13

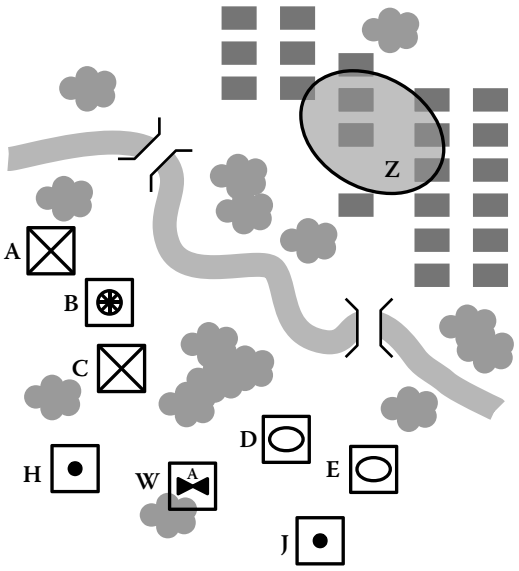
Hierarchical Plan-Space Planning for Multi-unit Combat Maneuvers

William van der Sterren

13.1 Introduction

In combat simulators and war games, coming up with a good plan is half the battle. Good plans make the AI a more convincing opponent and a more reliable assistant commander. Good plans are essential for clear and effective coordination between combat units toward a joint objective.

This chapter describes the design of an AI planner capable of producing plans that coordinate multiple units into a joint maneuver on the battlefield. First, it looks at how planning for multiple units is different from planning for a single unit. Then it introduces the basic ideas of hierarchical plan-space planning. These ideas are made more concrete for the case of combat maneuvers. The article wraps up with an evaluation of the design and ideas for further application of hierarchical plan-space planning.



"Our plan:
We'll clear objective Z, with A, B, C, D and E platoons forming up and launching a two pronged simultaneous attack. Afterwards, we'll regroup at objective Z.
B platoon will transport A and C to their form up areas. A and C platoons will attack across the northern bridge, D and E platoons will attack across the southern bridge.
Fire support is provided by batteries H and J and gunships W. Batteries H and J will fire smoke screens to cover the bridge crossings. W flight will be on call."

Figure 13.1

A multi-unit planning problem (left) and the result (right) as briefed to the player.

13.2 Planning for Multiple Units

Creating a plan for multiple units is different from planning for a single unit. Obviously, the plan needs to cater to all the units instead of a single unit, and will involve more actions. In many cases, these units will perform their actions concurrently.

But there is more to it: in most cases, these units will have to interact with each other to accomplish the goal. To coordinate this interaction, the plan needs to tell who needs to interact with whom, where, and at what time.

Another difference is in communication of the plan: the actions making up a single unit's plan typically require no additional explanation. However, when multiple units work together towards an objective, additional explanation is often expected (for example, as part of the briefing in Figure 13.1). How is the work split across subgroups? Who is assisting whom? What is each group's role? And for combat plans, what is the overall concept?

Given these differences, can we take a single-unit planner such as GOAP [Orkin 06] or an HTN planner [Ghallab et al. 04, Humphreys 13] and create plans for multiple units? For all practical purposes, we cannot. Both these kinds of planners construct their plan action for action, and traverse a search space consisting of world states (the state-space [StateSpaceSearch]). Our problem is the enormous state-space resulting from multiple units acting concurrently. For example, assume a single unit has four alternative actions to move about or manipulate its environment, and we are in need of a five-step plan. For this "single unit" case, the total state-space consists of $4^5 = 1024$ states, and can easily be searched. If we attempt to tackle a similar problem involving six units acting concurrently, the state-space size explodes to $(4^6)^5 \sim 1.15\ 10^{18}$ combinations. GOAP and, to a lesser extent, standard HTN planners struggle to search efficiently in such a large state-space.

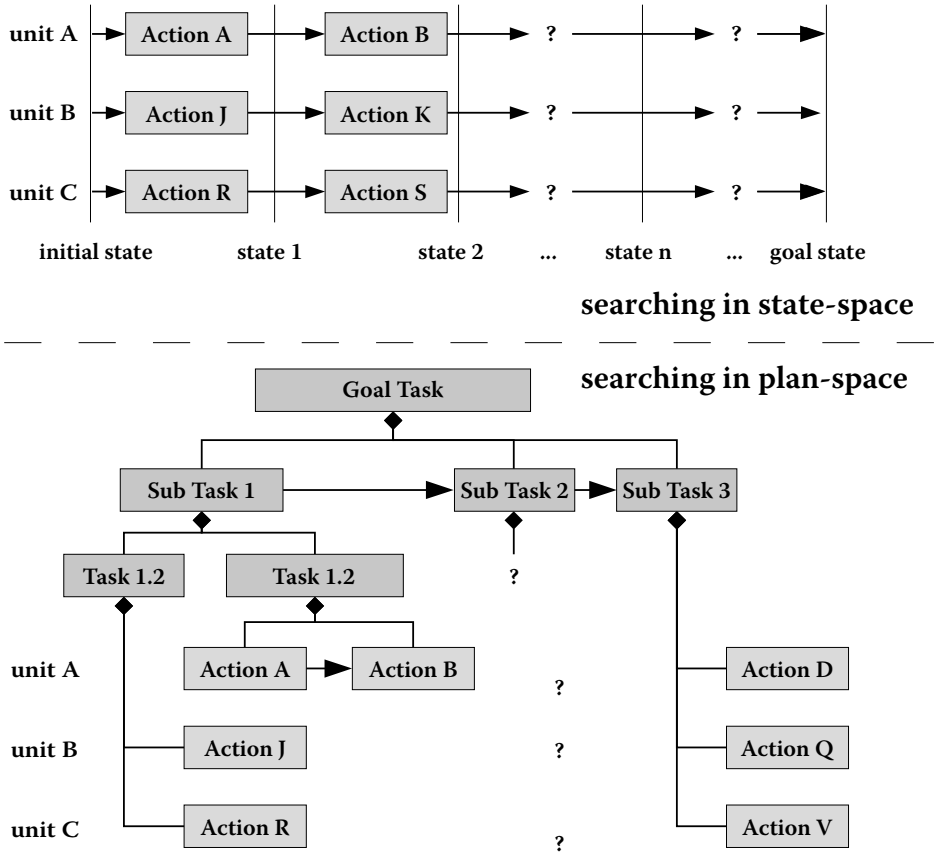


Figure 13.2

State-space search (top) compared with plan-space search (bottom).

Instead of searching in state-space, we can attempt to search in plan-space (see Figure 13.2). Plan-space represents all incomplete and complete plans. This may sound vague, but it actually is quite similar to how human project planners tackle planning problems. Project planners break down the overall problem into smaller tasks that together accomplish the goal. They then repeatedly break down these smaller tasks until the resulting activities are small enough to be accomplished by a single unit's action. See Figure 13.3 for an example of a fully detailed plan.

Working in plan-space offers three key advantages when tackling multiunit planning problems. First, we can make planning decisions at a higher level than individual actions by reasoning about tasks and subtasks. Second, we have the freedom to detail the plan in any order we like, which allows us to start focusing on the most critical tasks first. And, third, we can explicitly represent coordination (as tasks involving multiple units), and synchronization (as tasks not able to start before all actions of a preceding subtask have completed) in our plan. With these advantages, we are able to generate plans describing coordinated actions for multiple units even for a large search space.

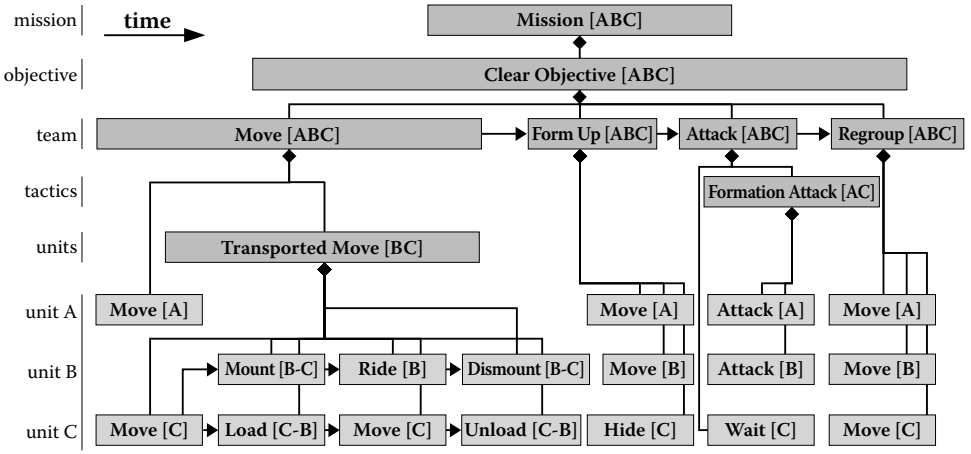


Figure 13.3

A complete plan with higher level tasks (top) and resulting unit actions (bottom).

This article continues by detailing this approach of hierarchical plan-space planning for a combat maneuver problem as illustrated in Figure 13.1.

13.3 Hierarchical Planning in Plan-Space: The Ingredients

We need four ingredients to implement hierarchical planning in plan-space: a planner main loop, the tasks and actions to represent the plan, a set of planner methods which can refine a partial plan by detailing one task in that plan, and finally the plan-space that holds and ranks all partial plans. We will look into these ingredients in this order.

13.4 Planner Main Loop: An A* Search through Plan-Space

The planner main loop executes the search through plan-space. The search starts with a single plan consisting of a single top-level task (the "mission"). Next, the main loop repeatedly picks the most promising plan from the open plans in plan-space and attempts to expand that plan by refining the plan's tasks. The main loop exits successfully when a plan is found that is complete. The main loop exits with a failure when there is no open plan left to be expanded. Figure 13.4 shows the pseudocode for the planner main loop.

The main loop expands a selected plan as follows. It first picks a single task requiring refinement from the plan. It then selects from the catalog of planner methods the methods that can refine this selected task. Each of these methods is applied separately, resulting in zero or more alternative expanded plans (we will discuss this in more detail later). Every expanded alternative plan is assigned a cost and added to the open list.

The main loop is quite generic and similar to an A* path search. Here, we are expanding plans into one or more neighboring plans which are closer to a fully detailed plan, instead of expanding paths into one or more neighboring locations which are closer to the destination. We are expanding plans in a best-first approach, something that is explained in more detail when looking into the plan-space.

```
loop
  current = get most promising plan from open list
  break if current.complete? or current.null?
  add current to closed list
  pick t = current.task_to_detail
  for every method m that applies to task t
     alternatives = m.generate(current, t)
     for every a in alternatives
          plan = clone current
          // refine using method m and alternative a
          m.plan(plan, t, a)
          compute plan's cost
          add plan to open list
```

Figure 13.4

Pseudocode for the planner main loop.

13.5 A Plan of Tasks

A plan consists of interdependent tasks. A task represents an activity for one or more units and consumes time. For our combat maneuver domain, we need tasks to represent basic unit actions, and we need tasks to represent higher level activity. Table 13.1 lists examples of both types of tasks, with unit level tasks in the bottom row. The scope reflects the various levels at which decisions are made and problems are broken down in the military: mission, objective, team, tactics, units, unit.

The basic unit tasks simply follow from the activity that a unit—such as an infantry squad, a tank platoon, or a gunship section—is capable of. We call these tasks "primitive" since we cannot decompose them. The higher level tasks are intended to help us make higher level planning decisions and break down the plan (as shown in Figure 13.3). In general, these tasks are about assigning resources to subgoals and coordinating subtasks. Concrete examples for our combat maneuver domain include a complete team moving to a form-up position, preparatory strikes by artillery and aircraft, or a para drop. These tasks are called "compound" since we can break them down into smaller tasks.

Tasks have a start time and duration. A task's duration is computed as the activity duration for primitive tasks, as the latest subtask's end-time minus earliest subtask's start-time for tasks already refined into subtasks, and as an estimated duration for a compound tasks not yet refined. We'll look into these estimates later.

In the plan, the tasks are organized as a graph. Every task has a parent except for the root task. Compound tasks have children (subtasks implementing their parent). Tasks

Table 13.1 Examples of tasks for combat maneuver domain, arranged by scope

Scope	Task examples
Mission	Mission
Objective	Clear, occupy, defend
Team	Move, form up, attack, air land, defend, counter-attack, para drop
Tactic	Formation ground attack, planned fire support, smoke screen
Units	Transported move, defend sector
Unit	Defend, guard, attack, hide, move, wait, air ingress, air egress, mount, dismount, load, unload, ride, para jump, fire artillery mission, close air support

```
# A LoadTask expects:
# - a start_state, indicating the unit's initial state
# - a target state, indicating the unit's loaded state
# - the passenger
class LoadTask < Task
  is_primitive
  has_scope :unit
  has_input :start_state,   :type => :unit
  has_input :target_state,  :type => :unit
  has_input :passenger,     :type => :unit

  def compute_expected_costs(context)
    15.0
  end
end
```

```
# An AttackAfterFormUpTeamTask expects:
# - an objective
# - a start state (as unit states preceding the form-up & attack)
# - an avenue of approach to use for this attack
# It outputs:
# - the objective area indicating what terrain to attack & from where
# - the assembly area for pre-attack form-up
# - per unit assembly positions in the form-up area
# - an end-state, as unit states after the attack in the objective area
class AttackAfterFormUpTeamTask < Task
  has_scope  :team
  has_input  :start_state,        :type => :units
  has_input  :objective,          :type => :objective
  has_input  :avenue_of_approach, :type => :avenue_of_approach
  has_output :objective_area,     :type => :area
  has_output :assembly_area,      :type => :area
  has_output :assembled_state,    :type => :units
  has_output :end_state,          :type => :units

  def compute_expected_costs(context)
    ...
  end
end
```

Figure 13.5

Two examples of tasks, with inputs and outputs.

may have preceding tasks which require completion before the task can start. For example, a team formation attack won't be able to start until all the form-up tasks of all involved units have been completed. These precedence relations between two tasks also imply all of the first task's subtasks precede the second task. Tasks may have successor tasks in the same way.

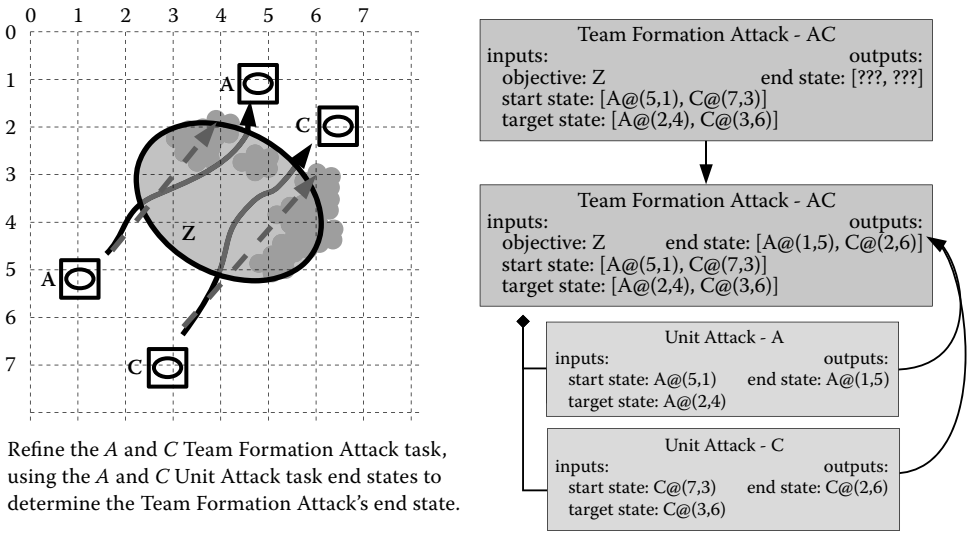
Tasks are parameterized with inputs and may provide outputs. In our combat maneuver domain all tasks take the units involved as input, typically with the units in the planned state (position, ammo level) at the start of the task. Primitive tasks deal with one single unit; compound tasks typically take an array of units. Many tasks take additional inputs—for example, to denote cooperating units, assigned targets or zones, or target states (in unit positions at the end of the task).

Figure 13.5 shows an example of two kinds of tasks, each taking inputs. The *LoadTask* represents the loading activity by a transporter unit such as an APC platoon. The *LoadTask* takes three inputs. The start-state input identifies the transporter unit and its initial state consisting of its position, and identifiers for any passenger units already being mounted. The target-state input is similar to the start-state but with the indicated passenger unit mounted. The passenger input identifies the passenger unit.

The *AttackAfterFormUpTeamTask* represents a multi-unit ground attack from a form-up position. It takes three inputs. The start-state input takes an array of units that will execute the attack. The objective input and avenue-of-approach inputs provide additional guidance from "higher up" on how to refine this team level task.

The *AttackAfterFormUpTeamTask* also provides outputs, as do many other tasks. The purpose of an output is to provide values to other tasks' inputs, enabling them to work from a resulting unit state, or from a tactical decision such as an avenue of approach.

A task input need not be set on task creation. It may be left open until the task is being refined. Or it can be connected to the input or output of another task and receive a value when the other side of the connection is set. Figure 13.6 illustrates this.



Refine the *A* and *C* Team Formation Attack task, using the *A* and *C* Unit Attack task end states to determine the Team Formation Attack's end state.

Figure 13.6

A parent's task output being determined by child tasks.

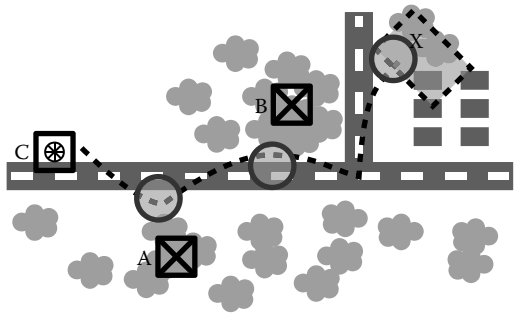
In Figure 13.6, a *TeamFormationAttack* task has been created involving tank platoons A and C. The task is given a start-state consisting of the A and C units with their start positions. The task's target-state indicates the tank platoons should move into positions at the far end of objective Z. The *TeamFormationAttack's* end-state output is left open intentionally, leaving detailed positioning of the tank platoons to more specialized subtasks. When the planner refines the *TeamFormationAttack*—for example, by adding two *UnitAttack* tasks, it connects the *UnitAttack's* end-state outputs to the *TeamFormationAttack* end-state output. When the planner refines the *UnitAttacks*, it will set the end-states with values representing positions close to the desired target-state but outside the woods. As soon as these *UnitAttack's* end-states are set, they will propagate to the *TeamFormationAttack's* end-state (and propagate further, if other inputs have been connected to that end-state).

Task outputs thus serve to pass on planning decisions and states along and up the chain of tasks. Connections between outputs and inputs determine how tasks share values. Connections can link inputs and outputs as a whole, but also (for arrays) on a per-element basis. In Figure 13.6, each of the *UnitAttack* tasks sets an element in the *TeamFormationAttack's* end-state.

We call *task* inputs that have all their values set "grounded" tasks. "Ungrounded" tasks lack one or more values in their inputs. We will revisit this distinction when discussing the order in which tasks are being refined.

13.6 Planner Methods

When the planner wants to refine a task in a partial plan, it selects the planner methods that apply to this task. It then applies each of these planner methods separately on a clone of the partial plan, and has the planner method generating alternative and more refined versions of the partial plan.



The task is for truck platoon C to pick up infantry squads A and B, and transport them to form-up area X. There A and B are to dismount, and all are to move to their target positions.

This requires decisions on:
– where to pick up A, and where B
– whether to pick up A before B
– where to drop off A and B
– exact final positions for A, B and C

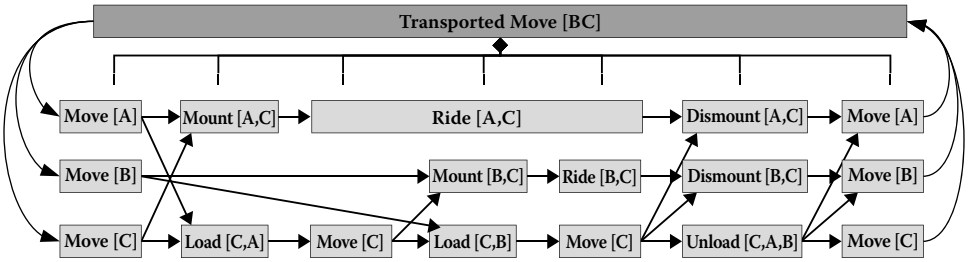


Figure 13.7

Decisions and subtasks when refining a TransportedMove task.

The role of the planner methods (we'll refer to them simply as "methods" from now on) is to refine a specific task in the plan. Methods themselves indicate which task or tasks they are able to refine. If the task to be refined is a primitive task, the method should compute and set this task's outputs. Figure 13.6 shows how the tank platoon's *UnitAttack* is given an output (a destination position outside the woods at the far end of the objective) that matches the tank unit's movement capabilities.

If the task to be refined is a compound task, then the method's responsibility is to decide how to implement that task, create the necessary subtasks, and connect the inputs and outputs of the task and subtasks. The method should ensure the outputs of the task being refined are set or have connections into them. Figure 13.7 illustrates an example of the decisions to be made, and the tasks, relations, and input/output connections to be created by a *TransportedMove* method in order to refine a nontrivial *TransportedMove* task.

To break down a *TransportedMove* task for a truck platoon C and two infantry squads A and B (in Figure 13.7), the *TransportedMove* method first makes a number of decisions. The method selects positions for the truck platoon to pick up squads A and B. Such a pick-up position needs to be accessible for the truck platoon and preferably close to the infantry squad. If the infantry squad had been in the open, the truck platoon might have picked it up at the squads' initial position. In this example, however, the infantry is positioned in the woods and needs to move into the open in order to be picked up. The drop-off point near form-up area X is picked in a similar way. The third decision is about picking up A before B or the other way around. Based on a few path-finding queries, picking up A before B is chosen. The final decision involves picking final positions for A, B, and C if not already given.

Table 13.2 Examples of planner methods and their responsibilities, arranged by scope

Scope	Planner method responsibility
Mission	Arrange objectives, allocate units to objectives
Objective	Define team activities, assign combat units and support units to teams
Team	Execute tasks as a team, distributing the work according to roles
Tactic	Synchronize tactical activity between multiple units
Units	Arrange cooperation between complementary units
Unit	Define end-state

Based on these decisions, the *TransportedMove* method can create the tasks for the two infantry squads and truck platoon. By making one task a predecessor of the other task, the method creates a sequence of tasks for each of the units. In addition, it synchronizes the load/mount actions and the unload/dismount actions by also making specific actions from other units a predecessor of these tasks. For example, the action for C to load A cannot start before both C and A have completed their moves to the pick-up position. Similarly, the infantry squads cannot disembark before the truck platoon has arrived at the drop-off position.

Since the *TransportedMove* method in this example already makes most of the decisions for all units and tasks involved, it can simply set output values and input values for most of the tasks.

To fully cover our combat maneuver domain, we need methods to set end-states for each of the primitive unit tasks, and we need methods to break down each of the compound tasks. For breaking down compound tasks into smaller tasks, we mirror the hierarchy chosen for tasks, from mission level methods down to unit level methods. As a rule of thumb, methods break down tasks into tasks of the next level, sometimes one level more. At each level, the methods have slightly different responsibility, as is illustrated in Table 13.2.

For most tasks, there will be a single corresponding method that is able to break the task down. For a few tasks it makes sense to have multiple methods for refining the task, each specialized in one type of tactical approach. To defend an objective, one method would create subtasks that make the available platoons each occupy a static position around the objective. Another method could create subtasks that have infantry platoons defending from static positions and keeping armor platoons in the rear for a counter-attack.

One benefit of using separate methods implementing different tactics is the ability to configure the planner's tactical approach (doctrine) by enabling or disabling certain methods for planning.

In the example of Figure 13.7, the *TransportedMove* method was able to consider two combinations (picking up A before B, and B before A) and pick the optimal one, because it understood how the task would be implemented in terms of primitive tasks. Methods working with higher level tasks often lack the understanding of how the plan will work out in detail, and have troubles make an optimal (or even "good enough") choice by themselves when facing multiple combinations. In these cases, the method will indicate to the planner main loop that it sees more than one alternative to refine the plan. The planner main loop then will iterate over these alternatives and create new plans by cloning the parent plan and asking the method to refine the plan for the given alternative

(see Figure 13.4). Although this adds a little complexity to the planner main loop, the benefit is for us developers having to write and maintain just a single method to break down a specific compound task.

A method may fail to set a task output or break down a compound task and not generate a more refined plan. For example, if an artillery unit has already spent all its rounds in two artillery missions early in the plan, it should not be planned to perform a third artillery mission. If a team of three mechanized platoons is tasked to attack in formation but has to cross a narrow bridge doing so, it won't be an attack in formation and the method should not refine the task.

When one method fails to refine a task, this is only a local dead end in planning if no other method is capable of refining the same task in the same plan. Remember that we're searching through alternative plans with A*: a dead end here doesn't mean there isn't another, perhaps very different, variant of a plan that is feasible.

13.7 Plan-Space

The plan-space is the collection of all generated (partial) plans. We keep track of all plans that can be further refined in an open list. The open list is sorted for the lowest cost (Figure 13.8).

We can choose what to use for costs: plan duration works in most cases and is particularly suited for combat maneuvers, where time plays a considerable role in the plan's quality. The quicker we launch an attack, or the quicker our defending units occupy their positions, the better.

We compute a plan's duration the way project planners do, using accurate data from primitive tasks when available and using estimates for compound tasks that have not been detailed yet. Starting at the root task, we repeatedly pick a child task that has no preceding tasks without a start-time and end-time. For this child task we set as the start-time the maximum end-time of its predecessors, and recursively compute its duration and end-time (start-time plus duration). After doing so for all children, we can set the task's end-time. The root task's end-time minus start-time gives us the plan's duration.

We need to recompute a plan's duration every time we update the plan. Newly added primitive tasks may have a duration different from what their compound parent task estimated. We can, however, cache a compound task's estimate once computed for its specific inputs.

We leave the estimation of a compound task's duration to the task itself. Each compound task should implement an "estimate duration" function. These functions use heuristics to come up with a decent estimate. Since we are using A* to search through plan-space, the estimate should be a close estimate without overestimating the duration. Figure 13.9 illustrates how to come up with a good estimate.

Figure 13.9 shows the same situation and *TransportedMove* task as Figure 13.7. Now we are interested in estimating the duration without going into all the decisions and details that we considered when refining the task. A good estimate would be for C to move to A, then to B and finally to X at its top speed, with time for loading and unloading A and B added. In the estimate, we can decide to move to A before B based on a simple geometric comparison: A is closer to C than is B. Alternatively, we can evaluate path durations for both cases, and pick the lowest estimate. We are underestimating the real costs in most situations, since actual movement will be slower than C due to the terrain.

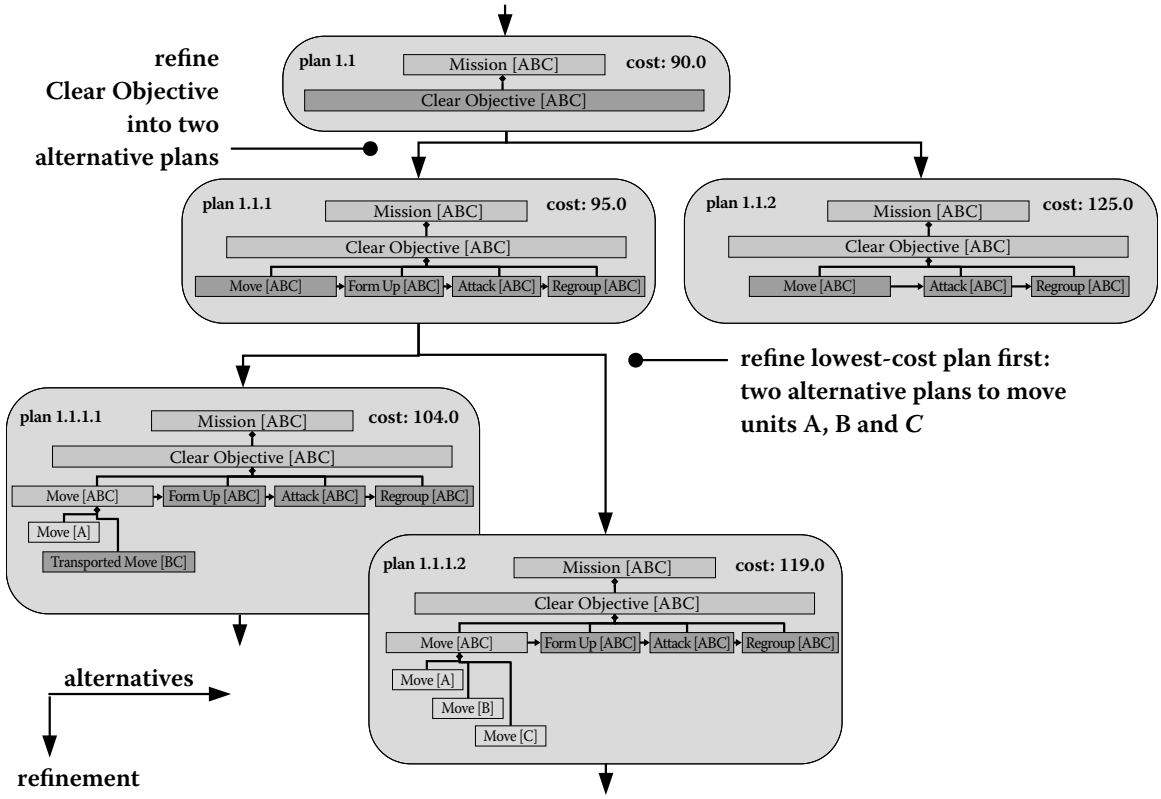
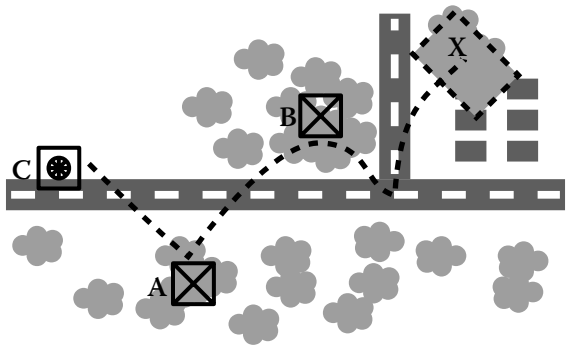


Figure 13.8

Plan-space, with incomplete plans as nodes and links representing refinement.



Cost (under) estimate for:
Transported Move (A,B,C) to X:
Path duration (C, A) +
Path duration (A, B) +
Path duration (B, X) +
2 × load time +
1 × unload time,
while assuming transport C
moves at maximum speed

Figure 13.9

Estimating the duration of a compound TransportedMove task.

As with A* pathfinding, we make the planner avoid certain tasks and plans by artificially inflating the duration of risky actions. For example, to make the attacker avoid using soft-skinned vehicles to transport infantry to the form-up location, we can raise the duration of the move task for soft-skinned trucks. When the planner also has available armored personnel carriers, he will be more likely to use these to transport infantry.

For tasks that are required for the plan, but not relevant for the quality of the plan, we may want to artificially deflate the duration. For example, for combat maneuvers, we typically don't have any use for transport helicopters after they have inserted their airborne infantry at a landing zone. We don't want their return flight duration to mask any duration differences in the tasks for the infantry's ground attack. To ignore the irrelevant return flight, we can use a small and fixed duration for the return flight tasks.

13.8 Making Planning More Efficient

As mentioned earlier, the biggest risk we run when creating plans for multiple units is the combinatorics problem (better known as the *combinatorial explosion*). Our hierarchical plan-space planner gives us several ways to reduce the number of options we consider, making planning for multiple units feasible and efficient.

First, we are using an A* search through plan-space expanding the lowest-cost "best" plan first. This helps us considering fewer options than a depth-first backtracking approach used by standard HTN planners.

Second, we are able to control the way an individual plan is expanded, and turn this into a "high-level decisions first" approach. In most cases, a plan will have more than one task that requires refinements and is grounded (has all its inputs set). The planner main loop in Figure 13.3 needs to pick a single task to refine. For the combat maneuver domain, where each task is associated with a command scope, we can have the planner main loop always pick the task with the highest scope as the task to refine first.

In Figure 13.10, this highest scope first task selection is illustrated. The partial plan consists of many compound tasks requiring refinement. Some of these, such as the *Attack* and *Regroup* tasks, cannot be refined yet, since they need inputs from preceding tasks.

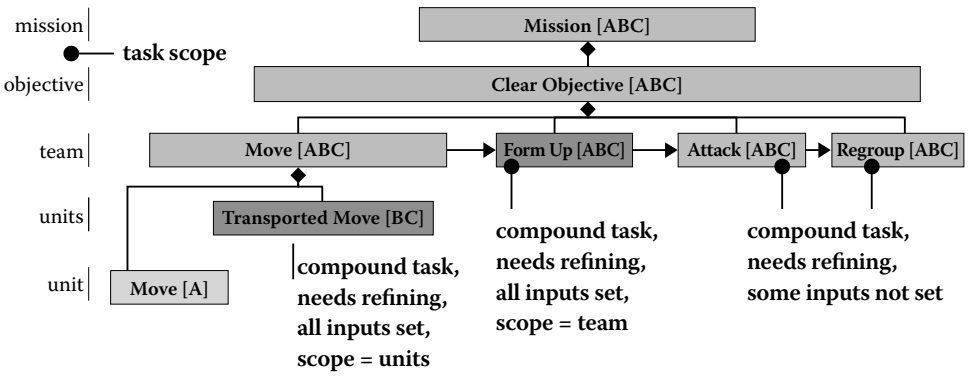


Figure 13.10

Selecting the highest scope task with all inputs set: FormUp.

Two tasks are grounded and ready to be refined: the *TransportedMove*, with "units" scope, and the *FormUp*, with "team" scope. Since "team" scope is higher, the planner will pick the *FormUp* task as the task to be refined first. Refining the *FormUp* task will set the inputs for the *Regroup* task, allowing that task to be refined next.

The benefit of refining higher level tasks first is that these tasks have larger impact on plan feasibility (do we have the maneuvering space for a combined attack by all our mechanized platoons?) and the cost of the plan. The planner should not busy himself detailing seating arrangements for the move to the form-up position before the attack is fleshed out. By making high-level decisions first, the planner needs far fewer steps to find a good plan.

A third way to consider fewer plans is the hierarchical plan-space planner's ability to plan from the "middle-out." In the military, planning specialists mix forward planning and reverse planning, sometimes starting with the critical step in the middle. When starting in the middle (for example, with the air landing or a complex attack), they subsequently plan forward to mission completion and backward to mission start. The military do so because starting with the critical step drastically reduces the number of planning options to consider.

We can mimic this by changing the input/output relations between tasks, and shifting some decisions from one method to another. Keep in mind that the only tasks that can be refined are the grounded tasks. Figure 13.11 shows an example of tasks connected to enable middle-out planning.

In Figure 13.11, a *ClearObjective* task is shown that has been broken down into a *Move*, a *FormUp*, an *AttackAfterFormUp*, and a *Regroup*. These tasks are to be executed in that order. However, refinement of these tasks should start with the *AttackAfterFormUp*. The input/output connections between the tasks are made in such a way that the *AttackAfterFormUp* is the first task having all its inputs set. The *FormUp* and *Regroup* task inputs depend on outputs from the *AttackAfterFormUp* task. The *Move* task depends on outputs from the *FormUp* task. The method refining the *AttackAfterFormUp* task has been

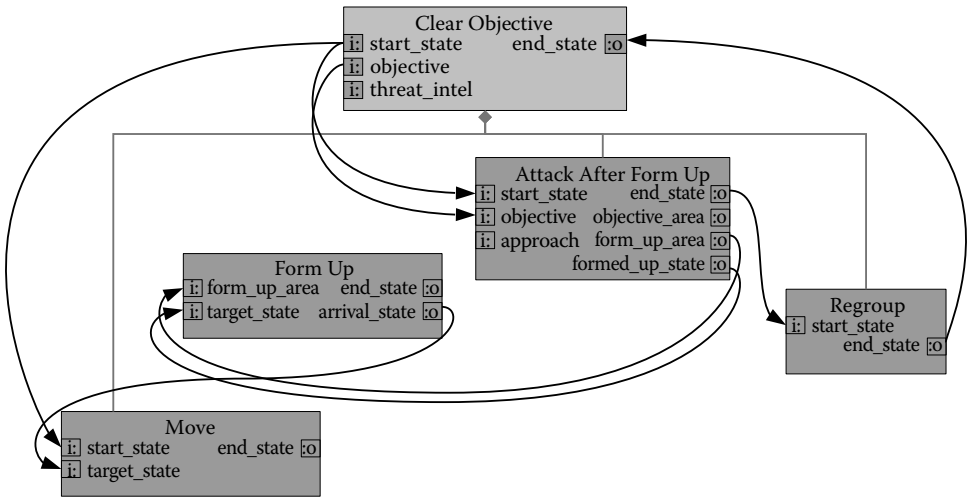


Figure 13.11

Middle-out planning from the AttackAfterFormUp by linking inputs.

modified to work with initial unit positions and the objective, and defines the attack move from the form-up locations through the objective. The *AttackAfterFormUp* task outputs the chosen form-up location and the positions of the involved units after form-up. It also outputs the positions of the units after attacking through the objective. These outputs enable the *FormUp* and *Regroup* tasks to be refined. The method refining the *FormUp* task defines where the units should enter the form-up area and with that output enables the *Move* task to be refined.

Middle-out planning requires changes to tasks and methods but it can greatly reduce the number of plans to consider by making critical decisions first. For combat maneuvers, middle-out planning also resembles a military practice, which makes it easier to translate military doctrine into tasks and planner methods.

13.9 Conclusion

We are able to successfully plan combat maneuvers involving over a dozen mechanized platoons, armor troops, gunship sections, and artillery batteries, taking into account tactical preferences and time. By working in plan-space instead of state-space, by breaking down the problem into high-level and low-level tasks and decisions, and by using a cost-based best-first search that expands high-level tasks first, we can avoid combinatorial explosion and deliver a good plan on short notice. The resulting plan includes not only the actions for each individual unit, but also the relations between these actions for coordination, and all higher level decisions. Turning such a plan into human understandable explanation or briefing is trivial.

The planner's design described here has been in action since mid-2009, generating tens of thousands of combat maneuvers from user input as downloadable missions [PlannedAssault 09]. The current implementation is in Ruby, running single-threaded on a Java VM (through JRuby) on an Intel Core2Quad Q8400, taking some 10s to 30s to generate a maneuver for 4×3 km terrain, with the majority of CPU time spent on terrain analysis and path-finding, not on plan expansion. The majority of plans are constructed in fewer than 200 planner main loop iterations.

13.10 Future Work

One nice side effect of planning in plan-space is the availability of all higher level tasks and decisions in the resulting plan, next to the actions for each of the units. Not only does this availability make it easier to turn the plan into a human readable briefing, it also makes the resulting plan great for use in monitoring the plan's execution. The original plan contains all the information to decide who is impacted by a task running late, which part of the plan needs repairs, and what the maximum allowed duration is for an alternative implementation of a plan part.

References

[Ghallab et al. 04] M. Ghallab, D. Nau, and P. Traverso. *Automated Planning, Theory and Practice*, pp. 229–259. San Francisco, CA: Morgan Kaufmann, 2004.

[Humphreys 13] T. Humphreys. "Exploring HTN planners through example." In *Game AI Pro*, edited by Steve Rabin. Boca Raton, FL: CRC Press, 2013.

[Orkin 06] J. Orkin. "Three states and a plan: The A.I. of F.E.A.R." *Game Developers Conference, 2006*. Available online (http://web.media.mit.edu/~jorkin/goap.html).

[PlannedAssault 09] PlannedAssault on-line mission generator for ARMA/ARMA2 games, http://www.plannedassault.com, 2009.

[StateSpaceSearch] Wikipedia. http://en.wikipedia.org/wiki/State_space_search.