A Java API for Planning in Time-Limited Domains

Lavindra de Silva, Anthony Dekker (DSTO), James Harland

Part of DSTO-RMIT Project on Planning
(1\textsuperscript{st} March – 30\textsuperscript{th} June, 2006)
Introduction

• DSTO-RMIT project consisted of 4 milestones

1. Extend CANPLAN semantics (AAMAS `06) so that agents can plan up to a limited depth (1 month)

3. Survey existing planners and suitability for integration with DSTO’s CAVALIER tool (1 month)

5. Create a Java API for planning in time-limited domains (1 month)

7. Empirically evaluate the benefits of using the API with CAVALIER (1 month)
Background

• First Principles Planning
  – Initial state
  – Set of operators
  – Goal state

• Planning involves finding a sequence of operators that can take the agent from the initial state to the goal state

• The result of planning is a sequence of ground actions

• Planning fails if no such sequence can be found
Example (Blocks World)

Operators

stack(\textit{?blockA}, \textit{?blockB})
\hspace{1cm} \textbf{pre} \hspace{0.2cm} \text{clear(\textit{?blockA}) \land clear(\textit{?blockB})}
\hspace{1cm} \textbf{post} \hspace{0.2cm} \text{on(\textit{?blockA}, \textit{?blockB})}

unstack(\textit{?blockA}, \textit{?blockB})
\hspace{1cm} \textbf{pre} \hspace{0.2cm} \text{clear(\textit{?blockA}) \land on(\textit{?blockA}, \textit{?blockB})}
\hspace{1cm} \textbf{post} \hspace{0.2cm} \text{on-table(\textit{?blockA})}

Initial State

on-table(\textit{b1}) \land on(\textit{b3,b1}) \land
on-table(\textit{b2}) \land clear(\textit{b2}) \land clear(\textit{b3})

Goal State

on(\textit{b1,b2}) \land on(\textit{b3,b1})

Solution

unstack(\textit{b3,b1}) \rightarrow stack(\textit{b1,b2}) \rightarrow stack(\textit{b3,b1})
Example (State-Space Search)

Initial State

Goal State
Objective

• Find an existing first principles planner, modify it to do time-limited planning, and provide a Java interface to it

• Some desired properties of the planner:
  – fast (based on ICAPS planning competition results)
  – documented
  – free 😊
  – Java based
  – allow numerical calculations within operators
    • important in real agent systems
Metric-FF (*Hoffmann* `03)

- Best match: *Metric-FF* – a popular First Principles planner, based on *FF* (*Fast Forward – Hoffmann & Nebel* `01)
- Some of the fastest planners are based on Metric-FF and FF
- Based on Best-First (Heuristic) Search
- Metric-FF extends FF to allow numerical calculations within operators
- E.g. moveRight(?person)
  
  ```
  pre (\textless \text{location(?person)} \text{-} \text{location(wall)} 1))
  post (+ \text{location(?person)} 1)
  ```
Extensions to Metric-FF

- Metric-FF was interfaced with Java, using the Java Native Interface (JNI)

- The **Best First Search** (BFS) search strategy was modified to stop search on reaching a user-defined *step-limit*, or to plan up to a maximum *depth-limit*
  
  - if no solution is found, most recent (incomplete) plan found is returned

- The planner was modified to read the initial state, goal state and set of operators as *strings* instead of files

- Extra memory management had to be added to the planner to explicitly free all global variables and static variables on completion
Interfacing via JNI

- Interfacing via JNI allowed Metric-FF to be used in the following manner:

  ```java
  MetricFFInterface met_ff_int = new MetricFFInterface();

  Vector planStrings = met_ff_int.callMetricFF (
      maxSteps,
      maxDepth,
      pddlDomain,
      pddlProblem);
  ```

- The Vector returned contains the solution
  - each element is an operator, along with variable bindings
  - element at position zero is first operator, position two is second ...
Modifications to BFS

• Modifications were made to allow the maximum depth of the search-tree to be limited by a user-specified value
  – the structure of an action (in the implementation) now contains an extra variable representing the instance’s depth in the search tree
  – if the user-specified depth-limit is reached, continue planning by backtracking to a smaller depth

• Modifications were made to allow total number of “planning steps” to be limited by a user-specified value
  – keep track of the total number of (successfully tried) action instances
  – if the user-specified step-limit is reached, return current plan
Reading Strings not Files

• Reading from files can be slow

• Better to provide the planner with strings, representing the initial state, goal state and operators
  – the appropriate set of operators are determined at runtime

• This was done by modifying bison (parser) generated C code to read input from strings instead of files
  – replace `fread()` and `getc()` with similar functions that read strings
Memory Management

• Using JNI on Windows meant a stand-alone DLL file needed to be created for the C source files

• However after loading a DLL file from within a Java application for use via JNI, the DLL cannot be forcefully unloaded

• This means the same DLL is used multiple times with same values in global/static variables left over from the previous call to the DLL

• This sometimes leads to Windows running out of memory
  – because of memory leaks in re-initialising global/static variables

• Therefore Metric-FF was modified to be able to store all `malloc()`, `calloc()` and `realloc()` calls done for global/static variables, and to free all such variables at the end of planning
  – linked list used to store pointers to all memory allocations
  – makes the planner a bit slower!
Discussion/Issues

• Metric-FF turned out to be faster than a basic hand-coded planner, despite the overheads, i.e.:
  – requiring a conversion of CAVALIER’s Java objects into strings that represent the initial state, final state and set of operators
  – re-converting (memory allocation/de-allocation) of strings into C variables for planning

• However, Metric-FF ran out of memory when the initial state was too large (e.g. 32x32 grid with many objects & obstacles on the grid), whereas the hand-coded planner didn’t
  – number of calls that can be made to calloc/malloc/realloc is limited
  – hand coded planner didn’t have to allocate memory for representing the initial state

• There were also pre-processing/initialisation steps (different to “planning steps”) that we needed to take into account
References

