Planning in a Multi-Agent Environment: Theory and Practice

Jürgen Dix  
Univ. of Manchester  
Oxford Road  
Manchester M13 9PL  
UK  
dix@cs.man.ac.uk

Héctor Muñoz-Avila  
Dept. of Computer Science and Engineering  
Lehigh University  
Bethlehem, PA 18015  
USA  
umoz@cse.lehigh.edu

Dana S. Nau  
Dept. of Computer Science and Inst. for System Research  
University of Maryland  
College Park, MD 20742  
USA  
nau@cs.umd.edu

Lingling Zhang  
Dept. of Computer Science  
University of Maryland  
College Park, MD 20742  
USA  
lzhang@cs.umd.edu

Abstract

We give the theoretical foundations and empirical evaluation of a planning agent, shop, performing HTN planning in a multi-agent environment. shop is based on A-SHOP, an agentized version of the original SHOP HTN planning algorithm, and is integrated in the IMPACT multi-agent environment. We ran several experiments involving accessing various distributed, heterogeneous information sources, based on simplified versions of noncombatant evacuation operations, NEO’s. As a result, we noticed that in such realistic settings the time spent on communication (including network time) is orders of magnitude higher than the actual inference process. This has important consequences for optimizations of such planners. Our main results are: (1) using NEO’s as new, more realistic benchmarks for planners acting in an agent environment, and (2) a memoization mechanism implemented on top of shop, which improves the overall performance a lot.

1. Introduction

Planning a course of action is difficult, especially for large military organizations (e.g., the U.S. Navy) that have their available assets distributed world-wide. Formulating a plan in this context requires accessing remote, heterogeneous information sources. For example, when planning for a Noncombatant evacuation operation, denoted by NEO, military commanders must access several information sources including: assets available in the zone of operations, Intelligence assessment about potential hostiles, weather conditions and so forth.

A-SHOP is an HTN planning algorithm for planning in a multi-agent environment. A-SHOP can interact with external information sources, frequently heterogeneous and not necessarily centralized, via the IMPACT multi-agent environment. The IMPACT project (see [ESP99; SBD+00]) and http://www.cs.umd.edu/projects/impact/) aims at developing a powerful and flexible, yet easy to handle framework for the interoperability of distributed heterogeneous sources of information.

In previous work we described the definition of the A-SHOP planning algorithm, an agentized version of SHOP that runs in the IMPACT environment and formulated the conditions needed for A-SHOP to be sound and complete (NN02).

In this paper we will focus on the actual implementation of A-SHOP following the principles stated in our previous work and experiments we did on a transportation domain for NEO operations. Our analysis of the initial runs of A-SHOP revealed that most of the running time was spent on communication between the IMPACT agents and accessing the information sources. Compared to that, the actual inferencing time in A-SHOP was very small. Furthermore, we observed that frequently the same IMPACT query was performed several times. To solve this problem we implemented a memoization mechanism to avoid repeating the same IMPACT queries. As we will show, the key for this mechanism to work is that the A-SHOP algorithm forms a planning technique called ordered task decomposition. As a result, A-SHOP maintains partial information about the state of the world. Experiments performed show that the memoization mechanism results in a significant reduction of the running time in A-SHOP. This reduction depends on the overall network time spent to access the information sources: the higher this network time, the higher is the gain obtained by our memoization technique.

This paper is organized as follows. The next section describes the Noncombatant evacuation operations (NEO’s) planning domain, which partly motivated our approach.

In Section 3 we introduce IMPACT, define A-SHOP and the results establishing the soundness and completeness of A-SHOP. Section 4 describes the actual implementation of A-SHOP. Section 5 describes the memoization mechanism and its dependence on the Ordered Task Decomposition planning technique. In Section 6 we describe several experiments with A-SHOP for logistics NEO problems. Finally, we discuss related work in Section 7 and conclude with Section 8.

2. Planning Noncombatant Evacuation Operations (NEO’s)

Noncombatant evacuation operations are conducted to assist the U.S.A. Department of State (DOS) with evacuating noncombatants, nonessential military personnel, selected host-nation citizens, and third country nationals whose lives are in danger from locations in a host foreign nation to an appropriate safe haven. They usually involve the swift insertion of a force, temporary
occupation of an objective (e.g., an embassy), and a planned withdrawal after mission completion. NEO’s are often planned and executed by a Joint Task Force (JTF), a hierarchical multi-service military organization, and conducted under an American Ambassador’s authority.

The decision making process for a NEO is conducted at three increasingly-specific levels: strategic, operational and tactical. The strategic level involves global and political considerations such as whether to perform the NEO. The operational level involves considerations such as determining the size and composition of its execution force. The tactical level is the concrete level, which assigns specific resources to specific tasks. Thus, this domain is particularly suitable for a hierarchical (HTN) planning approach.

JTF commanders plan NEO’s by gathering information from multiple sources. For example, in preparation for Operation Eastern Exit (Mogadishu, Somalia, 1991), commanders accessed Intelligence Satellite Photographs from NIMA (National Imagery and Mapping Agency), intelligence assessment information from the CIA, the Emergency Action Plan (EAP) from the US Embassy in Mogadishu, among others (Sie91). Any automated system planning in this domain must be able to access these multiple distributed information sources.

3. Planning with Remote, Heterogeneous Information Sources

In this section we review results obtained in (NN02). After giving a brief overview on SHOP and IMPACT, we state the main results of (NN02).

SHOP

Rather than giving a detailed description of the kind of HTN planning used by SHOP ((NCLMA99)), we consider the following example.

In order to do planning in a given planning domain, SHOP needs to be given knowledge about that domain. SHOP’s knowledge base contains operators and methods. Each operator is a description of what needs to be done to accomplish some primitive task, and each method is a prescription for how to decompose some complex task into a totally ordered sequence of subtasks, along with various restrictions that must be satisfied in order for the method to be applicable.

Given the next task to accomplish, SHOP chooses an applicable method, instantiates it to decompose the task into subtasks, and then chooses and instantiates other methods to decompose the subtasks even further. If the constraints on the subtasks prevent the plan from being feasible, SHOP will backtrack and try other methods.

As an example, Figure 1 shows two methods for the task of selecting a helicopter launching base: establishing the base within flying distance, and launch from carrier battle group (i.e., use the carrier as the helicopter launching base). Note that each method’s preconditions are not used to create subgoals (as would be done in action-based planning). Rather, they are used to determine whether or not the method is applicable. Establishing the base within flying distance requires to have transport helicopters and a security force available. Launching from carrier battle group also requires to have helicopters available and those helicopters have to have air refuelling capability (which wasn’t necessary in the first method because the helicopters are within flying distance).

If the method establishing base within flying distance method is selected, the select helicopter launching base is decomposed into three subtasks: transport security force (F) using the helicopters (H) to the selected launching base (A), position the security force in the base, and transport the fuel to the base. Some of these tasks, such as transporting the security force, can be further decomposed. Others such as position security force cannot. The former are called compound tasks, the latter primitive tasks.

IMPACT

To get a bird’s eye view of IMPACT, here are the most important features:

Actions: Each IMPACT agent has certain actions available. Agents act in their environment according to their agent program and a well defined semantics determining which of the actions the agent should execute.

Legacy Code: IMPACT Agents are built on top of arbitrary software code (Legacy Data).

Agentization: A methodology for transforming legacy code into an agent has been developed.

For example, in many applications a math agent is needed. This agent is able to do mathematical calculations shipped to it by other agents. For example it can determine the time it takes for a particular vehicle to get from one location to another. Another example is a monitoring agent, that keeps track of distances between two given points and the authorized range or capacity of certain vehicles. These information can be stored in several databases.

The Code Call Machinery To perform logical reasoning on top of third party data structures (which are part of the agent’s state) and code, the agent must have a language within which it can reason about the agent state. We therefore introduce the concept of a code call atom, which is the basic syntactic object used to access multiple heterogeneous data sources.

A code call executes an API function and returns as output a set of objects of the appropri-
ate output type. Going back to our agent introduced above, monitoring may be able to execute the cc monitoring: distance(locFrom, locTo). The math agent may want to execute the following code call: math: computeTime(cargoPlane, locFrom, locTo).

What we really need to know is if the result of evaluating such code calls is contained in a certain set or not. To do this, we introduce code call atoms. These are logical atoms that are layered on top of code calls. They are defined through the following inductive definition.

**Definition 1 (Code Call Atoms (in(\(X, cc\))):** If cc is a code call, and \(X\) is either a variable symbol, or an object of the output type of cc. then \(in(X, cc)\) and \(not\_in(X, cc)\) are code call atoms. \(not\_in(X, cc)\) succeeds if \(X\) is not in the set of objects returned by the code call cc.

Code call atoms, when evaluated, return boolean values, and thus may be thought of as special types of logical atoms. Intuitively, a code call atom of the form \(in(X, cc)\) succeeds if \(X\) can be set to a pointer to one of the objects in the set of objects returned by executing the code call.

As an example, the following code call atom tells us that the particular plane "f22" is available as a cargo plane at ISB1: \(in(f22, transport\_Authority: cargo\_Plane(ISB1))\). Often, the results of evaluating code calls give us back certain values that we can compare. Based on such comparisons, certain actions might be fired or not. To this end, we need to define code call conditions. Intuitively, a code call condition is a conjunction of code call atoms, equalities, and inequalities. Equalities, and inequalities can be seen as additional syntax that "links" together variables occurring in the atomic code calls.

**Definition 2 (Code Call Conditions (ccc)):**
1. Every code call atom is a code call condition.
2. If \(s, t\) are either variables or objects, then \(s = t\) is a code call condition.
3. \(s < t, s > t, s \geq t, s \leq t\) are code call conditions.
4. If \(X_1, X_2\) are code call conditions, then \(X_1 \& X_2\) is a code call condition.

A code call condition satisfying any of the first three criteria above is an atomic code call condition.

**IMPACTING SHOP**

A comparison between IMPACT's actions and SHOP's methods shows that IMPACT actions correspond to fully instantiated methods, i.e. no subtasks. While SHOP's methods and operators are based on STRIPS, the first step is to modify the atoms in SHOP's pre- and effects, so that SHOP's preconditions will be evaluated by IMPACT's code call mechanism and the effects will change the state of the IMPACT agents. This is a fundamental change in the representation of SHOP. In particular, it requires replacing SHOP's methods and operators with agentized methods and operators. These are defined as follows.

**Definition 3 (Agentized Meth.: (AgentMeth \(h \_\chi t\)):** An agentized method is an expression (AgentMeth \(h \_\chi t\)) where \(h\) (the method's head) is a compound task, \(\chi\) (the method's preconditions) is a code call condition and \(t\) is a totally ordered list of subtasks, called the task list.

The primary difference between definition of an agentized method and the definition of a method in SHOP is as follows. In SHOP, preconditions were logical atoms, and SHOP would infer these preconditions from its current state of the world using Horn-clause inference. In contrast, the preconditions in an agentized method are IMPACT's code call conditions rather than logical atoms. A-SHOP (the agentized version of SHOP defined in the next section) does not use Horn-clause inference to establish these preconditions but instead simply invokes those code calls, which are calls to other agents (which may be Horn-clause theorem provers or may instead be something entirely different).

**Definition 4 (Agentized Op.: (AgentOp \(h \_\chi_{add} \_\chi_{del}\)):** An agentized operator is an expression (AgentOp \(h \_\chi_{add} \_\chi_{del}\)), where \(h\) (the head) is a primitive task and \(\chi_{add}\) and \(\chi_{del}\) are lists of code calls (called the add- and delete-lists). The set of variables in the tasks in \(\chi_{add}\) and \(\chi_{del}\) is a subset of the set of variables in \(h\).

**The Algorithm**

```plaintext
procedure A-SHOP(t, D)
1. if \(t = nil\) then return \(nil\)
2. \(t :=\) the first task in \(t\); \(R :=\) the remaining tasks
3. if \(t\) is primitive and a simple plan for \(t\) exists then
4. \(q :=\) simplePlan\((t)\)
5. return concatenate\((q, A-SHOP(R, D))\)
6. else if \(t\) is non-prim. \(\land\) there is a reduction of \(t\) then
7. nondeterministically choose a reduction:
   Non-deterministically choose an agentized method, (AgentMeth \(h \_\chi t\)), with \(\mu\) the most general unifier of \(h\) and \(t\) and substitution \(\theta\) s.t.
   \(\chi_{add}\) is ground and holds in IMPACT's state \(O\).
8. return A-SHOP(concatenate\((t_{\delta}, R), D)\)
9. else return \(FAIL\)
10. end if
end A-SHOP
```

**Figure 2: A-SHOP**, the agentized version of SHOP. The A-SHOP algorithm is now an easy adaptation of the original SHOP algorithm. Unlike SHOP (which would apply an operator by directly inserting and deleting atoms from an internally-maintained state of the world), A-SHOP needs to reason about how the code calls in an operator will affect the states of other agents. One might think the simplest way to do this would be...
simply to tell these agents to execute the code calls and then observe the results, but this would not work correctly. Once the planning process has ended successfully, A-SHOP will return a plan whose operators can be applied to modify the states of the other IMPACT agents—but A-SHOP should not change the states of those agents during its planning process because this would prevent A-SHOP from backtracking and trying other operators.

Thus in Step 12, SHOP does not issue code calls to the other agents directly, but instead communicates to a monitoring agent. The monitoring agent keeps track of all operators that are supposed to be applied, without actually modifying the states of the other IMPACT agents. When A-SHOP queries for a code call \( cc = \mathcal{S} : f(a_1, \ldots, a_n) \) in \( \chi \) to evaluate a method’s pre-condition (Step 7), the monitoring agent examines if \( cc \) has been affected by the intended modifications of the operators and, if so, it evaluates \( cc \). If \( cc \) is not affected by application of operations, IMPACT evaluates \( cc \) (i.e., by accessing \( \mathcal{S} \)). The list of operators maintained by the monitoring agent is reset whenever a planning process begins. The apply function applies the operators and creates copies of the state of the world. Depending on the underlying software code, these changes might be easily reversible or not. In the latter case, the monitoring agent has to keep track of the old state of the world.

**Finite Evaluability of ccc’s and Completeness of ASHOP**

We have introduced syntactic conditions, similar to safety (and consequently called strong safety) in classical databases, to ensure evaluability and termination of ccc’s (see (ESR00; SBD’00)).

**Theorem 1 (Soundness, Completeness)** Let \( \mathcal{O} \) be a state and \( \mathcal{D} \) be a collection of agentized methods and operators. If all the preconditions in the agentized methods and add and delete-lists in the agentized operators are strongly safe wrt. the respective variables in the heads, then A-SHOP is correct and complete.

4. ASHOP: Implementation

Each cycle in the A-SHOP algorithm consist of three phases (see lines 3 and 7 of Figure):

1. **Selection Phase**: Selecting a candidate agentized method or operator to reduce a task.
2. **Evaluation Phase**: Evaluating the applicability of the chosen agentized method or operator.
3. **Reduction Phase**: Performing the agentized method or operator.

To accomplish these phases we have implemented 3 IMPACT agents which perform pieces of these phases:

- **ashop**: This is the agent that all IMPACT agents communicate with for generating a plan. It receives as input a problem and outputs a solution plan. The A-SHOP agent also performs the Selection Phase and the evaluation phase for the situation in which an operator is chosen. The operator is then send to the Monitor Agent, to perform a virtual execution of it. If the selection of a method is made, the A-SHOP agent sends a message to the Preconditions Agent with the code-call condition of the selected method.

- **preconditions**: Receives a code-call condition and evaluates each code-call by sending it to the Monitoring Agent.

- **monitoring**: The monitor agent has two functions: firstly, it receives a operator and performs a virtual execution of it. Secondly, it receives code-calls and evaluates them. We explain both of these operations in detail below as they are closely inter-related.

One of the main issues we are confronted with during the implementation is how to cope with the execution of agentized operators. In classical AI planning, where the state is centralized, executing an operator is a matter of simply making the changes to the state indicated by the operator and keeping track of those changes in an stack; if backtracking occurs, the stack is used to restore to the previous state.

This approach is not working in a multi-agent environment, where the state is distributed among several information sources. Firstly, remote information sources might not be able to backtrack to a previous state. Secondly, even if backtracking was possible, performing such an operation may be costly. Thirdly, executing an operation may make resources unavailable temporarily for other agents and if backtracking takes place, these resources could have been used. For example, an operator may reserve a recon plane but a later operator trying to provide flight escort to the recon plane might not succeed. In this case the original recon plane should have not been reserved in the first place.

The Monitoring Agent overcomes these problems by keeping track of each operator execution without accessing the corresponding information sources to request an execution of the operation. For this reason we refer to this as a virtual operator execution. Since monitoring keeps track of the changes in the states of the remote information sources, the preconditions sends the code-calls to the monitoring. monitoring makes the code-call to the corresponding information source and then checks if the answer is affected by the previously virtually executed operators before sending its answer to the preconditions.

5. Memoization in ASHOP

While our implementation secures that the produced plans are consistent, the resulting running time was large compared to the inferencing time (we will describe the experiments later). Our experiments show that the bulk of the planning time has been spent in accessing the remote information sources. Further analysis revealed that the same code-calls were repeatedly being executed during the planning process. Our solution was to implement a cache mechanism to avoid repeated evaluations of the same code call in IMPACT.
After receiving the answer from SHOP, for example, we use a hash table to quickly check the validity of a condition in the current state. Other planning systems use more sophisticated data structures to reduce the time for evaluating a condition in the current state. For example, TLPlan, the winner of the 2000 AI planning competition, uses a bit map that allows checking conditions in almost constant time (Bac01).

Obviously none of these techniques would be useful here since the information sources are remote and SHOP has no control over how data is stored there and how it is updated. However, implementing a memoization mechanism turned out to be adequate for SHOP for two reasons: Firstly, A-SHOP performs Ordered Task Decomposition. Secondly, all access to the information sources is canalized through monitoring.

The fact that access to the information sources is canalized through monitoring makes this agent the natural place for maintaining the updated partial state of the world. As a result, we modified monitoring:

- When it receives a code-call from preconditions, the monitoring will first check if the code-call can be answered based on previous code-calls and the modifications indicated by the virtually executed operators. Only if it is not possible to answer this code call, the remote information source is accessed via the IMPACT code-call evaluation mechanism.
- After receiving the answer from IMPACT for the evaluation of the code-call, monitoring records this answer.

In the example of the recon plane, after the first operator reserving the recon plane is virtually executed, monitoring knows that there are no more recon planes available. Thus, as it receives the code-call enquiring about the availability of recon planes it will answer that this code-call cannot be satisfied without having to access the corresponding remote information source via IMPACT. As will be shown next, these changes resulted in a reduction of the running time.

### 6. Empirical Evaluation

The test domain is a simple transportation planning for a NEO (MAAN+01). Its plans involve performing a rescue mission where troops are grouped and transported between an initial location (the assembly point) and the NEO site (where the evacuees are located). After the troops arrived at the NEO site, evacuees are re-located to a safe haven.

Planning involves selecting possible pre-defined routes, consisting of four or more segments each. The planner must also choose a transportation mode for each segment. In addition, other conditions were determined during planning such as whether communication exists with State Department personnel and the type of evacuee registration process. A-SHOP’s knowledge base included six agentized operators and 22 agentized methods. There were four IMPACT information sources available:

- Transport Authority: Maintains information about the transportation assets available at different locations.
- Weather Authority: Maintains information about the weather conditions at the different locations.
- Airport Authority: Maintains information about availability and conditions of airports at different locations.
- Math Agent: math evaluates arithmetic expressions.

Typical evaluations include the subtract a certain number of assets use for an operation and update time delays.

The top level task for each problem in this experiment was the following: to perform a troop insertion and evacuees extraction plan. This top level task is decomposed into several subtasks, one for each segment in the route that the troops must cover (these segments are pre-determined as part of the problem description). Within each segment, A-SHOP must plan for the means of transportation (planes, helicopters, vehicles etc.) to be used and select a route for that segment. The selection of the means of transportation depends on their availability for that segment, the weather conditions, and, in the case of airplanes, on the availability and conditions of airports. The selection of the route depends on the transportation vehicle used and may lead to backtracking. For example, the choice of ground transportation assets needs to be revised if no roads are available or they are blocked, or too risky to take.

We ran our experiments on 30 problems of increasing size. The first five problems had four segments passing over five locations (including a particular location known as the Intermediate Staging Base ISB), the next five problems had five segments passing over six locations (two ISB’s), and so forth until the Problems 26–30 which had nine segments passing over 10 locations (five ISB’s).

We ran shop in two modes: with and without the memoization mechanism and measured for each mode two variables: inferencing time and total time. The inferencing time includes the time spent in the three agents implementing the A-SHOP algorithm. Thus, the difference between the total time and the running time indicates the sum of the communication time needed by IMPACT to access the remote information sources and of the time needed by the information sources to compute the answers to the queries.

Figure 3 shows the results of the experiments. Not surprisingly the inferencing times with and without memoization are almost identical. More interesting is the fact that the inferencing time is only a fraction of the overall running time. In addition, the use of the memoization mechanism results in a decrease in the running time of more than 30%.

### 7. Related Work

Most AI planning systems are unable to evaluate numeric conditions at all. A few can evaluate numeric conditions using attached procedures (e.g., SIPE (Wil88), O-Plan (CT91), TLPlan (BK00) and SHOP (NCLMA99)), but the lack of a formal semantics for
these attached procedures makes it more difficult to
guarantee soundness and completeness. Integer Pro-
gramming (IP) models appear to have excellent poten-
tial as a uniform formalism for reasoning about com-
plex numeric and symbolic constraints during plan-
ing, and some work is already being done on the use
of IP for reasoning about resources (Köh98; KW99;
WW99). However, that work is still work in progress,
and a number of fundamental problems still remain to
be solved.

Approaches for planning with external information
sources typically have in common that the informa-
tion extracted from the external information sources
is introduced in the planning system through built-in
predicates (EWD92; GEW94; Kno96; FW97). For ex-
ample, a modified version of UCPOP uses information
gathering goals to extract information from the external
information sources (Kno96). The information gather-
ing goals are used as preconditions of the operators.
The primary difficulty with this approach is that since
it is not clear what the semantic of the built-in predi-
cates is, this makes it difficult to guarantee soundness
and completeness.

8. Conclusion

The original motivation of our work was to make HTN
planning available in a multi-agent environment. This
is beneficial for both, planners (they gain access to
distributed and heterogenous information sources for
free and can ship various tasks to other agents) as well
as agent systems (which usually do not have available
planning components that are highly sophisticated and
efficient).

After developing the theory and implementing it, we
ran experiments on a simplified version of the NEO
domain, where data needed for the planning process is
distributed and highly heterogenous. In such a situa-
tion, data changes dynamically, eg. weather conditions
or available resources. Thus the available data can not
be stored locally, because of the sheer amount and the
dynamic changes in the database.

Our experiments revealed clearly that most of the
time is spent on communication with the information
sources and therefore network time. Thus improving
the actual planning algorithm (as done by most plan-
ners that assume all info is there locally) does not pay
off: the amount gained is orders of magnitude less than
the overall time. We really need caching mechanisms, to
avoid computing the same results over and over again.
In the extreme case, when caching is just storing every-
thing locally, we would end up with our original local
planner. This is not feasible because of the amount of
data involved and the fact that it changes dynamically.
The other extreme is not to do any caching at all. Our
memoization technique seems to be a good compromise
between these two extremes. The decrease in time we
are getting depends on the overall network time spent to
access the information sources: the higher this network
time, the higher is the gain obtained by our memoiza-
tion technique. Consequently our experiments showed
an overall gain ranging from 20%-40%.

References


