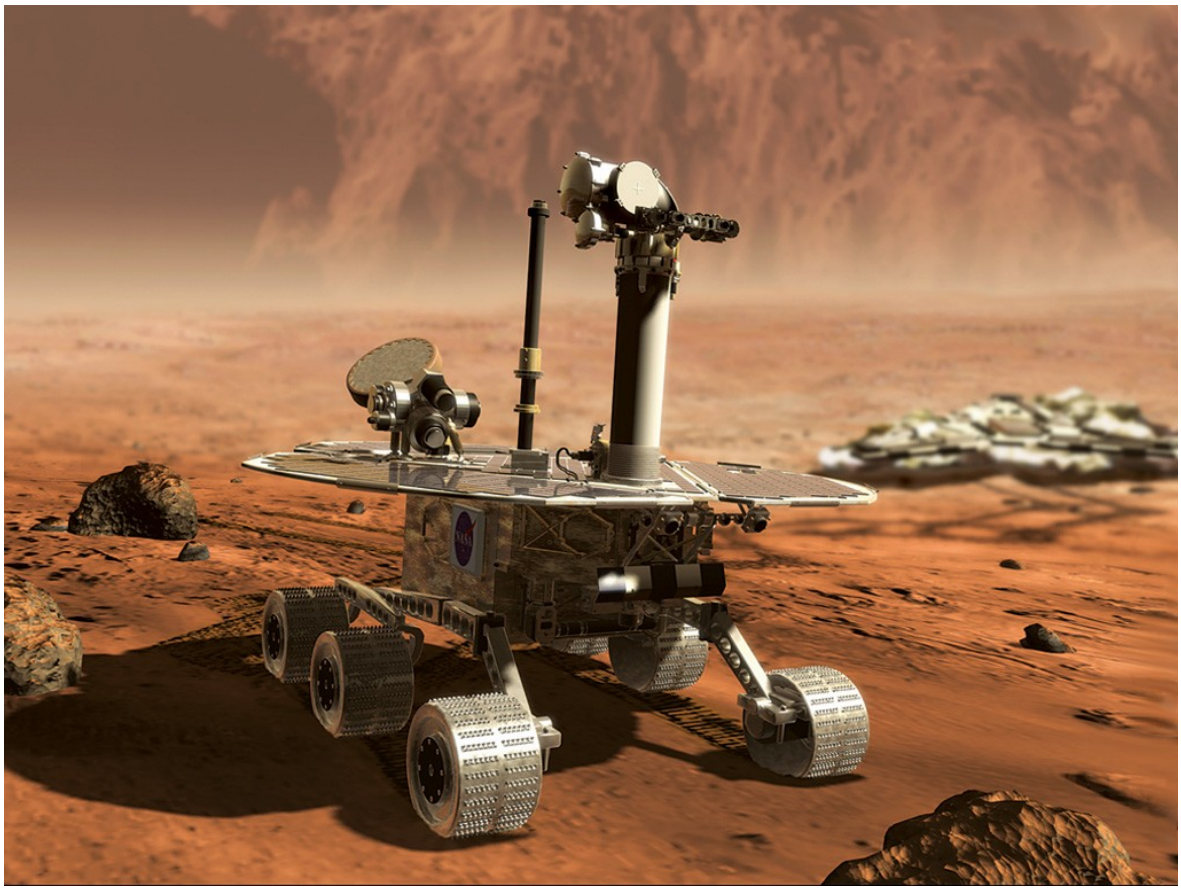
6CCS3AIP—Artificial Intelligence

Coursework, 2016

Amanda Coles & Daniele Magazzeni



```
(sample_rock rover0 rover0store waypoint3)
(calibrate rover0 camera0 objective1 waypoint3)
(take_image rover0 waypoint3 objective1 camera0 high_res)
(drop rover0 rover0store)
(communicate_image_data rover0 general objective1 high_res)
(communicate_rock_data rover0 general waypoint1)
```

Contents

Introduction

This booklet details the practicals for the coursework component of this module. Over the course of the labs, you'll use Java to write parts of a planner yourself, based on material covered in the lectures. The practicals are as follows:

1. **Beginning with JavaFF** (15 marks). Here, you'll get to grips with the planner JavaFF, familiarising yourself with the key parts of the source code. To help you get going, you'll make a few modifications, and then look at how these affect the performance of the planner.

2. **Good Old-Fashioned Hill Climbing** (25 marks). Building on the previous practical, you will implement a generic hill-climbing local search algorithm within JavaFF.

3. **Successor Selection and Neighbourhoods** (20 marks). As will be discussed in the lectures, in local search in general, the successor selection and neighbourhood functions are an important part of the system. In this practical, you will look at writing a few of these, and using them within your hill-climbing local search algorithm. Then, as in the first practical, you will measure their effect on performance.

4. **The King's Planning Competition** (40 marks). As the culmination of this work, you will split into teams and work together on producing a planner to be entered into a competition. You will look at two things: planning quickly; and how local search can be used for optimisation in planning, seeing how good a plan you can find after 10 minutes. There will be a prize for the best planner in each of these two categories.

For each practical, you need to submit it to KEATS by the relevant deadline; and then attend the subsequent lab session to have your work marked. The deadlines are as follows:

- Practical 1: submit by 23:55 on **October the 15th**. This work will be marked on **October the 18th**.

- Practical 2: submit by 23:55 on **November the 5th**. This work will be marked on **November the 8th**.

- Practical 3: submit by 23:55 on **November the 19th**. This work will be marked on **November the 22nd**.

- Practical 4: submit by 23:55 on **December the 10th**. Feedback will be delivered by **December the 13th**.

Usual rules for late submission of work apply. Similarly, as the lab is your *assessment opportunity* for the work, if you don't turn up to the lab, you get zero. In case of robot apocalypse, sickness, or other good cause, submit a Mitigating Circumstances Form to departmental office as soon as you are able to. If in doubt, talk to your personal tutor.

We hope you enjoy these practicals, and welcome any feedback about the work.

Practical 1

Beginning with JavaFF

JavaFF is an implementation of the planner FF [3] written in Java. Based on the source code for CRIKEY, written by Keith Halsey, it has been produced for this course to provide an extensible implementation of FF that cleanly separates the features we are interested in: search algorithm; heuristics; neighbourhoods; and so on.

In this practical, you'll be running the planner; then making some small modifications to it to gain familiarity with the code. You'll be able to complete Exercise 1 after the first lecture. For Exercise 2, you might want to wait until after the second lecture (on FF) before attempting it, but feel free to read on if you are curious.

1.1 Getting Started

First, the Java source code for JavaFF along with some example planning problems and domains is available from:

`http://www.inf.kcl.ac.uk/staff/andrew/JavaFF.tar.gz`

Then, expand this into somewhere in your home directory. Assuming you are using a Linux machine these two steps can be performed as follows:

```
cd
wget 'http://www.inf.kcl.ac.uk/staff/andrew/JavaFF.tar.gz'
tar -zxf JavaFF.tar.gz
```

This will create a Java package directory `javaff` and a directory `examples` containing the example planning domains and problem files. To compile JavaFF (optional at this stage—it comes pre-compiled) type:

```
javac javaff/JavaFF.java
```

Finally, to run JavaFF on one of the example files, type:

```
java javaff.JavaFF examples/driverlog/domain.pddl \
     examples/driverlog/pfile01
```

After a second, this should display a solution plan to the problem. If you want to run it on some more domains and problem files, the examples included are as follows:

- **driverlog**—a logistics domain, where the goal is to move some packages, trucks and drivers to from their initial locations to their specified goal locations.
- **rovers**—a domain modelling the activities of planetary rovers, where the goal is to have performed some science gathering activities (photographs, soil sampling) and have communicated the data to Earth.
- **depots**—a warehouse management domain, where the goal is to move around pallets and store them stacked in a specified configuration.

To run JavaFF on any of these, follow a similar process as above: specify the relevant `domain.pddl` as the first argument to the planner, and the problem file to use as the second. If you want to see what the example files look like themselves, open them with a text editor—PDDL, the language used to define problem domains and files, is a plain-text format.

NB: Depending on how much memory the Java VM uses by default, it may or may not run out of memory when running JavaFF. To allow it to use more, add the option `-Xmx512m` as the first parameter to `java`, where 512 is the maximum number of megabytes of memory it is allowed to use. So, for instance:

```
java -Xmx512m javaff.JavaFF examples/driverlog/domain.pddl \
     examples/driverlog/pfile12
```

1.1.1 Exercise 1: Gathering Data *5 marks*

An important part of science, and computer science is no exception, is experimentation to investigate hypotheses. In planning, the hypothesis is usually that modifying a planner in a certain way means it can find a plan in less time, and experiments take the form of comparing how long it takes to find a plan before and after a change is made. By comparing these data, the aim is to determine whether the modification is a good idea. In this exercise, you're going to gather some data on how long it takes JavaFF in its original form to find plans on some problems from the example domains. We'll use these data as a benchmark, later in this practical.

The two domains you're going to look at are **rovers** and **driverlog**. Run JavaFF on the first ten problem files (`pfile01` to `pfile10`) and note down the *planning time*, reported by JavaFF after printing the plan to screen. Store the data in a spreadsheet, with one sheet for **rovers** and another for **driverlog**.

Hint: to cut down on the amount printed to screen, pipe the output for the planner through the `grep` command as follows:

```
java javaff.JavaFF examples/driverlog/domain.pddl \
     examples/driverlog/pfile01 | grep "Planning Time"
```

This will only print out the Planning Time line, as nothing else matches the regular expression passed to `grep` in the quotes.

1.2 Looking Under the Bonnet

Now you've got JavaFF up and running, and gathered some data, let's take a look at the source code.

1.2.1 The Main Class

The first file to look at is `javaff/JavaFF.java`—the source code for the main class. Most of this file is 'boilerplate' that calls the code to load and parse the planning problem and domain; the interesting bit is at the bottom: the `performFFSearch` method. Open it up in your favourite text editor, and read through that method in its entirety to familiarise yourself with it at a high level. Then, come back here and read through a breakdown of what it does.

Call EHC Search

The first thing `performFFSearch` does is try and search using Enforced Hill Climbing, and only using the 'helpful actions' (see lecture for details). In code, this looks like:

```
EnforcedHillClimbingSearch EHCS
           = new EnforcedHillClimbingSearch(initialState);

EHCS.setFilter(HelpfulFilter.getInstance());

State goalState = EHCS.search();
```

The first line creates an instance of an EHC search algorithm object; the next sets it to use just the helpful actions; and the finally, `search()` is called to try and find a goal state.

Use Best-First Search

If EHC with helpful actions fails (if `goalState == null`), FF resorts to best-first search. Here, it doesn't use just the helpful actions: it uses all of the actions applicable in each state. In code, this looks like the following:

```
  BestFirstSearch BFS = new BestFirstSearch(initialState);

  BFS.setFilter(NullFilter.getInstance());

  goalState = BFS.search();
```

As you can see, this is fairly similar to invoking EHC. The differences are that a BestFirstSearch object is created; and we use `NullFilter`, as opposed to `HelpfulFilter`, to keep all actions in each state rather than just the helpful ones.

1.2.2 Filters

Referring to the lecture materials, when searching forwards for a solution plan (in FF [3], HSP [1], Identidem [2], ...) a state S is *expanded* by applying actions to it. The question, of course, is which actions to apply: in JavaFF, this is the role of Filters. Filters are used to filter the list of actions to consider applying in S when expanding it. By putting the filter code it in a separate class, and passing an instance of this to a search algorithm, we can use different filters without changing the algorithm class.

We've glossed over two `Filters` so far: `NullFilter` and `HelpfulFilter`. Let's have a look at them in some more depth, beginning with the `javaff.planning.Filter` interface, which these both implement:

```
public interface Filter
{
     public Set getActions(State S);
}
```

As you can see, it's quite a small interface providing just one method, `getActions`. This method takes a state S as as its argument, and returns a `Set` containing the actions to consider applying in that state. Now let's look at the two filters themselves.

NullFilter

The simplest filter, the `NullFilter` returns all the actions applicable in a state; no more, no less. This is what FF uses if it resorts to best-first search: it makes sure it looks at every possible action to consider that might be a good idea; hence preserving completeness. Open up `javaff/Planning/NullFilter.java` in a text editor and read through the code. The most interesting method is `getActions` method, which works as follows:

```
public Set getActions(State S) {
  Set actionsFromS = S.getActions();
```

This calls the `getActions` method on S. This method returns a `Set` of all the actions whose logical preconditions are satisfied in S; i.e. those which, logically, can be applied. There are a few nice short-cuts that can be taken to find the applicable action set quickly, and one of the nice aspects of reusing the JavaFF implementation is that those are all already done. If you want to know more, then delve into the code!

```
  Set ns = new HashSet();
  Iterator ait = actionsFromS.iterator();
  while (ait.hasNext())
  {
    Action a = (Action) ait.next();
    if (a.isApplicable(S)) ns.add(a);
  }
  return ns;
}
```

Having determined which actions' logical preconditions are satisfied in S, what remains is to check that the numeric preconditions are really satisfied too[1]. This is performed by calling `isApplicable` on each action - if it passes this additional test, it is added to the set ns to return.

HelpfulFilter

The `HelpfulFilter` returns all the helpful actions in a state; this is what FF uses when searching with EHC. If you recall the lecture on FF, the helpful actions are determined from the *relaxed plan* used to give the heuristic value for each state. If a state S has a relaxed plan which has the actions A, B, C in its first action layer, then any action that adds anything added by A,B or C is helpful. Open `javaff/Planning/HelpfulFilter.java` in a text editor and read through the code. As you can see, it's very similar to `NullFilter`, the main difference is in these three lines:

```
STRIPSState SS = (STRIPSState) S;
SS.calculateRP();
Iterator ait = SS.helpfulActions.iterator();
```

Skipping the first for a moment, `SS.calculateRP()` calculates the relaxed plan on the state. This is a member method of the `STRIPSState` class, and stores the results of the relaxed plan computation in the member variables of SS. The line after this then iterates over the helpful actions, which is the key difference between this and `NullFilter`: iterating over just the helpful actions rather than all the actions. Beyond this point, it proceeds as before, checking the actions really are applicable.

The first line, as you can see, is a cast: the method takes a `State`, and the method to calculate a relaxed plan is defined in `STRIPSState`. However, the `Filter` interface method expects a `State` so that's what this method has to take. For our purposes, this is entirely harmless: in practice, all the `States` we deal with will be `STRIPSState`, or a sub-class thereof.

1.2.3 A Moment's Breath....

By now, you should have some understanding of the following key aspects of JavaFF:

- The structure of the `performFFSearch` method (1.2.1); specifically, that EHC is called then, if necessary, Best-First Search.
- The role of a `Filter` in defining what actions to consider when expanding a state:
  - what the `NullFilter` (1.2.2) does; and
  - what the `HelpfulFilter` (1.2.2) does.

If you're unsure of any of these, read through the text again, ask the TA questions, or come to see us in our office hours. You don't need to be able to recite the code from memory, just be familiar with the key concepts employed. Once you're happy with this, read on....

[1]...and any temporal constraints—these will be covered later in the lecture series when discussing temporal planning

1.3 Making Some Changes

Over the course of these practicals, you'll be making a series of changes to JavaFF. To get you into the swing of things, you're going to start by making two changes. Then, you'll run the modified planner on a few of the example files and see what effect the modifications have had on its performance.

1.3.1 Exercise 2: EHC with the NullFilter *5 marks*

As discussed in Section 1.2.1, JavaFF searches in two phases:

1. Enforced Hill Climbing, with the HelpfulFilter; and if this fails,
2. Best-First Search, with the NullFilter.

The filter to use with an instance of a search algorithm object is set using the `setFilter` method, for instance:

```
Searcher.setFilter(HelpfulFilter.getInstance());
```

In this exercise, change the `performFFSearch` algorithm in `javaff/JavaFF.java` to use a `NullFilter` rather than a `HelpfulFilter` with the Enforced Hill Climbing Search. Compile it and run a few files on it to check it works (as per the instructions in Section 1.1) and when you're happy, move on.

Gathering Data

The change you have just made is to alter EHC to use all the actions, rather than just the helpful actions. The question is, is this a good idea? If you recall earlier, in Exercise 1, you gathered some data on the performance of the unmodified version of JavaFF. We can compare the performance of your modified planner to this to investigate the following hypothesis:

Using Helpful Actions with EHC offers improved performance.

The original data you have are with helpful actions enabled (using the `HelpfulFilter`). Now, use your modified planner to gather data on the performance with all the actions, in both **rovers** and **driverlog**. Record these data in the spreadsheet and plot both columns on a graph. What are your observations?

1.3.2 Exercise 3: Searching in Three Phases *5 marks*

In this exercise, change the `performFFSearch` algorithm in `javaff/JavaFF.java` to search in three phases, rather than two:

1. Enforced Hill Climbing, with the HelpfulFilter; and if this fails,
2. Enforced Hill Climbing, with the NullFilter; and if this fails,
3. Best-First Search, with the NullFilter.

Hint: `search()` returns `null` if it fails to find a plan.

Gathering Data

The change you have just made is to move over to a three-phase search process; and, like with the previous modification, we can gather data on whether or not this is a good idea. Use your modified planner to add a third column to your spreadsheets for the **rovers** and **driverlog** domains, and plot another graph. What are your observations?

1.4 Wrapping Up

By now, you should have some idea about the coarse structure of JavaFF, and the role of `Filters`. Also, you should be familiar with how to gather data on the performance of a planner, and plotting these data on graphs to illustrate comparative performance. In the next practical, you'll delve deeper into main search algorithm used (EHC), and write a new search class that uses Hill Climbing.

Practical 2

Good Old-Fashioned Hill Climbing

In this practical you'll make some more changes to JavaFF, this time on a grander scale: writing a new search algorithm, based on the idea of *hill climbing*. No evaluation this time: the focus is on writing the search algorithm, and we'll evaluate it (along with other ideas) in the next practical.

2.1 Starting Point: EHC

As discussed in lectures, the default search algorithm used in FF is Enforced Hill Climbing (EHC). EHC searches forwards from the initial state I, trying to find a state with a heuristic value better than the best seen so far. If it can find such a state, it commits to choosing it and searches from there. Otherwise, it uses breadth-first search to expand more states until it finds one.

The implementation of EHC uses an *open list* of states to visit, initially containing I, and keeps track of the best heuristic seen so far, initially $best = h(I)$. The main loop takes the state from the front of the list, S, and applies actions to the state finds the successor states, using a `Filter` (Section 1.2.2) to decide which actions to apply. One of three things then happens:

- if a successor S' is a goal state, it is returned as a solution;
- if a successor S' is found with $h(S') < best$, then the open list is cleared, and only S' is added to it; otherwise,
- all the successors S' are added to the open list, awaiting expansion.

To stop search going around in circles, a *closed list* is kept of the *memoised states* visited so far, and EHC never considers the same state twice: if a successor S' is in closed, it is discarded. If the open list becomes empty, then EHC has failed to find a plan.

2.1.1 EnforcedHillClimbingSearch.java

We'll now have a look at this, written in Java. First, open the file
`javaff/search/EnforcedHillClimbingSearch.java`

...in a text editor and read through it, particularly the `search()` method. Then, come back here for a breakdown of what it does.

The Member Variables

The class has four important member variables:

```
protected BigDecimal bestHValue;
protected Hashtable closed;
protected LinkedList open;
protected Filter filter;
```

The first three of these correspond to the best heuristic value seen so far, the closed list, and the open list, respectively. `filter` is the filter to use when expanding states. A further important member variable is inherited from `Search`—the initial state, `start`.

The `search` method

The `search` method forms the main body of the EHC search algorithm. Let's go through it, stage by stage. To reduce clutter in this document, we'll leave out any `println` statements and the comments.

```
if (start.goalReached()) {
  return start;
}

needToVisit(start);
open.add(start);
bestHValue = start.getHValue();
```

First, we check whether the initial state, `start`, is a goal state. In any interesting problem, it won't be, but it is sensible to check. The next three lines then initialise the member variables ready to start search: `start` is added to the closed list, by calling `needToVisit`; then it is put on the open list; and finally its heuristic value is taken as the best seen so far.

```
while (!open.isEmpty())
{

  ... expand the state s at the front of the open list...

}
return null;
```

Next is the main loop, which removes the next state off the open list and expands it. The code inside the while is responsible for expanding states, and we'll come onto that in a second. However, the key observation is that if the open list becomes empty, then the while completes and `null` is returned, indicating failure.

Now to inside the while:

```
State s = removeNext();

Set successors = s.getNextStates(filter.getActions(s));
```

This removes the next state from the open list (see the `removeNext()` method higher up the file), keeping it as the variable s. Then, the successor states to this are found:

- `filter.getActions(s)` uses `filter` to get the set of actions to consider applying in s;
- `s.getNextStates(...)` takes a set of actions and produces a set of the successor states reached by applying each of the actions individually to s.

```
Iterator succItr = successors.iterator();

while (succItr.hasNext()) {
  State succ = (State) succItr.next();

  if (needToVisit(succ)) {
```

Next, once we have the set of successors, we iterate over them, with `succ` denoting the current successor under consideration. For each successor, the first check is if we need to visit it: the call to `needToVisit` returns false if it is already in the closed list; otherwise, it adds it and returns true. Assuming it's not on the closed list, then....

```
if (succ.goalReached())
  return succ;
} else if (succ.getHValue().compareTo(bestHValue) < 0) {
  bestHValue = succ.getHValue();
  open = new LinkedList();
  open.add(succ);
  break;
} else {
  open.add(succ);
}
```

These three correspond to the branches detailed earlier in Section 2.1. The key to the middle branch is that the open list is cleared by replacing it with a new one containing only `succ`, and then `break` skips the evaluation of the remaining successors: search jumps back to the start of the outer loop (`while (!open.isEmpty())` etc).

2.2 Exercise 1: Turning EHC into HC *15 marks*

The first exercise in this practical is to turn the enforced hill climbing algorithm into a standard hill climbing algorithm. The pseudocode for normal hill-climbing, keeping the closed list, is shown in Algorithm 1: use this as a basis for your implementation.

First, copy `javaff/search/EnforcedHillClimbingSearch.java` to a new file `javaff/search/HillClimbingSearch.java`. Then, open the file in a text

Algorithm 1: Hill Climbing Algorithm

```
Data: I - initial state
Result: S - a goal state
S ← I;
while S is not null 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
```

editor, changing the class and constructor names, as appropriate. Now, make the necessary modifications to the `search()` method. Algorithm 1 will serve as a useful guide for this. There are three key differences between this and the EHC implementation to bear in mind when making your modifications:

1. the `bestHValue` variable is local to the while loop, not a member variable, so it keeps the best heuristic values of the successors to S, rather than the best heuristic value ever seen.

2. *all* the successors iterated over (no `break`), keeping the joint-best;

3. the open list only contains a single state: there should only be one `open.add(...)` line, adding one of the joint-best successors.

Once you have completed this step, modify `javaff/JavaFF.java` to create an instance of your `HillClimbingSearch` object, rather than an `EnforcedHillClimbingSearch` object. Compile it and run a few of the smaller problem files to check it works (as per the instructions in Section 1.1) and when you're happy, move on. You should notice that each time the planner is run, it can produce slightly different plans: this is because of the random choice between successors.

Hint: random numbers can be obtained by using the random number generator initialised in the main class; for instance, to get a random integer in the range 0 (inclusive) to 10 (exclusive):

```
int r = javaff.JavaFF.generator.nextInt(10);
```

2.3 Exercise 2: Depth-Bounded Hill Climbing *5 marks*

In this exercise you're going to make a small, but significant, modification to your HillClimbingSearch class. The aim of this modification is to add a depth bound to the search algorithm, so that after it's searched a number of steps, it will terminate early. The suggested way to do this is as follows:

1. add a class member variable, `maxDepth` along with a setter method;
2. add a variable `depth` to `search()`, incrementing it by 1 each time the main while loop goes around;
3. return null if `depth == maxDepth`.

You may however use any solution you wish, so long as it works. Add a line to set a depth bound of 20 `javaff/JavaFF.java` before calling `search()` on your HC object and see what happens.

The random choice between successors will mean it produce different solutions to the same problem, but now given we have a fixed depth bound of 20 this randomness will mean that sometimes it hits the bound and sometimes it doesn't – it's down to chance whether it actually manages to solve the problem. Let's do something about that....

2.4 Exercise 3: Depth-Bounded Search with Restarts

5 marks

The final exercise in this practical works on addressing the problem with randomness and the depth bound encountered in your previous depth-bounded hill-climbing algorithm. Modify `javaff/JavaFF.java` to place a `for` loop around the creation and use of the HillClimbingSearch algorithm that sets an increasing depth bound before calling it each time. Something like:

```
for (int depthBound = 5; depthBound < 100; ++depthBound) {
  ...create HillClimbingSearch Object and search to bound here
  ...return solution if not null
}
```

This will make a number of calls to your hill-climbing searcher, increasing the depth bound each time. In doing so, we get several chances of the random tie-breaking making the right choice, and don't have a fixed limit of 20 steps in a plan, preventing us from solving larger problems.

2.5 Wrapping Up

By now, you should have a feel for how the classical local search hill-climbing algorithm can be used as the basis of a planner. Even if you haven't got everything working, hand your work in on time, and take it to the lab to be marked.

Practical 3

Successor Selection and Neighbourhoods

In this practical, you're going to look at the role of successor selection functions and neighbourhood functions in forward-chaining local search. You will need your solution to practical 2 for this; if you have had trouble completing it, we will post a model answer to KEATS shortly after the deadline for practical 2.

3.1 Downloading the Second Code Bundle

For this practical, you'll need to download some additional Java code. The code is available from:

`http://www.inf.kcl.ac.uk/staff/andrew/JavaFFPackageTwo.tar.gz`

Expand this into your home directory as last time, and it will unpack some new files into your `javaff` package tree.

Assuming you are using a Linux machine this can all be done using:

```
cd
wget 'http://www.inf.kcl.ac.uk/staff/andrew/JavaFFPackageTwo.tar.gz'
tar -zxf JavaFFPackageTwo.tar.gz
```

3.2 Successor Selection

The important new interface introduced in this code bundle is `SuccessorSelector`. It is defined as follows:

```
public interface SuccessorSelector {

  State choose(Set toChooseFrom);

}
```

It defines a single method, which takes a set of successor states, and returns a single one of these. This can be used to abstract out the process of choosing the successors of a node, once it has been expanded. For instance, in hill climbing search, in the previous practical, the a successor with the (joint) best heuristic value was chosen. A class `BestSuccessorSelector` implements this. Open the file `javaff/search/BestSuccessorSelector.java` in a text editor and read through the code. It should be reasonably self explanatory: it loops over the states, keeping the best seen so far in a `HashSet`, `jointBest`. Then, it chooses one of these at random, and returns it. We'll come back to successor selectors later, but next we'll look at an example of an algorithm that uses them.

3.3 Exercise 1: Using Successor Selectors *6 marks*

In this exercise, you are to modify the hill climbing search algorithm you wrote in practical 2 to take an object of type `SuccessorSelector` and to use this to choose between the states, rather than it being fixed to choose one of the joint-best.

3.3.1 Coding

The key differences in terms of implementation are as follows:

- You'll need a member variable and setter function to pass the selector, much like the existing one for `Filter`.
- You'll need to keep all the new successors in a set, rather than just the best ones; and then,
- You'll use the successor selector to choose between these (if there are any).

Once you have made these modifications, modify `javaff/JavaFF.java` to pass an instance of a `BestSuccessorSelector` object to the search algorithm to use as its selector; for instance:

```
HillClimbingSearch hcs = new HillClimbingSearch(initialState);
hcs.setSelector(BestSuccessorSelector.getInstance());
```

3.3.2 Evaluation

As in practical 1 (Section 1.1.1), run JavaFF with your new hill climbing algorithm that uses `BestSuccessorSelector` on 10 problems from **driverlog** and 10 from **rovers**. Store the data in a spreadsheet, with one spreadsheet for each of the two domains.

3.4 Exercise 2: A 'Random Three Helpful' Filter *6 marks*

In this exercise, you are going to write a new `Filter` that chooses three random helpful actions, and returns these as the successors to consider visiting.

3.4.1 Coding

Copy the `NullFilter` source to a new file, `RandomThreeFilter`. Use the following code as a skeleton for the main body of the class (you can keep the existing package statement and imports).

```
public class RandomThreeFilter implements Filter
{
  private static RandomThreeFilter rf = null

  protected HelpfulFilter hf;

  private RandomThreeFilter()
  {
    hf = HelpfulFilter.getInstance();
  }

  public static RandomThreeFilter getInstance(int k)
  {
    if (rf == null) rf = new RandomThreeFilter();
    return rf;
  }

  public Set getActions(State S)
  {
    Set helpfulFiltered = hf.getActions(S);
    Set subset = new HashSet();

    // add code here to pick 3 from 'subset' at random
    return subset;
  }
}
```

Complete the implementation of `getActions`. Then, in `javaff/JavaFF.java`, change the filter used with hill climbing search to this, rather than `HelpfulFilter`.

3.4.2 Evaluation

You now have a choice of two interesting filters to use with the local search algorithm: `HelpfulFilter` and your new filter, `RandomThreeFilter`. The question now is which one gives the best performance? Run the new version of the planner, using `RandomThreeFilter`, and record the data next to those obtained with the unmodified version. What are your observations?

3.5 Exercise 3: Roulette Selection *8 marks*

In this exercise, you are going to write a new `SuccessorSelector` that implements roulette selection. Roulette selection is explained in the following Wikipedia article:

http://en.wikipedia.org/wiki/Fitness_proportionate_selection

When implementing it for use to choose between planning states, use $1/h(S)$ as the *fitness function* for each state S: the lower the heuristic, the fitter the state.

3.5.1 Coding

You may use the implementation of `BestSuccessorSelector`, the key difference here is that rather than picking out the joint-best states and returning one of these, you will need to calculate the fitness values of the states and then pick a random segment of the roulette wheel. Then, in `javaff/JavaFF.java`, change the filter back to `HelpfulFilter` and change the successor selector used to your roulette selector, rather than `BestSuccessorSelector`.

Hint: Sum the sizes of the segment, then use:

```
double r = javaff.JavaFF.generator.nextDouble() * sum;
```

... to choose how far around to spin the wheel, r.

3.5.2 Evaluation

You now have a choice of two interesting successor selection functions to use with the local search algorithm: `BestSuccessorSelector` and your new filter, `RouletteSelector`. The question now is which one gives the best performance? Run the new version of the planner, using `RouletteSelector`, and record the data next to those obtained with the unmodified version (from Exercise 1). What are your observations?

3.6 Wrapping Up

By now, you should have a good understanding of the role of how the `Filter` interface can be used to define neighbourhoods for use in local search, and how `SuccessorSelectors` can be used to define successor selection function. You should also have some more experience in gathering data, and comparing performance. In the next and final practical, you'll bring together all the ideas seen so far in this series of practicals to work on a further interesting slant for local search — optimisation.

3.7 Further Reading: `LocalSearch.java`

After the deadline for this practical, an implementation of another local search algorithm will be posted to KEATS: `LocalSearch.java`.

This works in a similar manner to Identidem [2], and is included as a further example of how local-search can be used in planning. Reading this may give you a competitive advantage in the next practical, indeed you may use it if you wish as a basis for your own work, so it's worth spending a few minutes looking over this.

The key difference between this algorithm and and hill-climbing is that it keeps track of the best state seen so far, and uses it in two ways:

- when the number of steps since the best state exceeds the *depth bound*, then rather than aborting it restarts from the best state; and instead,
- when the total number of these restarts exceeds a *restart bound*, it aborts.

Open the source code for it in a text editor, then come back to read through a discussion of it below.

3.7.1 Member Variables

The member variables are similar to those of the EHC class, with the addition of:

```
protected SuccessorSelector selector = null;

protected int depthBound = 10000;
protected int restartBound = 10000;
```

`selector` stores the `SuccessorSelector` to use, set using the corresponding setter method lower down in the file. As you'll see, this is used in the `search()` method. `depthBound` and `restartBound` are the depth and restart bounds; again, these are set using setter methods, but are initialise to large values so that the algorithm will still function even if the user forgets to do this.

3.7.2 The `search()` method

This will be similar to that of your HC algorithm from the last practical. The key differences are the tracking of the best state, the use of a `SuccessorSelector`, and the handling of restarts back to the best state. These will be covered now.

```
int currentDepth = 0;
int currentRestarts = 0;

State bestState = start;
BigDecimal bestHValue = start.getHValue();
Hashtable bestClosed = (Hashtable) closed.clone();
```

This initialises the number of depths and restarts performed to zero, the best state seen to the initial state, the best heuristic value to its heuristic value, and the closed list to the number of states visited by that point. The importance of `bestClosed` will be discussed in a moment.

Then, inside the `while` loop:

```
Set successors = s.getNextStates(filter.getActions(s));

Set toChooseFrom = new HashSet();

... put all the successors which haven't been
    seen before into this set                  ...

if (!toChooseFrom.isEmpty()) {
  State chosenSuccessor = selector.choose(toChooseFrom);
```

This illustrates the practical use of a `SuccessorSelector`, stored in the member variable `selector`. All the successors (that haven't been seen earlier) are placed into a set, and `selector` chooses one of these. Once this has been chosen, the next chunk of code decides what to do next:

```
if (chosenSuccessor.getHValue().compareTo(bestHValue) < 0) {
  currentDepth = 0;
  open.add(chosenSuccessor);
  bestState = chosenSuccessor;
  bestHValue = chosenSuccessor.getHValue();
  bestClosed = (Hashtable) closed.clone();
```

If the new successor is the best seen so far, update the details of the best state to match this: `bestState`, `bestHValue` and `bestClosed`. Then, put the successor on the open list so it is visited next.

Otherwise, if the successor isn't better than the best seen so far:

```
} else {

  ++currentDepth;
  if (currentDepth < depthBound) {
    open.add(chosenSuccessor);
```

This increments the counter of the number of steps taken since a new best state was seen—if this remains less than the bound, the chosen successor is added to the open list for expansion. Otherwise, recalling the outline discussion of the algorithm, we want to trigger a restart back to the initial state; or, if appropriate, terminate. First we check if the number of restarts has reached the restart bound, and if it has, return null to terminate:

```
  } else {
    ++currentRestarts;
    if (currentRestarts == restartBound) {
      return null;
    }
```

Otherwise, trigger a restart back to the best state:

```
    currentDepth = 0;
    closed = (Hashtable) bestClosed.clone();
    open.add(bestState);
  }
}
```

This adds `bestState` to the open list, to search forwards from it again. It also resets the depth count (the distance of `bestState` from itself is necessarily 0) and replaces the closed list. As was mentioned before, this closed list replacement is important. When the current best state is first expanded, its successors are added to closed. If we restart to this a second time, without replacing the closed list, all the successors will have already been seen, and search would terminate. Thus, when restarting to the best state, we also take the closed list to be just those states seen up to that point; not including those seen since.

Practical 4

The King's Planning Competition

In this practical, we will recreate one of the most exciting events in the planning calendar: the International Planning Competition (IPC). The IPC is a (roughly) biennial event, and provides a platform for planning researchers across the globe to run their planners head to head and see how good the state-of-the-art is. The Planning group at King's have organised and participated in past competitions, and their planners have received a number of awards. The domains you have looked at so far were used in the 2002 competition, organised by Maria Fox and Derek Long; the competition website is at `http://ipc02.icaps-conference.org/` or you can read the paper on the results [4]. For details of the other competitions, visit: `http://ipc.icaps-conference.org/`.

For this practical, unlike the others in this series, you will be working in groups. This will give you a good opportunity to pool your ideas, and work together on coming up with really good ideas for planning. Once we have all the entries, we will run them on a range of problems taken from the domains: those you have in the `examples` folder, plus another new domain. After running the planners, we'll analyse the data, and present prizes to the best groups in the last lecture of term.

4.1 Track 1: Satisfycing Planning *25 marks*

4.1.1 Outline

In this track of the competition, the goal is to write a planner which can find a plan as quickly as possible. Here, we don't care about the quality of the solution produced (i.e. how long it is), so long as it is a valid plan; we are only concerned with the planning time needed to find it.

To produce this planner, start with your hill climbing search algorithm and investigate how best to use this: filters, successor selectors, depth bounds, and so on. You may make whatever changes you want to your algorithm, invent new filters and so on, whatever it takes to make it faster. Some ideas you might want to consider are:

- how about choosing 2 or 4 at random instead of 3 in `RandomThreeFilter`?
- in roulette selection, how about using $1/(h^2)$ as the fitness function?

- have a look at the source for `LocalSearch.java` discussed at the end of the previous practical (Section 3.7) for some new ideas about restarts

These are only suggestions, though: extra marks will be awarded for novel solutions.

4.1.2 Deliverable

There are two things you need to give in. First, the source code for your solution. Submit this as a ZIP archive to KEATS, with the filename `FastPlanner.zip`. Second, you need to write around half a page to a page about your planner and how it works: what decisions you made about filters, successor selectors, any novel ideas, and so on. We'll give out these descriptions to the other groups after the results have been announced so you can see how other people approached the problem.

4.2 Track 2: Optimising Planning *15 marks*

4.2.1 Outline

In this track of the competition, you will work on a planner that aims to produce short plans, in terms of the number of actions they contain.

The simplest way to do this is to call your fast planning algorithm repeatedly, and every time it produces a solution, seeing if the solution plan is shorter than the best seen so far. For instance:

```
State bestGoalState;
int bestPlanLength = 100000;

for (int i = 0; i < 100; ++i)
{
  State goalState;

  ... use your fast planner here ...

  TotalOrderPlan thePlan = (TotalOrderPlan) goalState.getSolution();
  int planLength = thePlan.getPlanLength();

  if (planLength < bestPlanLength) {
    bestGoalState = goalState;
    bestPlanLength = planLength;
    infoOutput.println("Best length: " + bestPlanLength);
    thePlan.print(infoOutput);
  }
}

return bestGoalState;
```

It is possible to do better than this, though. For instance, you could set a depth bound with your local search algorithm to the best plan length seen so far, so that it stops if it reaches this: there's no point carrying on if it's definitely going to do worse. There are lots of good ways of doing this, and as in the previous track, there are marks going for anything that works, and extra marks for neat solutions

4.2.2 Deliverable

As in the previous track, submit an archive of your work, with the filename `QualityPlanner.zip`, along with half a page to a page about how you went about solving the problem.

4.3 Rules

The key emphasis is on coming up a domain-independent planner. Domain-specific code is not allowed. For instance, don't do something like:

```
if (domainFile.contains("driverlog")) {
  search this way
} else if (domainFile.contains("rovers")) {
  search another way
} ...
```

This will result in disqualification. Further, as mentioned in the introduction, along with the three domains you have, we will be testing them on one new unseen domain, details of which we will present after the competition.

When running the planners, we will subjecting them to a 10-minute time limit on each problem, and 2Gb of memory. With the optimising track, we will take the best solution after 10 minutes so you must remember to print the best plan every time a new one is found:

```
TotalOrderPlan thePlan = (TotalOrderPlan) goalState.getSolution();
int planLength = thePlan.getPlanLength();

infoOutput.println("Best length: " + planLength);
thePlan.print(infoOutput);
```

4.4 Go Forth and Plan...

That brings us to the end of the practical series. We hope you have found them to be a useful learning experience, helping you to really get to grips with important concepts in planning. But, above all, we hope you have enjoyed working on planning as much as we do, and have gained a good understanding of what planning research is all about: finding new and exciting ways of searching for solutions, always seeking to push the boundaries of what existing planners are capable of. If you have enjoyed the course, and want to further your knowledge in this area, the planning group is always interested in new, enthusiastic

postgraduate students; so contact one of the members of academic staff in the group to discuss opportunities for working with the group.

Bibliography

[1] Blai Bonet and Hector Geffner. HSP: Heuristic Search Planner. Entry at the AIPS-98 Planning Competition, Pittsburgh, June 1998.

[2] A. I. Coles, M. Fox, and A. J. Smith. A New Local-Search Algorithm for Forward-Chaining Planning. In *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling (ICAPS 07)*, September 2007.

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

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