Teaching Forward-Chaining Planning with JAVAFF

Andrew Coles, Maria Fox, Derek Long and Amanda Smith
Department of Computer and Information Sciences,
University of Strathclyde, Glasgow, G1 1XH, UK
email: firstname.lastname@cis.strath.ac.uk

Abstract

In this paper we present the results of our work looking at how to provide a hands-on learning experience in AI planning to an undergraduate audience, complementing a conventional lecture series. At the core of our approach is a new Java implementation of the well-known planner FF, JAVAFF. By using object-oriented programming techniques, JAVAFF serves as a flexible and comprehensible substrate for student planning exercises. As a proof-of-concept, we present the exercises we constructed for use with the planning part of a final-year undergraduate AI module. The exercises involve making modifications to JAVAFF to implement given research ideas, and culminate in a group-based 'planning competition'. We also detail the new directions in which we are taking JAVAFF, in anticipation of the next academic year.

1 Introduction

When designing a curriculum for an undergraduate AI course, one key challenge is how to construct practical exercises to accompany the taught material, bearing in mind the abilities of the students and the amount of time available for the practical work. Considering exercises in AI planning, several possibilities immediately emerge as being viable, and have been used in past AI courses:

- Supply a natural-language description of a planning problem to the students, and set the task of writing an accurate PDDL2.1 domain model (Fox & Long 2003) and a problem file generator.
- Provide a collection of benchmark domains and planners, giving the students experience in using planners as problem-solving tools. This can be accompanied with data collection and analysis tasks, asking students to compare the performance of a selection of planners across a given selection of domains.
- Set the task of manually drawing a planning graph (Blum & Furst 1995) for a small example problem, drawing the action and fact layers and the necessary mutexes (omitted if a relaxed planning graph is to be drawn).

Whilst these tasks have good potential as part of a toolkit of practical exercises in planning, they do not always serve to best complement the emphasis of the lectures. In the first two of these, for instance, by treating planners as 'black-box' problem solving tools, our practical exercises are directly at odds with the lectures, where the emphasis is on *how* planners work in terms of algorithms, heuristics, search paradigms, and so on. The ideal towards which we strive is to provide practicals which support and reinforce the materials covered in the lectures, encouraging a deeper understanding of planners, to complement practicals looking at other areas such as domain modelling.

Due to the scale of modern planning systems, it is infeasible to expect students to implement a whole planner: the task of writing even just a parser, or implementing one of the popular heuristics (e.g. FF's Relaxed Planning Graph (RPG) heuristic) (Hoffmann & Nebel 2001a) is beyond the scope of an undergraduate degree module. What is practical, however, is to focus on a subset of the implementation of a planner, and use this as a substrate for practical exercises. In this work, we present JAVAFF, an implementation of FF (Hoffmann & Nebel 2001a) written in Java, a language with which many students are familiar. By employing object-oriented programming techniques to encapsulate the more complex areas of code, JAVAFF serves as an intuitive basis for student planner development.

There are several possible uses for JAVAFF, ranging from small exercises to large projects, looking at many or fewer areas of the planner: different state-space search algorithms; modifying the heuristic; exploring optimisation approaches; and many others. In this paper, we will describe the practical exercises we devised for use in our teaching, with the focus being on the development and evaluation of search algorithms for forward-chaining planning. These will be followed with some remarks on the exciting directions in which we are now taking JAVAFF, in preparation for a new taught postgraduate MRes course in 'Automated Planning for Autonomous Systems', to begin in September 2008.

2 JavaFF: An Overview

JAVAFF is a Java implementation of the basic structure of Hoffmann's FF (Hoffmann & Nebel 2001a), based on the source code of CRIKEY[1] (Coles *et al.* 2008). In JAVAFF, as in FF, state-space forward-chaining search is guided by the RPG heuristic, with a two-phase planning process. First, an

[1] http://planning.cis.strath.ac.uk/CRIKEY/

attempt is made to find a plan using Enforced Hill Climbing (EHC) and considering only the 'helpful actions' at each state. If this fails to find a plan, best-first search is performed considering all applicable actions in each state. To allow the code presented to students to remain as clean as possible, we omitted features from the more recent versions of FF, such as goal ordering, and focussed on the feature set of FF v1.0. As such, JAVAFF is not a full rational reconstruction, but preserves the essence of FF and the clarity we afford through these restrictions allows it to serve as a useful teaching tool.

To aid student understanding, a tutorial booklet was prepared to accompany the code, describing the classes relevant to the practical exercises we devised. For reasons of space, we are unable to reproduce the detailed code descriptions from the booklet here in their entirety, or to go beyond those and explore other areas of the code. To briefly illustrate the elegance of the JAVAFF implementation we shall however highlight two key methods. The source code for JAVAFF (including comments, again omitted here) and the accompanying booklet is available for download[2].

2.1 The Main Search Method

The main search method, presented in Figure 1, constitutes the outline of the main search process of JAVAFF, and is the point where we introduce students to the code. In a few lines of code, it captures the essence of the two-stage search process of FF:

1. The first stage of `performFFSearch`, a group of three lines, attempts to search for a solution plan using Enforced Hill Climbing, and only using the helpful actions.
2. If EHC with helpful actions fails (if `goalState == null`), best-first search is used. Here, not just the helpful actions are used: a `NullFilter` is used, which preserves all actions applicable in a state.

As can be seen, invoking best-first search is similar to invoking EHC: both implement a common interface, `Search`. The key difference is that a `NullFilter` is used rather than a `HelpfulFilter`, to lift the restriction of only considering helpful actions, but the `HelpfulFilter` could equally well be used with best-first search if so desired. The `Filter`, which both of these implement, provides an abstraction over the selection of which states S' to consider as successors to a state S. Exploiting an object-oriented framework in this manner provides much scope for practical experimentation by making only isolated changes, as we shall show in our practical exercises.

2.2 The EHC Search Method

The EHC search method, presented in Figure 2 and defined in the class `EnforcedHillClimbingSearch`, provides an implementation of FF's EHC algorithm. The EHC class has helper methods and member variables (referred to from the code in Figure 2) that are responsible for maintaining an open list and memoising visited states. Perhaps the most interesting thing to observe is that the `State` class provides several methods that allow the implementation of the algorithm to be very clear:

- the method `getNextStates` returns the set of successors to the state (reached through helpful actions or all applicable actions, depending on the filter passed);
- the method `goalReached` ascertains whether or not a state is a goal state: if it is, we have found a solution;
- the method `getHValue` returns the RPG heuristic value of the state.

Note that for `getHValue`, `State` objects transparently cache their RPG heuristic values and actions that are helpful in that `State`. In this manner, the performance of the planner is improved; and, again, through using object-oriented programming, the low-level details of this are encapsulated.

As an illustration of how a minor change can be demonstrated using JAVAFF, removing the 'break' inside the while loop in Figure 2 and adding an extra Boolean variable will change EHC to use steepest descent, rather than taking the first successor with a better heuristic value than the best seen so far.

3 Practical Exercises for use with JavaFF

As a novice-friendly introduction to forward-chaining planning, there are many possibilities for the use of JAVAFF in teaching. Here, we shall detail how we used it as part of our teaching of planning, within a varied final-year undergraduate AI module. The module, as a whole, was $1/8^{th}$ of the year's course load, and the planning exercises were allocated 60% of the practical work time within this. Our key four learning objectives for the exercises, **LO1–LO4**, were:

1. An understanding of how planning search algorithms can be presented in Java code, and how modifications can be made to explore alternative strategies;
2. The ability to perform basic empirical analyses on the performance data obtained by a planner, in the context of investigating the efficacy of new algorithms or in changes to the existing code;
3. The ability to implement pseudocode algorithm descriptions (from textbooks or papers);
4. Evidence of an intrinsic understanding of search, reflected in an open-ended task, with the three key indicators being (i) ingenuity, (ii) independent research, and (iii) the application of techniques covered in lectures.

As with the code, we encourage the reader to follow the footnote link to the course website to obtain a full copy of the practical materials; what follows is an abridged description of the practicals, relating them to our learning objectives, and indicating how JAVAFF allowed us to achieve these. We will then present a pedagogical evaluation drawn from the performance of the students and their feedback.

3.1 Beginning with JavaFF

The purpose of this practical exercise is to ensure the students have a working copy of JAVAFF; are comfortable with parts of the source code; and are able to run the planner, gather data, and analyse results obtained. The booklet begins with explaining how to download a tar.gz bundle containing the source code of JAVAFF and some IPC3

[2] `http://cis.strath.ac.uk/~ac/JavaFF`

```
public static State performFFSearch(State initialState) {

  EnforcedHillClimbingSearch EHCS = new EnforcedHillClimbingSearch(initialState);
  EHCS.setFilter(HelpfulFilter.getInstance());
  State goalState = EHCS.search();

  if (goalState == null) {
    BestFirstSearch BFS = new BestFirstSearch(initialState);
    BFS.setFilter(NullFilter.getInstance());
    goalState = BFS.search();
  }

  return goalState;
}
```

Figure 1: The Main Search Method, `performFFSearch`

```
public State search() {

  if (start.goalReached()) return start;

  needToVisit(start); // dummy call (adds start to the list of 'closed' states)
  open.add(start);
  bestHValue = start.getHValue();

  while (!open.isEmpty()) {
    State s = removeNext();
    Set<State> successors = s.getNextStates(filter.getActions(s));
    Iterator<State> succItr = successors.iterator();

    while (succItr.hasNext()) {
      State succ = succItr.next();
      if (needToVisit(succ)) {
        if (succ.goalReached()) {
          return succ;
        } else if (succ.getHValue().compareTo(bestHValue) < 0) {
          bestHValue = succ.getHValue();
          open = new LinkedList<State>();
          open.add(succ);
          break; // and skip looking at the other successors
        } else {
          open.add(succ); // otherwise, add to the open list
        }
      }
    }
  }
  return null;
}
```

Figure 2: The EHC Search Method

benchmark problems domains (Rovers, Driverlog and Depots) (Long & Fox 2003). To ensure the students are able to run the planner, their first task is to run JAVAFF on the first ten problems from two benchmark domains (Rovers and Driverlog) and to record the time taken to find a plan in each of these.

Once this task has been completed, an overview of the code is provided, covering the `performFFSearch` method (Figure 1) and the role of `Filters`. From here, the students are set two implementation tasks (supporting **LO1**):

1. Change `performFFSearch` so that the `NullFilter` is used in place of the `HelpfulFilter` when using EHC search. (This change was chosen to support students' reading of the paper *What Makes the Difference Between HSP and FF?* (Hoffmann & Nebel 2001b).)
2. Change `performFFSearch` so that search is performed in three stages: EHC with helpful action pruning; EHC without helpful action pruning; best-first search without helpful action pruning.

With each of these two configurations, the students gather data on the same benchmark problems used with the stock JAVAFF implementation, and conclude the practical by drawing comparisons between the performance of the three configurations tested. Drawing comparisons at this early stage was related to **LO2**: that students would have a grasp of planning research as an empirical discipline.

3.2 Local Search with Restarts

The main purpose of this practical exercise is to satisfy **LO3**. The students are set the task of implementing a stochastic hill-climbing search algorithm for use within JAVAFF, similar to that employed in HSP (Bonet & Geffner 1998). The booklet first takes the students through the JAVAFF implementation of EHC search (Figure 2) before presenting the pseudo-code of a new algorithm to be implemented (Algorithm 1). Once implemented, the exercise requires two further changes: the ability to specify a depth bound; and the use of restarts when the depth bound is reached, rather than aborting when no solution plan is returned.

Whilst the students could have prepared this algorithm in isolation, the use of JAVAFF allowed us to use the search algorithm within a planner rather than on a simple canonical search problem (e.g. a domain-specific 15-puzzle solver, with the Manhattan distance heuristic). Due to the substantial implementation task prescribed for this practical, an analysis phase was omitted, but we will return to this in the next practical.

3.3 Successor Selection and Filters

This practical continues the line of exploring the use of hill-climbing techniques in forward-chaining planning, following the ideas of the planner IDENTIDEM (Coles, Fox, & Smith 2007). First, the concept of an explicit successor selection function is introduced, using a new Java interface, (`SuccessorSelector`). As a starting point a `BestSuccessorSelector` is provided, which takes a set of `States` and returns the state with the best heuristic

Algorithm 1: Hill Climbing Algorithm

```
Data: I - initial state
Result: S - a goal state
S ← I;
while S do
    successors ← successor states to S;
    bestsuccessors ← ∅;
    bestheuristic ← ∞;
    foreach S' ∈ successors do
        if need to visit S' then
            if S' is a goal then return S;
            if h(S') < bestheuristic then
                bestheuristic ← h(S');
                bestsuccessors ← {S'};
            else if h(S') = bestheuristic then
                add S' to bestsuccessors;
    if bestsuccessors = ∅ then
        S ← null;
    else
        S ← random choice from bestsuccessors;
return failed
```

value (breaking ties randomly). The first practical task is to modify the search algorithm class constructed in the previous practical to be able to use a `SuccessorSelector` object to select successors.

At this point, the students now have a stochastic hill-climbing algorithm, which they implemented starting from pseudocode, and which supports interchangeable successor selectors and filters. Before making any further changes, the students are set the task of collecting data concerning the performance of the planner on the same benchmark problems used in Practical 1 (the first 10 problems from Driverlog and Rovers). After this, two implementation tasks were set, following further the ideas of IDENTIDEM:

1. Writing a new filter, `RandomThreeFilter`, based on a code outline provided in the booklet (**LO1**). This filter serves as a wrapper around the `HelpfulFilter`, and chooses just three of the helpful actions at random.
2. Writing `RouletteSuccessorSelector`, a new successor selector, that performs roulette selection over the successors based on their heuristic values (the fitness of a state S is $1/h(S)$). For this, students were provided with a link to the Wikipedia article[3], which provides an abstract description of the technique (**LO3**).

After these implementation tasks, the students collect data on two planner configurations and compare to the data obtained earlier in this practical. The first configuration employs the `RandomThreeFilter` rather than `HelpfulFilter`. The second employs `RouletteSuccessorSelector` rather than `BestSuccessorSelector`. Again, this empirical

[3] `http://en.wikipedia.org/wiki/Fitness_proportionate_selection`

enquiry reinforces the role of data analysis in planning research (**LO2**). Further, the object-orientated nature of JAVAFF allows us to cleanly interchange system components, reducing the overhead of setting up comparative tests and increasing the reliability of results: apart from the component changed, the remainder of the implementation remains the same.

3.4 The Planning Competition

The culmination of the practical exercises is set up as a competition, with the aim being to satisfy all four learning objectives. The inspiration for this comes from the International Planning Competition series, an important biennial event in the planning community's calendar. Our competition has two tracks: satisficing, the aim being to find a plan quickly; and quality, the evaluation criterion being the plan length (number of actions) after ten minutes. The students are divided into teams and submit a modified version of JAVAFF to each track, along with a short description of what changes they made to the code beyond the first three exercises. Some suggestions are made as to possible modifications that could be made (**LO1**), and a suggested reading list is provided (**LO3**), but the exercise is designed to be as open-ended as possible (**LO4**) and the students were encouraged to consider several possibilities and run their own empirical analyses before submitting their final team entry (**LO2**).

To evaluate the competition entries, the students planners are run on a selection of benchmark domains: two of the domains which were provided to the students at the start of the course; and a further, unseen, domain. This follows the process of this year's planning competition, where the planners entering the competition will be required to submit source code and an executable to be ran by the competition organisers on unseen problems. Once the data have been collected, and a winner has been decided for each track, the results and prizes are presented to students.

4 Pedagogical Evaluation

We shall evaluate our pedagogy in two stages: the individual performance of students on practical exercises 1 to 3, along with feedback on their learning experience; and the range of approaches taken in the competition entries received for exercise 4. Whilst exercise 4 was designed to assess all four of our learning outcomes, the scrutiny of the performance of individual students on the other practicals allows us to build a good overall picture. 16 students took the course in total, and student feedback was obtained through informal discussion at a weekly two-hour lab session.

4.1 Individual Evaluation

Exercise 1: All but one student received full marks: this was a positive early sign, as it indicated that the students were able to make at least small changes and perform empirical analyses. Student feedback at this stage indicated they were comfortable with the practical work, and felt that the JAVAFF code had been adequately explained.

Exercise 2: The first task (implementation of a pseudocode hill-climbing algorithm) was reported to be challenging, but the students were well motivated and the majority completed this task well; the most common error was taking the first joint-best successor without randomly tie-breaking between them. This was a pleasing outcome, as the ability to implement pseudocode is an important research skill, and modulo minor errors the students were able to do this within JAVAFF. Not all students had time to complete the remaining tasks in the exercise (implementation of a depth bound), so a model answer was provided at this point.

Exercise 3: This exercise was split into three tasks. The first (supporting `SuccessorSelector`s in their answer to exercise 2) was completed well by most students; the most common error was only passing the joint-best successors to the `SuccessorSelector` object, rather than all of them. The second task (finish the outline of the code for `RandomThreeFilter`) was better received, as the concept of a `Filter` was already well understood by this point.

The third task, implementing roulette selection, was designed to be the most challenging stage of this practical. No pseudo-code was provided, only a reference to the Wikipedia article and the hint to use $1/h(S)$ as the proportion of the roulette wheel to give to a state S. Nonetheless, nearly all the students submitted an answer, and most were correct. In discussions with students, it transpires the key obstacle to success was being short of time: the deadline for exercise 3 was close to that for their final-year project dissertations.

4.2 Group Evaluation: The Planning Competition

The competition submissions are perhaps the most important tool for evaluating our pedagogy. The exercise was designed to be as open-ended as possible: a few suggestions were made in the booklet as to which directions to pursue, but students were encouraged to read around the literature, experiment with their own ideas, and perform their own analyses to come up with what they considered to be effective search strategies. Looking at the wide-range of approaches submitted by the four teams, we consider our practical series to be a resounding success. Some ideas submitted to the competition include:

An extension of hill climbing (satisficing track), where the states along the path to the current local minimum were maintained in a set P. Then, when expanding a state S and obtaining a successor set S', roulette selection was used to choose from the set $P \cup S'$. In doing so, the algorithm employs stochastic backjumping: occasionally jumping back to an earlier higher-heuristic state from which it might be easier to reach the goal.

A hybrid of EHC with best-first search (satisficing track). First, EHC is called, starting from the initial state. If it fails to find a solution then best-first search is called, its open list primed with the initial state and the best state seen during EHC. If at any point best-first search finds a state S with a lower heuristic than that found by EHC, best-first search is terminated and EHC restarts from S. This cycle continues until a plan is found.

Phased `SuccessorSelector` usage (quality track). The planner repeatedly used a hill-climbing algorithm, terminating when plan length exceeds that of the best plan seen so far. This was called 300 times, employing a cheap successor selector (best- or roulette-selection), and a further 200 times employing a novel successor selector that leads to shorter plans, but requires the evaluation of the heuristic values of two random successors of each state. The team provided data indicating that this 300/200 mix of successor selector usage was more effective than a 0/500 or 500/0 mix.

A portfolio system (quality track), with six in-built stochastic search algorithms, with different filters and successor selectors, each with strengths and weaknesses in certain domains or on certain problems. Prior to the competition entry, the students ran each of the six on a range of problems from a range of domains and the algorithms were ranked in the order 1–6 (with 1 being the best and 6 the worst) based on the average plan length they returned. The planner submitted to the competition then repeatedly chooses one of the algorithms at random, and if it returns a shorter plan than the best seen so far, stores it. To bias towards choosing the better algorithms, roulette selection was used to choose between them, with $fitness(A) = 13 - 2 * rank(A)$.

4.3 Overall Evaluation

As demonstrated by the individual and group performance, JAVAFF has proved to be an excellent tool for achieving our learning outcomes. Performance on the first three practicals indicates that the implementation barrier for planner development was lowered enough for students to complete the practical exercises we prescribed; from minor two- or three-line changes, to the implementation of whole sections of pseudocode. The competition is evidence that these three practicals gave the students the understanding needed to prosper in an open-ended task, developing new algorithms and evaluating their efficacy to build a competitive planner. Further, the vast range of approaches submitted indicates that the JAVAFF framework is not overly restrictive, and manages to be a useful substrate for student planner development without stifling innovation and student potential.

5 The Future Direction of JavaFF

As can be seen in this document, JAVAFF has a great potential for use as a basis for student practical exercises. At present, we are pursuing two interesting directions. First, we are developing an extended set of JAVAFF practicals for use by postgraduate students following a new MRes course in 'Automated Planning for Autonomous Systems'[4], beginning in September 2008. Working with postgraduate students, over a longer time period, the practicals will be extended to cover more ground. In particular, as JAVAFF is based on the source code of CRIKEY (Coles *et al.* 2008), we plan to devise practical exercises working on interesting aspects of temporal planning.

Second, we are developing a collection of larger dissertation project outlines for use with two cohorts of students: final year undergraduates, as the basis of their undergraduate dissertation; and the postgraduate MRes students, as part of the planning systems project. Possible directions for projects of this scale include extending JAVAFF to handle the language features of PDDL3 (Gerevini & Long 2006), exploring the exploitation of domain-specific control knowledge in a hybrid domain-specific–domain-independent manner, and an investigation into plan optimisation techniques based around plan refinement using state-to-state search within candidate solution plans.

[4] `http://www.strath.ac.uk/cis/courses/mresautomatedplanningforautonomoussystems/`

6 Conclusions

In this paper, we outlined JAVAFF, a well-crafted Java implementation of the planner FF, designed to be as accessible as possible by a student audience. As a proof-of-concept, we presented an overview of the practical exercises we constructed to complement the planning component of an undergraduate final-year AI degree module, and an *a posteriori* evaluation of these indicates that JAVAFF serves as an excellent vehicle for student planner development.

We welcome any feedback on JAVAFF, including further suggested practicals for use with it. We aim to make JAVAFF available as a community resource before September 2008, and would welcome any additions to the repository of associated literature we intend to assemble for use with it.

Acknowledgements

The authors would like to thank Keith Halsey for his work on the planner CRIKEY, from which several components were taken in the construction of JAVAFF, and the University of Strathclyde course 52426 undergraduates for their feedback on the JAVAFF practical exercises.

References

Blum, A., and Furst, M. 1995. Fast Planning through Planning Graph Analysis. In *Proceedings of the International Joint Conference on Artificial Inteligence (IJCAI-95)*.

Bonet, B., and Geffner, H. 1998. HSP: Heuristic Search Planner. Entry at the AIPS-98 Planning Competition, Pittsburgh.

Coles, A. I.; Fox, M.; Halsey, K.; Long, D. P.; and Smith, A. J. 2008. Managing Coordination in Temporal Planning using Planner-Scheduler Interaction. *To appear, Artificial Intelligence*.

Coles, A. I.; Fox, M.; and Smith, A. J. 2007. A New Local-Search Algorithm for Forward-Chaining Planning. In *Proceedings of ICAPS 07*.

Fox, M., and Long, D. 2003. PDDL2.1: An extension of PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research* 20:61–124.

Gerevini, A., and Long, D. 2006. Preferences and Soft Constraints in PDDL3. In *Proceedings of ICAPS-06 workshop on Planning with Preferences and Soft Constraints*, 46–53.

Hoffmann, J., and Nebel, B. 2001a. The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research* 14:253–302.

Hoffmann, J., and Nebel, B. 2001b. What Makes the Difference Between HSP and FF? In *Proceedings IJCAI-01 Workshop on Empirical Methods in Artificial Intelligence*.

Long, D., and Fox, M. 2003. The 3rd International Planning Competition: Results and Analysis. *Journal of Artificial Intelligence Research* 20:1–59.