

Decision-Theoretic Military Operations Planning

Douglas Aberdeen
National ICT Australia
Canberra, Australia
doug.aberdeen@anu.edu.au

Sylvie Thiébaux
National ICT Australia &
The Australian National University
Canberra, Australia
sylvie.thiebaux@anu.edu.au

Lin Zhang
Defence Science and
Technology Organisation
Edinburgh, South Australia
lin.zhang@dsto.defence.gov.au

Abstract

Military operations planning involves concurrent actions, resource assignment, and conflicting costs. Individual tasks sometimes fail with a known probability, promoting a decision-theoretic approach. The planner must choose between multiple tasks that achieve similar outcomes but have different costs. The military domain is particularly suited to automated methods because hundreds of tasks, specified by many planning staff, need to be quickly and robustly coordinated. The authors are not aware of any previous planners that handle all characteristics of the operations planning domain in a single package. This paper shows that problems with such features can be successfully approached by real-time heuristic search algorithms, operating on a formulation of the problem as a Markov decision process. Novel automatically generated heuristics, and classic caching methods, allow problems of interesting sizes to be handled. Results are presented on data provided by the Australian Defence Science and Technology Organisation.

Introduction

Operations planning is concerned with assigning appropriate tasks and resources for a mission, while minimising multiple, and possibly conflicting, costs. Task effects vary with probabilistic success or failure. The objective is to find a policy with the minimum expected cost, sensibly trading-off the individual cost criteria.

Existing planning codes schedule concurrent actions in probabilistic settings (Younes, Musliner, & Simmons 2003), but do not consider resources and conflicting costs. Other planners cope with concurrent tasks and multiple conflicting costs (Do & Kambhampati 2002), but do not consider probabilistic action outcomes. This paper deals with concurrent tasks, uncertainty, resources, task redundancy, and conflicting costs in a single planning tool. It demonstrates that using real-time heuristic search on a Markov decision process (MDP) formulation of the problem is a simple and effective approach to handling domains with these features.

Specifically, the planner uses the *labelled real time dynamic programming* (LRTDP) heuristic search algorithm (Bonet & Geffner 2003) as the core optimisation method. LRTDP starts with a factored (PDDL-like) representation of

the domain and incrementally generates and explores parts of the search space, guided by an admissible heuristic.

Approximately optimal policies are guaranteed to be found in finite time. The degree of approximation can be chosen. In practice, good plans are generated quickly and improve as more planning time is devoted to them. Optimisation of the policy can be based on minimising the probability of failure, makespan (policy duration), resource costs, or any ranking of those criteria. Optimised policies describe which tasks to start based on the history of task success and failure to the current point in time.

The quality of the heuristic has a large impact on the performance of LRTDP. A common theme in recent papers, also exploited here, is the automatic generation of good admissible heuristics.

This paper provides a description of the domain and previous work. It then explains how operations planning can be cast as an MDP, followed by a review of LRTDP and a description of the automatically generated heuristics. Experimental results are presented on synthetic and real operations planning scenarios, finishing with ideas for future work.

Military Operations Planning

Australian Defence Force planning doctrine stipulates four phases in the planning process (Zhang *et al.* 2002): (1) mission analysis, i.e., desired operational outcomes are described; (2) course-of-action development, i.e., tasks are chosen which may depend on previous operational outcomes and establish new outcomes; (3) course-of-action analysis, i.e., weaknesses in the course-of-action are identified and corrected; (4) decision and execution, the commander implements the most appropriate course of action. The four phase process repeats as new information becomes available. The planner described in this paper fits into phases (2) and (3), taking tasks created by planning staff, or using a repository of tasks, and generating a course-of-action along with low-level analysis.

Tasks are the basic planning unit, ranging from logistical, such as “Establish forward base”, to strategic or front line operations, such as “Amphibious assault on island”. Tasks are grounded durative actions. The outcome of a task is to set facts to true or false. Each task has a set of precondition facts, effect facts, resource requirements, a fixed probability

```

(:durative-action refuel-convoy
 :duration (= ?duration 10)
 :condition (and (at start convoy-needs-fuel)
                 (at start convoy-at-rendezvous)
                 (at start (>= cash 100))
                 (at start (>= tankers 1)))
 :effect (and (at start convoy-delayed)
              (at start (decrease cash 100))
              (at start (decrease tankers 1))
              (at end
                (probabilistic
                 0.95 (and (not convoy-needs-fuel)
                          (increase tankers 1))
                 0.05 (tanker-destroyed))))))

```

Figure 1: PDDL-like operations planning task description

of failure, and a fixed duration.¹ Tasks are assumed to run for exactly their specified duration, regardless of whether they succeed or fail. This is a conservative assumption because a commander might, in reality, react much earlier to a failure. A task can begin when its preconditions are satisfied and sufficient resources are available. A starting task removes resources from the global pool and may have some immediate effects. As each task ends a set of effects appropriate to success or failure are applied. These may set facts to true or false. Typically, but not necessarily, succeeding tasks set some facts to true, while failing tasks do nothing or negate facts. Resources are occupied during task execution and a variable fraction of these resources are permanently consumed when the task ends. Failure typically consumes more resources than success. See Figure 1 for a PDDL-like description of a task.

The policy objective is to make a subset of the facts true, corresponding to achieving all the desired operational effects. The form of the policy is a tree of contingency plans, termed a *schedule tree*. It describes which tasks should be started given the history of the operation up to the current time. Each branch, or contingency, in the schedule tree can be labelled with the probability that the contingency will be needed, the probability of achieving the operational objectives, the current truth values of the facts (sub-goal outcomes), and resource usage. Such information about possible outcomes is useful during the plan analysis phase.

Using this feedback the planner can be used in a semi-interactive fashion. If the schedule tree reveals that a policy is poor in terms of probability of success, makespan, or resource use, the tradeoff between these costs can be adjusted in a way designed to be intuitive to planning staff. A new schedule tree is generated using the same basic tasks, but with altered perception of the importance of achieving the objectives compared to time and resource use. The cost tradeoff mechanism will be described in detail.

Military planning staff can build redundancy into the task specifications, i.e., they may specify multiple tasks with similar effects. They may also specify tasks that can be repeated

¹The domain model used by the Defence Science and Technology Organisation allows for variable duration tasks, but this is simplified to a fixed duration for this paper.

if they fail — the default is that tasks are not repeatable, which contrasts with the default assumption in AI planning that allows multiple applications of actions. The planning software chooses the best tasks given previous successes and failures. Automated planning can also reveal inconsistencies, and remove duplicated effort, which might arise when multiple staff work on hundreds of tasks.

Existing Domain Tools

This paper extends the capabilities of the Course Of Action Scheduling Tool (COAST) developed by the Australian Defence Science and Technology Organisation (Zhang *et al.* 2002). COAST allows planning staff to describe the tasks that might be needed to carry out an operation. It includes a Coloured Petri Net (CPN) module for finding feasible plans. However, in its current form the CPN module does not take into account the probability of task failure or the cost of resources. Also, the analysis does not scale because it explicitly enumerates the *entire* planning state space.

JADE (Joint Assistant for Deployment and Execution) constructs plans for deploying military assets (Mulvehill & Caroli 2000). The user positions resources on a geographic map and a STRIPS style planner generates operations to deploy the resource to that location. JADE also makes use of a case based approach to re-use previous deployment plans. It does not take probabilities of success into consideration.

TOPFAS is a planning tool developed for the North Atlantic Treaty Organisation (Thuve 2001). A set of rich graphical interfaces guides commanders through the planning process but does not fully automate it.

Anecdotal evidence indicates that Microsoft Project remains a popular tool within the domain.

Related AI Planning Work

The AI planning community has long been interested in planning concurrent tasks under temporal and resource constraints (Tate, Drabble, & Dalton 1996). Somewhat independently, the machine learning community has developed Markov decision process (MDP) algorithms (Howard 1960) to cope with the uncertainty in feedback systems. The mixing of these two fields, termed decision-theoretic planning (Blythe 1999), provides the benefits of factored state and action descriptions, with the real world necessity of dealing with uncertainty.

Closest to the present paper is that of Younes, Musliner, & Simmons (2003), who adopt a generalised semi-Markov process to describe concurrent actions. This allows them to model uncertainty in state transitions (task consequences) and state durations (task durations), but only the former is considered in this paper. Younes, Musliner, & Simmons apply a *continuous stochastic logic* plan verification-repair method, which uses heuristics to guide the repair phase, but does not consider resources and cost tradeoffs.

Previous work involving conflicting makespan and resource costs includes that of Do & Kambhampati (2002), who hard-wire a linear tradeoff between costs. Haslum & Geffner (2001) attempt to optimise for both makespan and resources assuming there is no interaction between the two. Both of those papers, and Younes, Musliner, & Simmons,

make use of automatically generated heuristics to accelerate planning. The heuristics developed here share some ideas with those used by Gerenini & Serina (2002) in a Graphplan context.

In general, previous research has combined temporal planning with uncertainty (Precup, Sutton, & Dasgupta 2001), and in some instances with resources as well (Atkins 1999), but without concurrency.

Other applications of planning to the military are diverse. Meuleau *et al.* (1998) shows how a complicated multiple-target limited-weapons assignment problem can be decomposed into simple subproblems, each of which achieves a sub-goal of destroying one target. Each sub-problem is solved using a value-iteration approach, similar to this paper. The sub-problems are later recombined heuristically. The assumption is that concurrent tasks are nearly independent, which is not the case for general operations planning. Laskey *et al.* (2000) propose a hierarchical Bayesian network for fusing many sources of uncertain information in real time. The military application is to fuse intelligence sources to alert commanders to important changes on the battlefield.

MDP Formulation of Operations Planning

This section explains how operations planning can be cast as an MDP. This involves describing the MDP framework, and then defining, in the particular case of operations planning, the MDP state and action spaces as well as the cost structure.

Definition 1. *A finite Markov decision process consists of:*

1. a finite state space \mathcal{S} ;
2. a finite set of actions \mathcal{A} ;
3. an initial state $s_0 \in \mathcal{S}$;
4. a set of terminal states $\mathcal{T} \subseteq \mathcal{S}$;
5. computable probabilities $\Pr[s'|s, a] : \mathcal{S} \times \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$ of making state transition $s \rightarrow s'$ under applicable action a ;
6. the cost of each state $c(s) : \mathcal{S} \rightarrow \mathbb{R}$.

In the context of this paper, an action is a decision to start a subset of the tasks, explained in detail below. Also, terminal states in \mathcal{T} have associated costs representing operation success or failure and should not be confused with the operation goal states.

A solution to an MDP is a stationary policy $\pi(s) : \mathcal{S} \setminus \mathcal{T} \rightarrow \mathcal{A}$, mapping non-terminal states to actions. This paper assumes that all policies reach terminal states in finite time when executed from s_0 . This is enforced by limiting the maximum makespan.

The aim of MDP algorithms is to find the optimal policy to move from s_0 to a terminal state in \mathcal{T} . Such a policy incurs the minimum expected sum of costs, also called the optimal *value* of the state s_0 . After optimisation, the action with the optimal value in each state defines the policy, which therefore specifies the “best” action to choose at each step.

More formally, the value of state $s \in \mathcal{S} \setminus \mathcal{T}$ can be computed using *value iteration*² (Howard 1960):

$$V_{t+1}(s) = c(s) + \min_a \sum_{s' \in \mathcal{S}} \Pr[s'|s, a] V_t(s'). \quad (1)$$

LRTDP is essentially an efficient implementation of (1). One efficiency gain arises from using the Q form of values

$$Q_{t+1}(s, a) = c(s) + \sum_{s' \in \mathcal{S}} \Pr[s'|s, a] \min_{a'} Q_t(s', a'), \quad (2)$$

which reduces computation time at the expense of storing values for each (state,action) pair. Only the actions valid from each state are considered. In following sections, $V_t(s)$ is used as shorthand for $\min_a Q_t(s, a)$. The values $Q(s, a)$ are considered converged when $V_{t+1}(s_0)$ cannot change by more than ϵ from $V_t(s_0)$. The final policy is formally defined by $\pi(s) = \arg \min_a Q(s, a)$.

Operations Planning State Space

For operations planning the state description contains: the state’s time, a queue of impending events, a list of eligible tasks, the truth value of each fact, the available resources, and each cost component.

Any task that has not yet been started, or failed and is repeatable, is considered an *eligible* task for the purposes of the state description. In a particular state, only a subset of the eligible tasks will satisfy all preconditions for execution. This subset is called the *enabled* task list.

Operations Planning Action Space

The enabled list of a state is used to determine the available actions. The set of valid actions is the power set of the enabled tasks. Any action that violates resource limits or other mutual exclusion requirements, as in PDDL 2.1, is removed from the action set. For n enabled tasks the state has a maximum of 2^n valid actions, including 0 task starts. Starting no task can be useful if delaying to the next event allows a better task to be started.

An alternative to the power set of enabled tasks is to consider each task in turn. An independent decision is made for each of the n enabled tasks. The fundamental branching factor is the same because there are 2^n outcomes of making n binary decisions. However, making independent decisions can be sub-optimal because of resource constraints. Starting one task may leave insufficient resources for preferable tasks yet to be considered. Bacchus & Ady (2001) adopt the independent decision approach, avoiding resource and other conflicts by relying on the domain writer to block conflicting actions through use of preconditions.

The generation of these action sets is shown in Algorithm 1. The `findSuccessors()` routine determines the successor states and their probabilities for each action. Thus, line 10 stores a distribution of possible successor states for action a .

²Equation (1) is often expressed with a discount factor in the second term. However, this artificially attributes more weight to short-term costs and is not necessary if all policies terminate in finite time.

Successor State Generation

The generation of successor states is shown in Algorithm 2. The algorithm first checks termination conditions (lines 2-13). Then, the next event for state s is processed. Events have probabilistic consequences. For example, tasks can end in two ways: success or failure. Enhanced task definitions may include more than two outcomes. Line 15 of Algorithm 2 loops over all possible outcomes of the event.

Future states are only generated at points where tasks can be started. Thus, if an event outcome is processed and no tasks are enabled, the search recurses to the next event in the queue. In the worst case no new tasks are enabled until all events on the queue are processed (see lines 19-22). E.g., if n tasks-ends are on the event queue, the worst case can return up to 2^n new reachable states. The actual number of successor states is usually much less. For example, if a task is enabled regardless of the success or failure of the next event, the `findSuccessor()` search will return immediately with exactly 2 successor states.

Event queues have previously been used in temporal planning (Bacchus & Ady 2001; Do & Kambhampati 2002). This setting extends the event queue to decision-theoretic planning.

Algorithm 1 findActions(State s)

```

1: Variables: Action  $a$ , State  $s'$ , State newState, StateList
   successors, Task  $t$ 
2: for each  $a$ =subset of non-mutex tasks do
3:   newState =  $s$ .copy()
4:   for each  $t$ =task started by  $a$  do
5:     newState.startTask( $t$ )
6:     newState.addEvent(end of  $t$ )
7:   end for
8:   newState.prob = 1.0
9:   findSuccessors(newState, successors)
10:   $s$ .addAction( $a$ , successors)
11: end for

```

State Costs

MDP algorithms generally assume a single scalar cost per state $c(s)$. However, operations planning involves multiple conflicting costs, e.g., minimising operation duration may require expending more resources. The costs components for the MDP states are: (1) a fixed failure cost if the goal conditions become unreachable; (2) the duration of a state; and (3) the resources consumed over the duration of the state. These individual components are computed as a side-effect of Line 17, Algorithm 2.

A scalar cost could be formed by an explicit linear combination of individual costs (Do & Kambhampati 2002), however, military commanders are unwilling to assign specific weights to each of these costs. They are more willing to rank the costs.

Risk adverse MDP algorithms (Geibel 2001) could be extended to this setting. These algorithms successively re-plan, starting with the most important cost and working

Algorithm 2 findSuccessors(State s , StateStack successors)

```

1: Variable: State newState, Stochastic Event event
2: if  $s$ .time > maximum makespan then
3:   successors.pushFailGoal( $s$ )
4:   return
5: end if
6: if  $s$ .operationGoalsMet() then
7:   successors.pushSuccessGoal( $s$ )
8:   return
9: end if
10: if  $s$ .eventQueueEmpty() &  $\neg s$ .findEnabledTasks() then
11:   successors.pushFailGoal( $s$ )
12:   return
13: end if
14: event =  $s$ .nextEvent()
15: for each event.outcome do
16:   newState = copyState( $s$ )
17:   newState.implementEffects(event.outcome.effects)
18:   newState.prob =  $s$ .prob  $\times$  event.branch.prob
19:   if newState.tasksEnabled() then
20:     successors.push(newState)
21:   else
22:      $s$ .findSuccessors(newState, successors)
23:   end if
24: end for

```

down the ranking. Such algorithms guarantee that more important costs are never traded for less important costs.

This paper takes a less restrictive and computationally faster approach. A cost function is constructed that places exponentially more weight on more important costs. If the exponent α is sufficiently large the state cost $c(s)$ will reflect the desired ordering exactly. A smaller exponent α allows large but low-importance costs to have equal weight with small but important costs. Thus, the exponent can act as a tuning parameter, allowing the planning staff to specify how strong the cost ordering should be.

Let $c(s, i)$ be cost component $i \in 0, \dots, C-1$ in state s . Define $i=0$ as the least important cost, $i=1$ the next least important cost, and so on. The state cost function is

$$c(s) = \sum_{i=0}^{C-1} c(s, i)\alpha^i. \quad (3)$$

Suppose planning encounters a state s with cost components $c(s, 0) = 2$, $c(s, 1) = 71$, $c(s, 2) = 1$. If $\alpha = 100$ then $c(s) = 17102$. The costs in this example represent 2 resource units, 71 duration units, and 1 failure unit. So $c(s)$ has been constructed to ensure the cost of operation failure dominates all other costs. For smaller α , for example $\alpha = 50$, using 71 duration units would dominate the cost of failure. Depending on the needs of the commander, this might be a sensible tradeoff.

However, the long-term values $Q(s, a)$ computed with (2) and (3) are not guaranteed to maintain the ordering on costs in the sense that

$$Q(s, a) \geq \sum_{i=0}^{C-1} Q(s, a, i)\alpha^i, \quad (4)$$

where $Q(s, a, i)$ is the long-term value computed with cost component i only. To see this, consider how the \min_a operator in (2) induces a single policy for value function $Q(s, a)$ but may select different, and independently minimising, policies for each value function $Q(s, a, i), i = 0, \dots, C - 1$. Practically, the final policy may not be the policy that completely minimises the costs in strict order. Rather, there is always some trade-off between the costs, decreasing as α increases. To guarantee strictly ordered costs, slower risk adverse algorithms can be used (Geibel 2001). For operations planning, the ability to trade-off costs is desirable. Planning staff initially define an order on costs and chooses a large α . If analysis of the first policy reveals high probability of undesirably large lower-importance costs, then α can be decreased and a new policy regenerated.

This scheme is essentially just a linear weighting scheme, however the weights are chosen to reflect a *soft* ordering on the costs. The degree of softness is controlled by α .

Finally, the experiments in this paper all resources are of equal weight, although the cost ordering can be trivially extended to each resource.

LRTDP

LRTDP (Bonet & Geffner 2003) is an efficient implementation of value iteration (2). Efficiency gains come from: using greedy simulation to focus on relevant states, using admissible heuristics, and labelling converged states. Furthermore, this implementation relies on efficient data structures and memory management policies to store visited states.

Alternative heuristic search algorithms, such as LAO* (Hansen & Zilberstein 2001), could also be used. LRTDP was chosen because there is some evidence that it can be more efficient than LAO* (Bonet & Geffner 2003).

Greedy Simulation

LRTDP starts with a factored representation of the state space, i.e., probabilistic planning operators. Starting at s_0 , it simulates the current greedy policy, explicitly generating reachable states along the way, and updating their values. The order of state updates ensures rapid convergence of the values for states that are reached when following the current greedy policy.

Convergence to the optimal policy is guaranteed despite the use of a greedy action selection policy. Because values monotonically increase to their true long-term expectation, all poor actions are eventually exposed and rejected in favour of actions that lead to smaller state values. Good admissible heuristics can reduce the number of updates by orders of magnitude (Bonet & Geffner 2003), also reducing the number of states that are explicitly generated.

LRTDP is described by Algorithm 3. The `action = s.greedyAction()` routine returns $\pi(s)$. If this is the first time a transition out of s has occurred, `s.greedyAction()` invokes Algorithm 1 to find possible successor states for each eligible action. The `s.update()` routine evaluates (2) for state s . Finally, the `s.pickNextState(a)` randomly chooses the next state according to the distribution induced by $\Pr[\cdot|s, a]$ and returns this state.

Labelling

The `checkSolved(s, ϵ)` function check does a convergence check on all states seen during a simulation trial. An algorithm for this routine is given by Bonet & Geffner (2003). If $Q(s, a)$ changed by less than ϵ , and all possible state trajectories from s encounter only solved states, s is labelled as solved. Thus, the algorithm terminates when all states encountered by the current policy cannot change their value by more than ϵ in one update. The ‘real-time’ name is used because the policy improves rapidly early on, then converges to the ϵ -optimal policy. Thus, early termination results in a useful policy.

Memory Management

States visited by LRTDP are stored in a hash table³ for future use. This implementation also stores information about transitions for which $\Pr[s'|s, a] > 0$. This is done in Algorithm 1, which adds successor states to the global hash table (line 10). If the state already exists in the hash then alternative branches of the policy have merged and the state is not added again. Algorithm 1 is only run the first time a transition out of the state s is requested. All subsequent transitions out of state s retrieve the previously computed successor state distribution for the desired action. Such storage is useful for MDP methods because any given state s is visited many times, updating its value $Q(s, a)$ each time. Recomputing the state variables and searching for successor states every time is too slow. At the very least, values for “interesting” states must be stored.

Although the majority of the state space is typically not visited, even modest operations planning scenarios with around 25 tasks still visit millions of states. Further measures are needed to ensure planning does not consume all available memory. When values $Q(s, a)$ indicate an action is very expensive, LRTDP’s greedy policy ignores that state and its descendents for the rest of the search. Such obsolete states can account for the majority of memory use and should be ejected. However, it is possible for an obsolete state to become active again. States that move in and out of the active policy demonstrate strong temporal locality. That is, if a state has not been part of the active policy for a long time, it is highly unlikely to be part of the active policy in the future. This motivates a least recently used replacement policy for states in memory. The hash table is checked for obsolete states when memory is low. A state is deleted if it is not part of the active policy and is in the bottom 30% of most recently visited states. If a deleted state is revisited it is regenerated, but its value must be learnt from the beginning. Overall, the benefit of keeping states is retained while avoiding an explosion of states in memory.

³The hash key is a string representation of the state, summarised using the MD5 message digest algorithm. The final key is a 32 bit hexadecimal string which can be hashed quickly and reduces memory usage per state compared to the full string. The probability of two states mapping to the same key is negligible.

Algorithm 3 LRTDP(State s_0 , Real ϵ)

```
1: Variables: State  $s$ , StateStack visited, Action  $a$ 
2: while  $\neg s_0$ .solved do
3:   visited = EMPTY STACK
4:    $s = s_0$ 
5:   // Simulation loop
6:   while  $\neg s$ .solved do
7:     visited.push( $s$ )
8:     if  $s$ .goal then break
9:      $a = s$ .greedyAction()
10:     $s$ .update( $a$ )
11:     $s = s$ .pickNextState( $a$ )
12:   end while
13:   while visited  $\neq$  EMPTY STACK do
14:      $s =$  visited.pop()
15:     if  $\neg$  checkSolved( $s$ ,  $\epsilon$ ) then return
16:   end while
17: end while
```

Heuristics

The aim of heuristics is to provide the best possible guess of $V_0(s)$ more efficiently than LRTDP could converge to the same value. *Admissible heuristics* compute initial values such that $V_0(s) \leq V(s)$ for all s . If a heuristic mistakenly sets $V_0(s) > V(s)$ then that state and its descendents may never be searched because of the greedy action selection policy. If those states turn out to be the best path to the goal, the mistake will cause the final policy to be non-optimal.

This section provides separate heuristics for computing lower bounds on the probability of failure $V(s, fail)$, makespan $V(s, makespan)$, and resource consumption $V(s, resource)$. If a non-terminal leaf state is encountered during the LRTDP search then successor states are computed for each valid action (Algorithm 2). Heuristic values are computed for each successor and combined using (3), forming a heuristic successor state value that is the same for all actions. The successor values are propagated back to the original leaf state by a value update for all choices of leaf-state action. This provides a different heuristic value for each action, hence guiding the selection of actions. The new successor states are now leaf states. In some instances the heuristics can detect if an operation success goal has become unreachable, allowing the successor state to be labelled as a failure terminal state. All the heuristic lower bounds are computed from a list of tasks that satisfy each desired outcome. Outcomes are expressed as the setting of goal facts to true or false.

Probability of Success

Tasks set facts to true or false, depending on the success or failure of that task. If all the tasks that can establish the goal state of a fact fail, then the operation goal cannot be reached. Detecting this situation can be done neatly by expressing the overall goal state as conjunctive normal form boolean equation of facts. Each clause represents a goal, written as a fact being true or false. Each clause is then expanded into the task successes or failures that set that fact to

the desired value. For example, suppose there are two goal facts $F_1 = true$, written as just F_1 ; and $F_2 = true$, written as just F_2 . Suppose task t_1 *succeeding* asserts F_1 and F_2 ; t_2 *failing* asserts F_2 ; and t_3 *succeeding* asserts F_2 . Embedded in a boolean equation, t_i should be read as true if the task completed successfully, and false if it failed, thus

$$\begin{aligned} success &= F_1 \wedge F_2 \\ &= (t_1 \vee t_3) \wedge (t_1 \vee \neg t_2). \end{aligned}$$

If a task t_i is yet to run, or is repeatable, any clause containing t_i or its negation immediately evaluates to true because it may be true at a later time. The boolean expression is trivial to compute before optimisation begins. If it evaluates to false at state s , the operation goal is no longer reachable and state s is a failure terminal state.

An upper bound on the probability of reaching a success state is

$$\begin{aligned} \Pr[success] &= \Pr[F_1 = true \wedge F_2 = true] \\ &= \Pr[F_1 = true] \Pr[F_2 = true | F_1 = true] \\ &< (1 - \Pr[t_1 = fail]) \Pr[t_3 = fail] \times 1. \end{aligned}$$

Terms $\Pr[t_i = fail]$ are known from the task description. The term for $\Pr[F_2 = true | F_1 = true]$ evaluates to 1 because t_1 can satisfy both F_1 and F_2 , i.e., the probability that t_1 succeeds has been integrated into computing the probability that F_1 is true. To avoid double counting $\Pr[t_1 = fail]$, while still maintaining the upper bound, t_1 is assumed to have succeeded. Better heuristics could remove this assumption. Tasks that have already succeeded or failed have their probabilities set to 0 or 1 accordingly.

The required lower bound on failure is the upper bound on success subtracted from 1. This bound ignores the fact that tasks cannot run until all their preconditions are met, which can only increase the probability that a task will not run, and hence only increases the probability of failure.

Makespan and Resource Bounds

These bounds are loose but fast to compute given the list of which tasks assert which facts. The makespan bound is the *maximum* of all the durations required to establish each goal fact. The duration for each goal fact is the *minimum* remaining duration for each task establishing the fact. If \mathcal{F} is the set of goal facts, \mathcal{T}_F is the set of tasks that assert F , and $dur(t)$ is the duration of task t , then

$$V(s, makespan) \geq \max_{F \in \mathcal{F}} \min_{t \in \mathcal{T}_F} dur(t).$$

A simple resource bound is the sum of the minimum resources that would be consumed establishing each goal fact. This assumes the task that uses the least resources will successfully establish the fact. However, tasks that contribute to multiple goal facts must not have their resources counted twice. Exactly computing the subset of tasks that assert all goal conditions, with minimal resources, is an NP-hard problem. A sub-optimal solution is to divide the resources for a task by the number of goal facts the task asserts, then allow task resources to be re-counted. If a task has the lowest resource use for all the facts it asserts, then that task will

contribute exactly its true resource usage. If a task has the lowest resource use for only some of the facts it asserts, then it contributes less than the true resource use. If $res(t)$ is the minimum resources consumed by task t , then

$$V(s, resources) \geq \sum_{F \in \mathcal{F}} \min_{t \in \mathcal{T}_F} \frac{res(t)}{|\{F : t \in \mathcal{T}_F\}|}.$$

If the minimum resources needed is greater than the minimum resources available and tied up in running tasks, then the operation success state is unreachable.

Experiments

Initial testing was performed on 85 synthetic planning scenarios. A second set of experiments focuses on two scenarios provided by the Australian Defence Science and Technology Organisation, based on de-classified information from military exercises.

Synthetic Scenario Generation

Synthetic scenarios are a poor substitute for real domain data. However, in the absence of a large database of domain scenarios, synthetic scenarios were useful for testing the methods presented in this paper.

Synthetic scenario generation attempts to mimic the domain. Each scenario consists of 25 tasks and 25 facts. The goal state of the synthetic scenarios is to assert all facts to be true. Redundancy is introduced into the planning scenario by allowing tasks to satisfy multiple facts. On average, half the tasks set one fact and half set two facts. For example, Task 1 might set Fact 3 and Task 2 might set Facts 3 and 4. A good policy would be to try Task 2 first — because it satisfies multiple facts — and only try Task 1 if Task 2 fails.

Tasks depend on a subset of the facts affected by previously generated tasks. Generation also ensures that there is scope for concurrent tasks.

The resources occupied by tasks are drawn from a initial pool of 10 types of resource, each with 20 units. Each task uses a random amount from up to 5 resources types. The resources consumed on failure are also generated randomly, possibly consuming all allocated resources. On success, a maximum of half the allocated resources are consumed. Resource usage is high enough to constrain the feasible policies. All resources are assumed to have equal cost, so the maximum cost of a task's consumed resources is $5 \times 20 = 100$ units. Task durations are drawn randomly from $[1, 100]$. The maximum makespan permitted is 2500 units, despite the fact that repeating tasks allow the goal to be reached after 2500 units. Tasks fail with 0 to 40% probability. One tenth of tasks are designated as repeatable.

To summarise, these synthetic planning scenarios are relatively small, but do have scope for choosing tasks instead of merely scheduling them. All synthetic scenarios are guaranteed to have at least one policy which will reach the operation goal assuming all tasks succeed.

A loose upper bound on the number of states for any scenario is $O(Md^T 2^C u^R)$ where M is the maximum makespan, d is the maximum task duration, T is the number of tasks, C is the number of conditions, u is the maximum

Table 1: Results averaged over 85 sythetic scenarios, with 25 tasks each. The H column indicates if heuristics were used. MS is the average makespan, Res is the average resource consumption, Secs is the mean optimisation time, and $|\mathcal{S}|$ is the number of visited states. Each policy is tested with 100,000 simulations of the scenario.

	H	Fail %	MS	Res	Secs	$ \mathcal{S} $
No Opt		96.5	346	47.8		
No Opt	•	88.6	355	51.0		
Time/Res		78.4	335	53.2	231	197000
Time/Res	•	76.8	332	52.0	131	120000
Res/Time	•	77.0	383	49.9	144	86400
$\alpha = 1$	•	83.6	231	44.5	81	54900

resource units over all resource types, and R is the number of resource types. For 25 tasks, maximum duration 100, makespan limit 2500, 25 conditions, and 20 units for each of 10 resource types, there are 6×10^{73} representable states. Although only a small fraction of these states can ever be realised, it demonstrates the potential for huge state spaces and the importance of efficient heuristic search.

LRTDP optimisation was performed with various settings and compared to a hard-wired ‘No Opt’ policy which always starts one task. Failure was always the the most important cost, with an individual component of $c(s, 2) = 1000$ for failure and 0 otherwise. The actual cost of failure is much higher after applying (3). For the ‘Time/Res’ optimisations makespan is more important than resource use. Similarly, ‘Res/Time’ optimisations make resources more important than time. The exponent α used in (3) was set to 1000 for all optimisations except those labelled ‘ $\alpha = 1$ ’, for which individual costs are simply added. Optimisations were run with and without heuristic initial values. Heuristics can also be used to modify the ‘No Opt’ policy, choosing the set of tasks with the lowest heuristic value as the final policy.

The ϵ parameter was set to 1, thus optimisation halts when the initial state value $V_i(s_0)$, and all descendant state values, cannot change by more than 1. Out of 85 scenarios, 5 failed to halt within a 10 minute time limit. Optimisations were performed on a cluster of 85 Pentium III 800MHz Coppermine CPUs, each with 384Mb of memory.

Results Table 1 shows the final value of the three cost criteria averaged over the 85 planning scenarios. The cost for each scenario was the average over 100,000 test runs. The results show that optimisation reduces the probability of operation failure and the makespan of the operation. Figure 2 shows, for each scenario, the ‘Time/Res’ improvement in probability of failure. Most of the unoptimised policies fail because resources run out. The lowest failure rate is still high at 76.8%, reflecting the true difficulty of the scenarios rather than the optimisation algorithm failing. When resources are more important than duration there is a reduction in the resource usage and increase in duration. This demonstrates the ability of the planner to trade-off cost criteria.

A decrease in duration and resource use was observed when α was decreased from 1000 to 1. A corresponding increase in the probability of failure demonstrates the desired

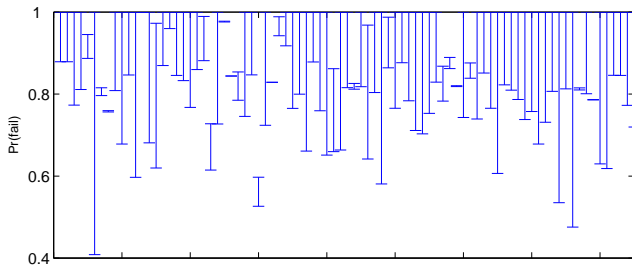


Figure 2: For each synthetic operation the top of the bar is the pre-optimisation failure probability, and the bottom is post ‘Time/Res’ heuristic optimisation failure probability.

effect of trading failures for a decrease in the lesser costs.

Initialising $V_0(s)$ using the automatic heuristics had a benefit even without optimisation. Using heuristics during optimisation reduced both optimisation time and the number of states explored. The slightly different average costs using heuristics are expected if the optimisation does not run to $\epsilon = 0$ convergence.

Island Assault Scenario

Figure 4 lists 20 tasks, provided by the Defence Science and Technology Organisation, modelling an assault on an enemy occupied island. There is a large amount of redundancy in the tasks, making this scenario much more complex than the synthetic planning scenarios. For example, ‘‘Amphibious assault on island’’, and ‘‘Airborne ops to occupy island’’, both achieve the final goal which is to set fact ‘‘Objective island secured’’ to true. Also, each fact can be asserted by up to 3 tasks. However, none of the tasks are repeatable. Very few resources are consumed in this scenario. Thus, while resource *allocation* constrains the policy space, resource *consumption* costs do not differ much over the feasible policies.

For this scenario the maximum makespan was set to 300 hours. Choosing the lowest reasonable makespan minimises the search space. As before, optimisations were limited to 10 minutes with $\epsilon = 1$. All the optimisations ran to the 10 minute time limit. The real time qualities of LRTDP are demonstrated by Figure 5. After 10 minutes $V(s_0)$ has reached 88% of the value obtained after 22 minutes. This indicates that truncating optimisation at 10 minutes still provides a reasonable policy.

Results Table 2 summarises results obtained for various optimisation parameters after 30 optimisations and 100,000 test runs of each optimised policy. As observed for synthetic scenarios, adding heuristics immediately improves the results. Applying LRTDP optimisation further improves the results, especially the makespan.

Some standard deviations appear quite high. Figure 3 provides an indication of the true spread of makespan and resources after one optimisation, revealing multiple modes which tail off towards high values of time and resources. This kind of analysis of possible outcomes can be useful to planning staff.

Table 2: Results on the Assault Island scenario. The \pm quantities indicate one standard deviation over 30 optimisations runs. Each policy is tested with 100,000 simulations of the scenario.

H	Fail %	MS	Res
No Opt	13.5	213	9.11
No Opt •	12.5	204	9.00
Time/Res •	7.47 ± 0.867	119 ± 14.9	9.83 ± 1.12
Time/Res	7.92 ± 0.728	123 ± 10.6	10.2 ± 1.63
Res/Time •	7.65 ± 0.828	162 ± 31.1	10.9 ± 1.89
$\alpha = 1$ •	7.48 ± 0.704	117 ± 14.2	$.040 \pm .004$

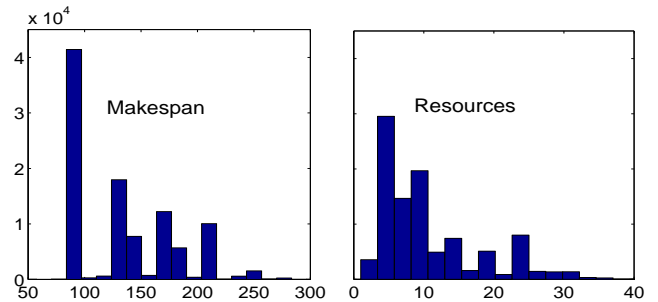


Figure 3: Histogram of makespans and resource use for 100,000 trial runs of one ‘Time/Res’ optimisation of the Assault Island scenario.

A *schedule tree* is a graphical representation of a policy for durative actions with uncertain consequences.⁴ Nodes represent action decision points in the policy. Arcs are labelled with the tasks to begin and the tasks which are assumed to succeed or fail in that period. The length of an arc represents the period until the next action point. Figure 4 provides the task list and schedule tree that results from a Time/Res optimisation on the Assault Island scenario. To save space, only the nodes with probability greater than 0.1 have been shown. The first node can be interpreted as: start with anti-submarine operations using submarines and disrupt enemy air capabilities with fighter jets. The next action point is 12 hours later, when enemy air capability has been knocked out with probability 60%.

Webber Island Scenario

The second set of provided tasks also models the capture of an Island, however this scenario is more detailed. A full description of this scenario would require more space. In brief, there are 41 tasks and 51 facts. Tasks range from ‘‘Insert special forces’’, to ‘‘Provide air-to-air refuelling’’. There are 19 resource types. Two examples include an airborne battalion and 6 mine counter measure ships. The operation goal is for the ‘‘Enemy Forces on Webber Island evicted’’ fact to become true. There are 8 tasks that establish this goal, and

⁴Visualisation of complex policies with probabilistic contingencies is an interesting problem. Schedule trees provide one solution, but do not scale well and are not friendly to planning staff trying to interpret results.

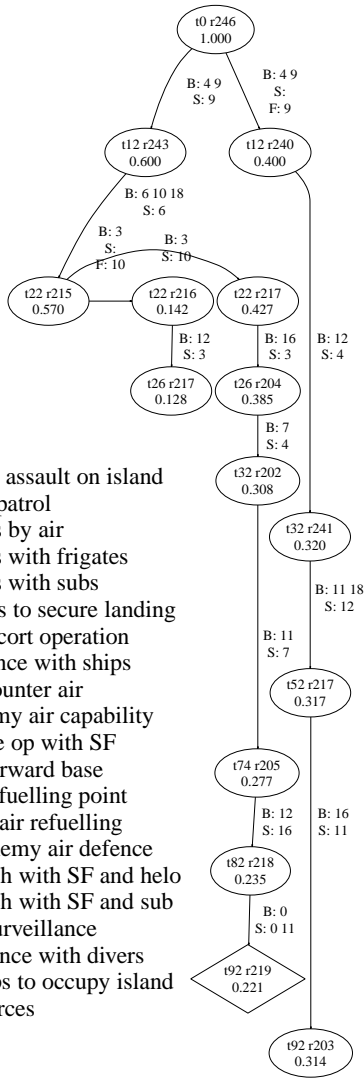


Figure 4: Truncated schedule tree for the Assault Island operation; optimised for probability of failure, duration, then resources. The diamond is a goal state. The numbers 't0 r246 1.00' are the state time stamp, remaining resource units, and the state probability. The arc labels such as 'B: 4 9 S: 9 F: 4' are the tasks to begin, successful tasks, and failed tasks respectively.

many ways to establish the preconditions for those 8 tasks. This dataset approaches the complexity that will be encountered during real operations planning.

Optimisation and testing was performed identically to the Assault Island scenario, except that the maximum makespan was increased to 1000 hours to account for tasks that take up to 288 hours to complete.

Results This data set proved extremely challenging. The results presented in Table 3 are preliminary only. A large number of states are generated very quickly. There are often 9 or more tasks that can run at any one time, resulting in

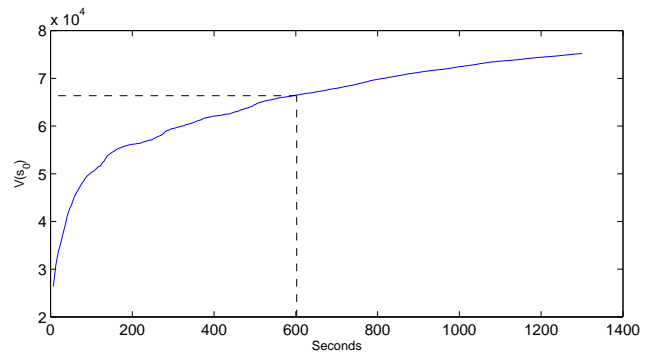


Figure 5: Convergence of the initial state value $V(s_o)$ for one 'Time/Res' optimisation of the Assault Island scenario. The quoted result is based on 600 seconds of optimisation.

Table 3: Results on the Webber Island scenario. Results are presented identically to Table 2.

H	Fail %	MS	Res
No Opt	58.4	675	4.73
No Opt •	58.3	670	4.74
Time/Res •	58.1 ± 0.217	239 ± 22.3	4.52 ± 0.587
Time/Res •	58.0 ± 0.275	235 ± 23.9	4.44 ± 0.349
Res/Time •	58.1 ± 0.245	257 ± 39.0	4.28 ± 0.336
$\alpha = 1$ •	58.0 ± 0.268	237 ± 20.3	$0.016 \pm .001$

several hundred valid actions at some decision points. It was noted that the majority of the 10 minutes of optimisation time was spent doing operating system memory allocation and deallocation. This will be rectified by writing dedicated memory allocation routines for the planning code.

When optimisation is used there is a small decrease in the probability of failure, and a large decrease in the makespan. A small degree of tradeoff occurs between runs where the importance of time and resources are reversed. A single-sided t-test indicates 98% confidence that the observed tradeoff is significant, for both time and resources. The no-heuristic optimisation result is under investigation because it seems to do *slightly* better than the heuristic version. A t-test shows indicates a 94% confidence that this is true for probability of failure, and a 75% confidence that this is true for both makespan and resources.

For both the Assault and Webber Island scenarios, and to some degree the synthetic scenarios, unexpectedly good performance was observed when $\alpha = 1$, especially for resources. In this case no cost ranking was performed, although minimising the probability of failure was still important because failure had a fixed cost of 1000. One possibility is that over-separating the cost components may have adverse numerical effects. Small but interesting variations in cost can be lost during floating point calculations when costs are in the order of 1×10^9 and probabilities are small.

The source code and examples for this paper are currently available from <http://csl.anu.edu.au/~daa>.

Conclusions and Future Work

LRTDP, combined with good heuristics and sensible state storage, is a practical way to perform decision-theoretic planning under the general framework of concurrent tasks, resources, and conflicting costs.

Unfortunately no direct comparisons with other planning methods were possible because no existing planning packages handle all the characteristics of this domain. Work is underway extending existing planners, and developing alternative planning methodologies, for the probabilistic operations planning domain.

Current work is applying a probabilistic version of GraphPlan (Blum & Langford 1999) to generate better heuristic values. The advantage of GraphPlan is its ability to efficiently propagate the consequences of trying multiple actions in an admissible fashion, potentially directing the LRTDP search much more efficiently and greatly reducing the number states of states generated.

Using factored representations of values, such as algebraic decision diagrams (Hoey *et al.* 1999), or symbolic LAO* (Feng & Hansen 2002) could improve convergence and decrease the number of values that need to be stored.

A related approach might assign a simple planning agent to each task. Each agent has a simple job: it learns the optimal conditions under which its task should begin. The idea is to factor a complex policy into a group of simple agent policies. Multi-agent MDP methods provide the training tools (Tao, Baxter, & Weaver 2001).

A small change in task failure probability can have a large impact on the final policy. Unfortunately, it is hard to accurately predict the probability of failure of a task. Using a *possibilistic* Markov decision process could reduce the sensitivity of the policy to the initial task failure probabilities.

Acknowledgements

The authors thank the reviewers for their helpful suggestions. National ICT Australia is funded by the Australian Government's Department of Communications, Information Technology and the Arts, and the Australian Research Council through Backing Australia's Ability and the ICT Centre of Excellence program. This project was also funded by the Australian Defence Science and Technology Organisation.

References

- Atkins, E. M. 1999. *Plan Generation and Hard Real-Time Execution With Application to Safe, Autonomous Flight*. Ph.D. Dissertation, University of Michigan.
- Bacchus, F., and Ady, M. 2001. Planning with resources and concurrency: A forward chaining approach. In *IJCAI*, 417–424.
- Blum, A., and Langford, J. 1999. Probabilistic planning in the graphplan framework. In *ECP*, 319–332.
- Blythe, J. 1999. Decision-theoretic planning. *AI Magazine* 1(20).
- Bonet, B., and Geffner, H. 2003. Labeled RTDP: Improving the convergence of real-time dynamic programming. In *Proc. ICAPS*.
- Do, M. B., and Kambhampati, S. 2002. Planning graph-based heuristics for cost-sensitive temporal planning. In *Proc. AIPS*.
- Feng, Z., and Hansen, E. A. 2002. Symbolic LAO* search for factored markov decision processes. In *AIPS Workshop on Planning via Model Checking*.
- Geibel, P. 2001. Reinforcement learning with bounded risk. In *Proc. ICML*, 162–169.
- Gerenini, A., and Serina, I. 2002. LPG: A planner based on local search for planning graphs with action costs. In *Proc. AIPS*.
- Hansen, E., and Zilberstein, S. 2001. Lao*: A heuristic search algorithm that finds solutions with loops. *Artificial Intelligence* 129:35–62.
- Haslum, P., and Geffner, H. 2001. Heuristic planning with time and resources. In *Proc. ECP*, 121–132.
- Hoey, J.; St-Aubin, R.; Hu, A.; and Boutilier, C. 1999. SPUDD: Stochastic planning using decision diagrams. In *Proc. ICML*, 279–288.
- Howard, R. A. 1960. *Dynamic Programming and Markov Processes*. Cambridge, MA.: MIT Press.
- Laskey, K. B.; D'Ambrosio, B.; Levitt, T. S.; and Mahoney, S. 2000. Limited rationality in action: Decision support for military situation assessment. *Minds and Machines* 10:53–77.
- Meuleau, N.; Hauskrecht, M.; Kim, K.-E.; Peshkin, L.; Kaelbling, L.; Dean, T.; and Boutilier, C. 1998. Solving very large weakly coupled markov decision processes. In *AAAI/IAAI*, 165–172.
- Mulvehill, A. M., and Caroli, J. A. 2000. JADE: A tool for rapid crisis action planning. In *5th International Command and Control Research and Technology Symposium*.
- Precup, D.; Sutton, R. S.; and Dasgupta, S. 2001. Off-policy temporal-difference learning with function approximation. In *Proc. ICML*, 417–424.
- Tao, N.; Baxter, J.; and Weaver, L. 2001. A multi-agent, policy-gradient approach to network routing. In *Proceedings of the Eighteenth International Conference on Machine Learning*, 553–560. Morgan Kaufmann.
- Tate, A.; Drabble, B.; and Dalton, J. 1996. *Advanced Planning Technology*. Menlo Park, CA: AAI Press. chapter O-Plan: A Knowledge-Based Planner and its Application to Logistics.
- Thuve, H. 2001. TOPFAS (tool for operational planning, force activation and simulation). In *6th International Command and Control Research and Technology Symposium*.
- Younes, H. L. S.; Musliner, D. J.; and Simmons, R. G. 2003. A framework for planning in continuous-time stochastic domains. In *Proc. ICAPS*, 195–204.
- Zhang, L.; Kristensen, L. M.; Janczura, C.; Gallasch, G.; and Billington, J. 2002. A Coloured Petri Net based tool for course of action development and analysis. In *Workshop on Formal Methods Applied to Defence Systems*, volume 12. Australian Computer Society.