 1: **move_to_rule** Description: move to a target to get close to it Action: move_to (actor, target) cost = distance (actor, target) actor $\in$ {agents}, target $\in$ {location, object} Precondition: exist_path(actor, target) = true Effect: distance(actor, target) < access_distance
Domain rule 2: **path_transmission_rule** Description: if there exists a path from x to y, and y to z, then there exists a path from x to z. Note that this is a propositional rule. Action: do_nothing , cost = 0

Precondition: path_exists(x, y) = true path_exists(y, z) = true Effect: path_exists(x, z) = true x, y, z ∈{location, object}
Domain rule 3: **adjacent_access_rule**[1] Description: if x is adjacent to y, and the agent can stand on both x and y, then there exists a path from x to y. Action: do_nothing, cost = 0 Precondition: adjacent(x, y) = true can_be_stood_on(x) = true can_be_stood_on (y) = true Effect: path_exists(x, y) = true x, y ∈{location, object}
Domain rule 4: **build_block_rule**[2] Description: if a block is built immediately under a certain location, then the state "can_be_stood_on" for this location will become true. Action: build_block_at (actor, x), cost = 10.0 Precondition: is_under(x, y) = true is_solid(x) = false is_solid(y) = false x, y ∈{location} Effect: can_be_stood_on(y) = true

It's easy to see that the **path_transmission_rule** is a recursive rule, which breaks the goal path_exists(x, z) into subgoals path_exists(x, y) and path_exists(y, z). And **path_transmission_rule** and **adjacent_access_rule** are q corresponding pair -- both solve the problem path_exists.

Given these rules, the agent knows, for example, that building a block in a certain location means it will be able to stand on it, and that when two adjacent locations can both be stood on, the agent can move from one to the other. Hence the structure it builds will eventually create a path to the unreachable targets, no matter how it will look.

IV. Planning-time Sensing

For a large 3D MineCraft world (at least 100 units per dimension), it is not practical to store all the spatial states for each location / block. Therefore in our system, there is a set of planning-time sensors that can return the spatial states whenever a planning step requires this information, e.g., is_under (x, y), is_solid(x), distance(x, y), path_exists(x, y). This assists with determination of the reason why a path_exists(x, y) state sensor returns false - which part of the path is missing and how to build blocks to fill it up.

[1] Note: the given form of this rule is simplified. In a more general MineCraft world, even when two blocks are adjacent, there may be situations in which the path from x to y will be blocked by other adjacent blocks, or blocks the agent cannot stand on such as water. Our implementation covers these cases as well.

[2] The rule we have implemented in our software requires the actor to be close enough to the location.

For example, in Figure 2, the agent's target is in a tree which is not directly reachable, though it is possible to get near the target by climbing a different tree. A human agent would instantly identify the absence of a block between the two trees as the reason for the pathfinder failing to find a path from the agent to the target and that the most efficient solution would be to build a connecting block at point C. For an AI to recognize this fact however requires more techniques, which will be discussed below.

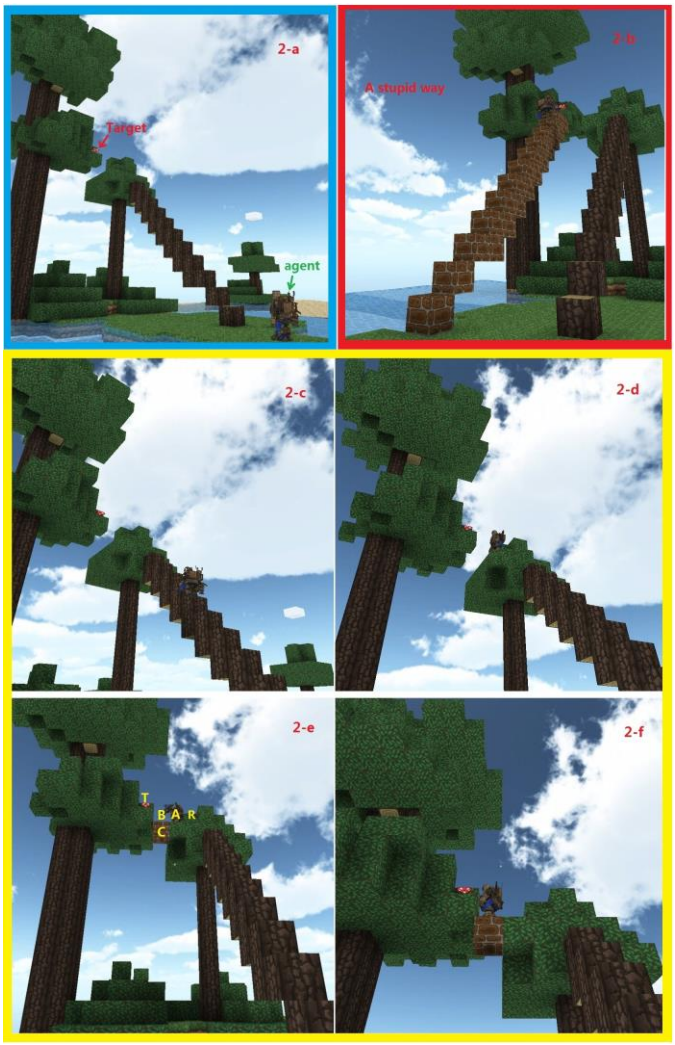

Fig. 2. Solving an example problem in the MineCraft construction domain. 2-a shows the initial states; 2-b is a stupid way to solve the problem as some scripted game AI may do; 2-c,d,e,f shows our agent reaching the target by climbing up the stairs to get on the right tree, and then only needing to build one block to connect the left tree and right tree so it can access the target.

V. Experiment of Automated Construction Agents in MineCraft Worlds

A. Planning Logic

We now describe how these ideas have been used to unify navigation and planning in the Minecraft construction domain. Although we use a backward planning network to illustrate the planning steps, the application of our new strategies is not limited to backward planning. For very large Minecraft worlds, we expect that a combination of backward and forward planning will be preferable.

We will use Figure 2-a as a running example. The target, a red mushroom (T), is on the left tree, the agent is on the ground, and there is an existing staircase leading to the top of the tree on the right. The purpose of this example map is to show the significant difference between general planning and scripted game AI. A game AI script for building stairs may result in building a new stairs all over again just as Figure 2-b shows, because it doesn't understand the reason for every block it builds - it just builds a structure in the shape of stairs. As a general planner, it will know the reason for building a block at a specific position is to allow the agent to access the position above its adjacent position. In this example map, our agent will climb up the existing stairs to get onto the right tree, stand at position A as shown in Figure 2-e, build a block at position C because it needs to be able to move to B, so it can finally access the target T adjacent to B. This means the agent only needs to build one block at C, instead of building a whole new staircase.

TABLE II. MAIN STEPS OF SOLVING THE EXAMPLE PLANNING PROBLEM

Step 1: Select rule: **move_to_rule** grounded variables: agent = robot, target = T preconditions: path_exists(robot, T) = true Unsatisfied
Step 2: Select rule: **path_transmission_rule** grounded variables: x = robot, y = A, z = T preconditions: path_exists(robot, A) = true Satisfied path_exists(A, T) = true Unsatisfied
Step 3: Select rule: **path_transmission_rule** grounded variables: x = A, y = B, z = T preconditions: path_exists(A, B) = true Unsatisfied path_exists(B, T) = true Unsatisfied
Step 4: Select rule: **adjacent_access_rule** grounded variables: x = A, y = B preconditions: adjacent(A, B) = true Satisfied can_be_stood_on(A) = true Satisfied can_be_stood_on (B) = true Unsatisfied
Step 5: Select rule: **build_block_rule** grounded variables: actor = agent, x = C, y = B precondition: is_under(C, B) = true Satisfied is_solid(C) = false Satisfied is_solid(B) = false Satisfied

The planning steps are given in Table 2. Next, we explain how the steps given in the table apply the 5 strategies to ground variables. Recall that **path_transmission_rule** and **adjacent_access_rule** form a corresponding recursive rule and primitive rule pair (see Definition 5). According to Strategy 1 and 2, Step 2 and 3 select **path_transmission_rule** but not **adjacent_access_rule**, because the scale of the problem is not small enough to apply **adjacent_access_rule** yet. If **adjacent_access_rule** is selected for application, its precondition: adjacent(x, y) = true would not be satisfied both in Step 2 and 3, and no rules can achieve this precondition, so **path_transmission_rule** is selected to break the goals into smaller scales. Strategy 3 allows in Step 2 **path_transmission_rule -** a propositional rule which has no cost evaluation itself – can inherent the cost evaluation from its direct backward rule Step 1 **move_to_rule**, which makes **path_transmission_rule** able to choose a variable with the lowest cost for "move". The candidates for a numeric variable are generated by a genetic algorithm.

In Step 2, for the selection of a middle value y to ground the **path_transmission_rule,** how can the agent find position A to ground y? If we only apply Strategy 3, according to its inherited cost evaluation function from its direct forward step: cost = distance(robot,y) + distance (y,T), then any position y on the straight line from the robot to T will give a minimum cost. But such a y will not satisfy either precondition 1 or precondition 2 in Step 2, which will make the problem more complex rather than simplify it. So according to Strategy 4, the planner will select y from positions that at least satisfy one of the preconditions, and pick the one having the lowest cost. This is why it can find position A (see Figure 2-e) as the optimal value to ground variable y.

Step 3 will try to achieve the unsatisfied precondition of Step 2: **path_exists(A,T) = true**. If we try to apply **adjacent_access_rule** for step 3, one of its preconditions **adjacent(A,T) = true** is not satisfied and there is no rule that can make **adjacent(A,T)** true. Therefore it is not applicable in step 3, so the only rule left is **path_transmission_rule**. That is why step 3 chooses this rule. However, when the planner tries to ground **path_transmission_rule** in step 3, finding a value to ground variable B, it finds no other positions that will satisfy either precondition 1 or 2 with acceptable cost. (E.g. position R to the right of A (see Figure 2-e) will satisfy precondition1 path_exists(A, R) = true, but it will make the cost to achieve precondition 2 **path_exists(R,T) = true** even higher than its goal **path_exists(A,T) = true**). Thus the planner fails in selecting an optimal value for this rule according to Strategy 3 and 4, which suggests that after decomposition of the original numeric problem **path_exists(robot, T) = true**, it finds that the subgoal **path_exists(A, T) = true** is the critical subproblem which causes the pathfinding to fail. Therefore, according to Strategy 5, in step 3 the **path_transmission_rule** will borrow the preconditions from its corresponding Primitive Rule: **adjacent_access_rule**. As mentioned above, because the precondition of the "adjacent" state has no rules to achieve it, while the "can_be_stood_on" preconditions have the rule **build_block_rule** to achieve them, the "adjacent" precondition is much more essential, which suggests it is the one that should be borrowed. That is why in step 3, the planner selects position B to ground this rule (B is adjacent to A with a lower cost). Another purpose for the borrowing of preconditions from a Primitive Rule is to create the option for the next step to apply this Primitive Rule. As discussed above, **adjacent_access_rule** is not applicable due to its impossible precondition **adjacent(A,T) = true**, but because adjacent(A,B) = true is now satisfied, in step 4 the planner can choose **adjacent_access_rule**, and then it will find that the precondition can_be_stood_on(B) is false. Finally in step 5 it will apply the **build_block_rule** to build a block below B.

B. Planning Network Implementation

The implementation of our planning algorithm is based on backward chaining network, which is composed of StateNodes and RuleNodes alternately. The planning process is to construct such a network. Each planning step contains the following procedures:

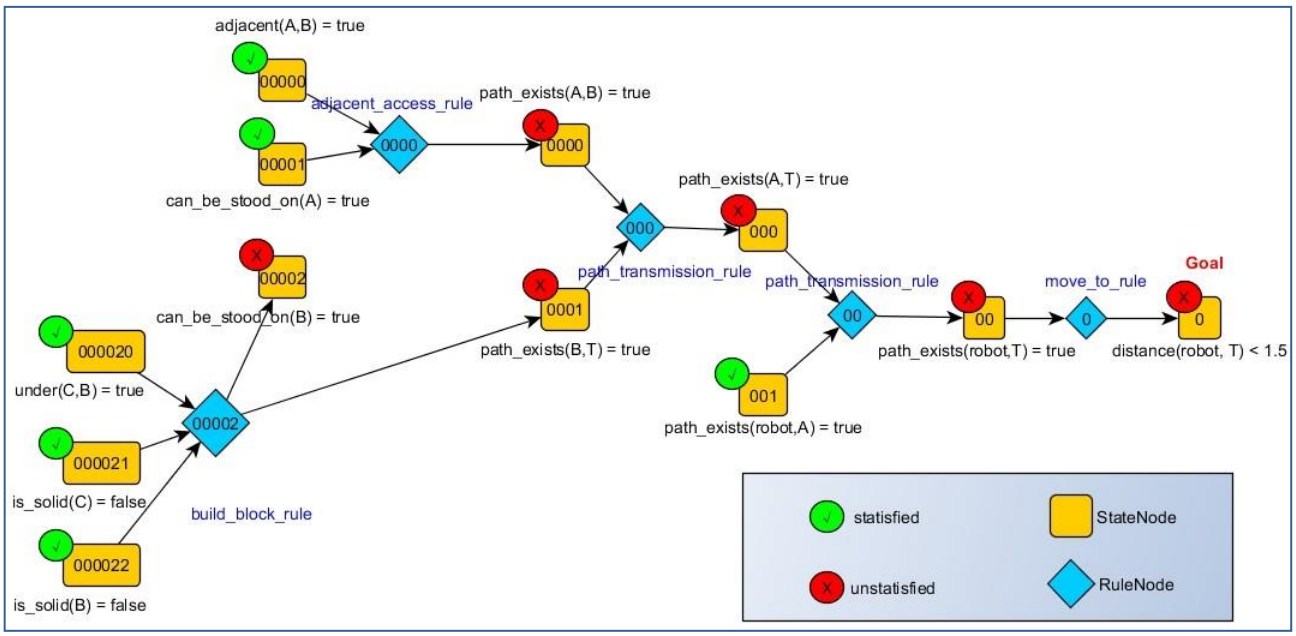

Fig. 3. The backward planning network for the example problem. Because this is backward chaining, please read from right to left. The number in each StateNode and RuleNode stands for its depth in the planning network. If there are bigger gaps between A and T, the planner will recursively apply path_transmission_rule and adjacent_access_rule iteratively to build more blocks.

- Select a StateNode to satisfy, from all the unsatisfied StateNodes (The goal is the root StateNode);
- Select which rule to apply in order to satisfy the selected StateNode;
- Select proper values to ground all the variables in the selected rule;
- Construct a RuleNode for this grounded rule;
- Generate all the precondition nodes for this grounded RuleNode; every precondition is represented as a new grounded StateNode; Check these precondition StateNodes satisfied or unsatisfied in the current world state;
- Generate all the effect nodes for this grounded RuleNode; one of the effect should be the selected StatedNode, so connect this RuleNode to the selected StatedNode; if there are other effect for this rule, check if any effect happen to satisfy other unsatisfied StateNodes; if there are, connect this RuleNode to the corresponding existing StateNodes; create new grounded StateNodes for the rest of the effect.

All the selection of StateNodes, rules and values to ground are decided by the 5 planning control strategies discussed in section C in II. Figure 3 illustrates the planning network of the planning logic discussed in session A in V.

VI. Other Experiments

A. *Planning Problem in Fig. 1-a*

Fig. 4 shows our agent found the most efficient solution that climb up the stairs near the target and then build two blocks to fill the gap between the target and the top of the stairs.

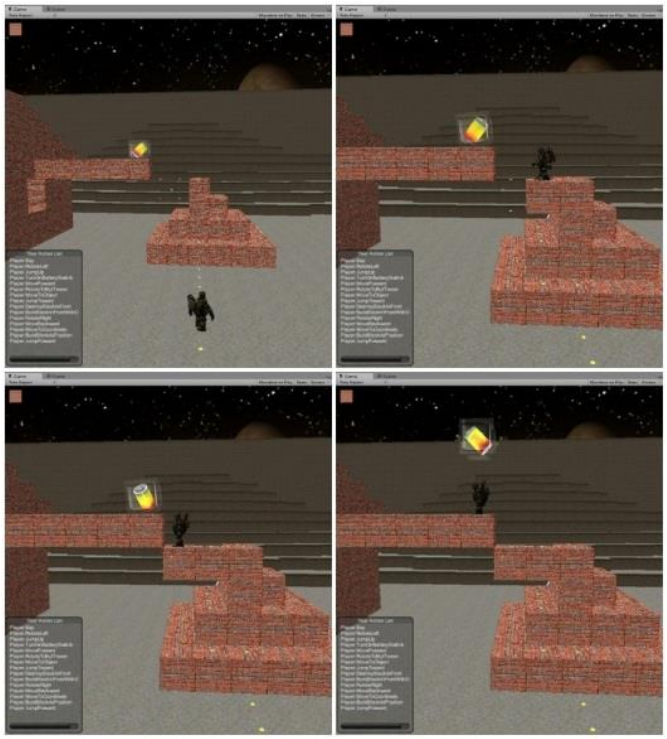

Fig. 4. The screenshots for solving the problem in Fig. 1-a

B. *Extending the Block-building Problem with Einstein's Logic Puzzle Using the Same Planning Network*

In order to test the generality of this planner, this experiment requires the agent to solve a simplified Einstein logic puzzle - "who keeps the fish?" (The original version is too large to be described here) and then build blocks to construct a path to access the person who keeps the fish. The problem is defined as bellow:

- There are 3 men on the map: An American, a German and a British person (as A, G, B on the map).

- There are 3 pets: dog, cat and fish, and 3 drinks: water, milk and tea.
- The goal is to be close to the man who keeps the fish as pet.

The known facts and rules are:

- The British person doesn't drink water.
- The German doesn't drink milk.
- The British person doesn't keep dogs.
- The person who drinks tea keeps cats.
- The person who drinks water keeps fish.

The answer is that the German keeps the fish. First the planner needs to solve the puzzle to figure out who keeps the fish, and then move to this person. Because the German is in an unreachable place, after the planner figures out that "G" keeps the fish, it needs to build blocks to construct a path to "G". Our planner solved this problem successfully with 36 planning steps in total (The planning output logs are in the appendix).

VII. RESULTS

A. *Results of MineCraft Construction Agent Experiment*

Fig 5 shows the average results of running our planner to solve the problem in Fig 6-1, at different scales (where "scale" means the number of the blocks that needs to be built). The number of steps and the CPU time both increase linearly as the problem scale increases, so the time complexity O(n) = n. The number of steps and the CPU time for solving the same problems in a small map (20 x 20 x20) are the same with a bigger map (100 x 100 x 100), so the map size does not affect the problem scale in our planner, because the cost evaluation heuristics will make the agent stay close to the context.

Problem scale	Planning steps	CPU time
1	9	103ms
2	13	133ms
3	17	184ms
4	21	221ms
5	25	262ms
7	33	323ms
9	41	419ms
11	49	555ms
13	55	655ms

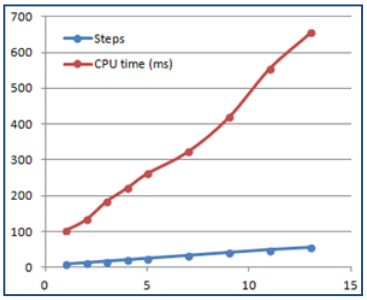

Fig. 5. The results of steps and CPU time

B. *Comparison with Other HeuristicPlanning Approaches*

Fig 6-2 shows the result of our approach. By vertically decomposing the goal recursively, the agent figures out that the critical problem in this experiment is the gap between the target and the top of the stairs. Therefore it discovers the solution to climb up the stairs and build two blocks to fill the gap. Fig 6-3 shows the result of other planners with heuristics, which work with various variations on A*. These kinds of planners usually only consider the planning distance between the target state and current state in heuristic evaluation, which will result in making a whole new staircase from the original location of the agent to the target as Fig 6-3 shows. The planning result solution of our approach is superior to other backward heuristic planners that lack the ability to vertically decompose numeric goals.

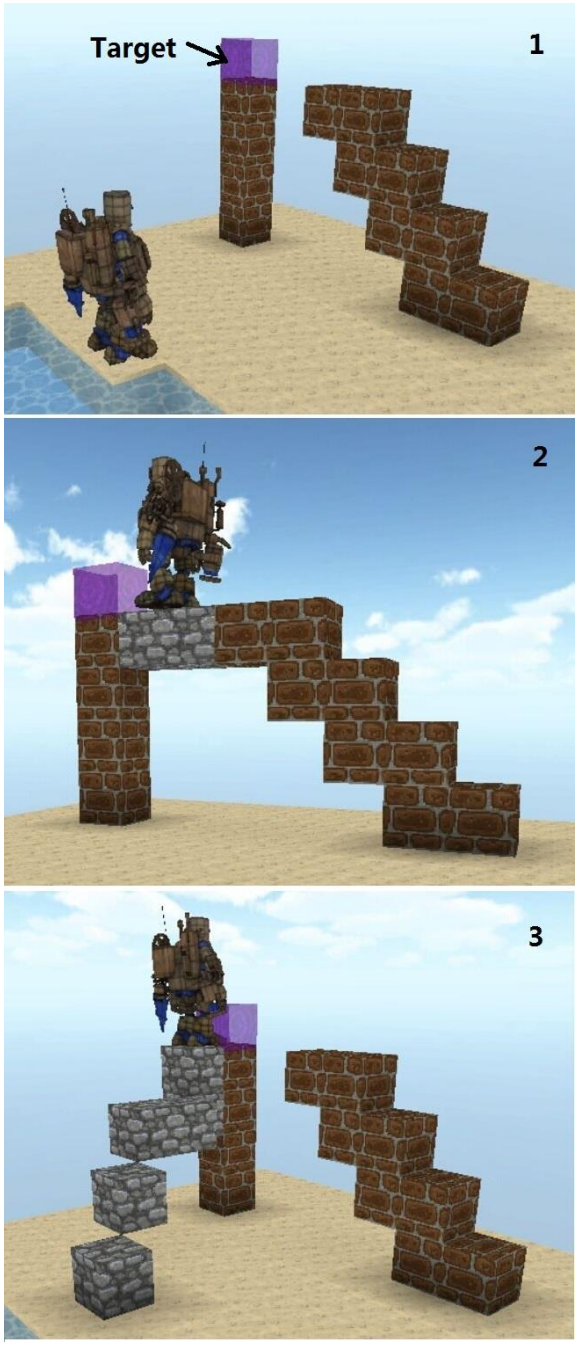

Fig. 6. The screenshots for the Experiment

C. *Comparison with FF and SAT Approaches*

Any variation of FF planners [15] or SAT [14] approaches may be able to come up with the same solution as our approach, but at a much higher CPU cost. These types of planners consider all possible combinations of state transmission at each step by constructing the full planning graph from the current layer to the next layer (FF planners), or encoding all possible formulae from the current time step to the next time step (SAT planners). However this will also cause a combina-

torial explosion in a 3D Minecraft world. Considering the same planning problem in Fig 5:

For each step, there are 16 action options:

move_to any of 8 adjacent locations

build_block_at any of 8 adjacent locations

For problem scale = n, the number of all probabilistic combinations of action sequences could be estimated as:

$N_a = 16^n$

Table III shows the CPU cost of just looping through N_a number without doing anything, for the problem scales 7, 9, 10 and 11 in Fig 5:

TABLE III. RESULTS COMPARISON BETWEEN OUR APPROACH AND THE OTHER APPROACHES THAT CAN GENERATE THE SAME EFFICIENT SOLUTIONS

Problem scale	**N_a**	**CPU cost for other approach**	**CPU cost for our approach**
7	268435456	739ms	0.323s
9	68719476736	3m15s	0.419s
10	1099511627776	51m33s	0.459s

Note: Problems of a scale lower than 7 are too small and don't show much difference in the CPU cost. The CPU cost for problems above scale 10 will exceed several hours, for which it doesn't make a lot of sense to run tests. So we only list the results for the problem scales 7, 9 and 10.

Table III shows our approach is significantly faster than other approaches that are able to generate the same efficient solutions for the MineCraft construction domain problem. Although applying heuristics to FF planners or SAT planners to select an action for each step will result in a much lower CPU cost, the resulting solutions will mostly turn out to be the kind of inefficient solutions shown in Fig 6-3, meaning that in this case the CPU cost should not be used as a comparison with our approach.

VIII. CONCLUSION

We have modeled the navigation problem within a general planning architecture, via the deployment of 5 strategies for recursive vertically decomposition of numeric goals. These strategies are based on exploiting the intrinsic relationship between Recursive Rules and Primitive Rules. They allow recursive goal decomposition until the critical subgoals which are the main cause of the initial difficulty in goal achievement have been found.

We have shown that this approach enables agents to automatically construct paths in order to efficiently reach initially unreachable targets in a 3D MineCraft game world (a demonstration video url can be found in the appendix). This both validates our overall approach to fusing navigation and planning, and provides a useful tool for the control of AI agents in MineCraft-like worlds. The experiment in 6.2 involving solving a simplified Einstein logic puzzle as well as building blocks in the same planning network shows the generality of our planner.

Future work will involve application of the same sort of process in broader domains, for instance robot navigation, and game worlds involving more complex primitive objects than blocks.

APPENDIX

- A demonstration video of our planner can be found here:

http://youtu.be/X5CFd8RPFJM

- The output log for solving the planning problem in section B in VI:

https://docs.google.com/document/d/1ZaCSbP0YYJqq6yrOvADA-2ghT2ONzlZEjaLcQu4EkqA/edit

REFERENCES

[1] Hoffmann, J.: Everything You Always Wanted to Know About Planning (But Were Afraid to Ask) (2011)

[2] Long, D. et al., 2000. The AIPS-98 Planning Competition. AI Magazine, 21(2), p.13.

[3] Fulgenzi, C. et al.: Probabilistic navigation in dynamic environment using Rapidly-exploring Random Trees and Gaussian processes. In IEEE/RSJ International Conference on Intelligent Robots and Systems, 2008. IROS 2008. IEEE/RSJ International Conference on Intelligent Robots and Systems, 2008. IROS 2008. pp. 1056–1062 (2008)

[4] Kleiner, A. et al.: Hierarchical visibility for guaranteed search in large-scale outdoor terrain. Autonomous Agents and Multi-Agent Systems, 26(1), pp.1–36.

[5] Zhang, S., Sridharan, M. & Washington, C., 2013. Active Visual Planning for Mobile Robot Teams Using Hierarchical POMDPs. IEEE Transactions on Robotics, 29(4), pp.975–985 (2013)

[6] Fikes, R. & Nilsson, N.: Strips: A new approach to the application of theorem proving to problem solving. Artificial Intelligence, 2(3-4), pp.189–208 (1971)

[7] Gupta, N. & Nau, D.S.: On the Complexity of Blocks-World Planning. Artificial Intelligence, 56, pp.223–254 (1992)

[8] Nau, D. et al.: SHOP: Simple Hierarchical Ordered Planner. In Proceedings of the 16th International Joint Conference on Artificial Intelligence - Volume 2. IJCAI'99. pp. 968–973 (1999)

[9] Q. Yang.: A Descomposition and Abstraction Based Approach. Intelligent Planning, Springer, Berlin, Heidelberg (1997)

[10] Hoffmann, J., Porteous, J. & Sebastia, L.: Ordered Landmarks in Planning. J. Artif. Int. Res., 22(1), pp.215–278 (2004)

[11] Sebastia, L., Onaindia, E. & Marzal, E.: Decomposition of planning problems. AI Commun., 19(1), pp.49–81 (2006)

[12] Yu, H. et al.: Planning with Recursive Subgoals. In M. G. Negoita, R. J. Howlett, & L. C. Jain, eds. Knowledge-Based Intelligent Information and Engineering Systems. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 17–27 (2004)

[13] Mcdermott, D. et al.: PDDL - The Planning Domain Definition Language (1998)

[14] Kautz, H., Selman, B. & Hoffmann, J.: SatPlan: Planning as Satisfiability. The 5th International Planning Competition (2006)

[15] J. Hoffmann, "FF: The Fast-Forward Planning System," AI Magazine, vol. 22, no. 3, p. 57, Sep. 2001.