# HLogEdu-Search

carlos@diei.udl.cat
jponfarreny@diei.udl.cat

## How to display the list of algorithms and problems

```
$ hlogedu-search list
```

## How to display algorithms and problems information

```
$ hlogedu-search infoa <algorithm>
$ hlogedu-search infop <problem>
```

## How to execute a search

```
$ hlogedu-search run -a <algorithm> -p <problem>
```

## How to limit the search depth

```
$ hlogedu-search run -a <algorithm> -p <problem> -md <max_depth>
```

## How to specify a heuristic function

```
$ hlogedu-search run -a <algorithm> -p <problem> -hf <heuristic>
```

## How can I load my own problems

You can manually specify the directory that contains your implemented problems using the command line option "-pd" or "--problems-dir" and then the desired action.

```
$ hlogedu-search -pd <path> [run/list/infoa/infop] …
```

**Note**: By default the tool looks for Python files "*.py" inside a subdirectory *problems*, in the current working directory.

## How to convert the output into an image

Edulog-Search outputs the search tree using the graph description language "dot"[1], this output can be passed to any of the *graphviz*[2] package tools to transform it into a PNG/SVG/PDF/PS/… file. The tool that produces the best results out of the box is *dot*, and the operation can be simplified by pipelining both programs:

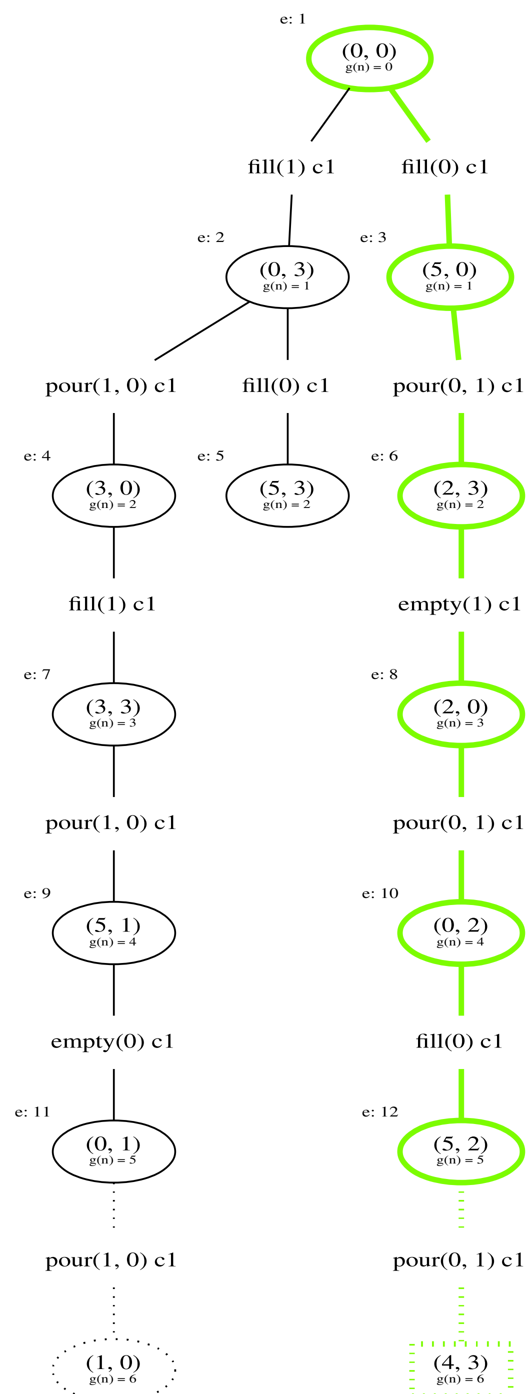
```
$ hlogedu-search run -a <algorithm> -p <problem> … | dot -K dot
  -T <format> -o <output_file>
```

**Note:** Although *graphviz* tools can cope with thousands of nodes, not all the image viewers will be able to render the generated images.

1 https://en.wikipedia.org/wiki/DOT_(graph_description_language)
2 http://www.graphviz.org/

# Interpreting a search tree image

- Box/Square shaped nodes represent goal states.
- Oval/Circle shaped nodes represent non-goal states.
- Nodes highlighted in Green are those in the solution “path”.
- Nodes with a solid border are those that have been expanded during the search.
- Nodes with a dotted border are those that were in the fringe at the end of the search.
- Nodes with a dashed border are those extracted from the fringe but not expanded yet.
- Edge labels are composed by the action, with its parameters, and its associated cost.
- Each node contains a textual representation of the state plus any of the following costs.
    - g(n): accumulated cost of the actions to reach the node.
    - h(n): heuristic value of the node’s state.
    - f(n): actual value used by the algorithm to the determine the extraction order. It depends on the algorithm, but typical options are: g(n), h(n), g(n) + h(n) or *path-max*.
- Labels “e: 1”, “e: 2”, etc., on the top left corner of the nodes indicate the order in which the nodes have been expanded.




The image on the right is the result of running the graph version of the BFS algorithm on the Jars problem.

We can clearly see all the nodes of the explored search space as well as the actions taken to transit from one to another. In this case each node contains one state of the problem and the accumulated cost of the actions [*g(n)*] to reach that state. Notice that, since BFS does not use a heuristic function h(n) and f(n) have been omitted.

If we look at the action label betweens two state nodes edge, we can see that all of them have the string *c1* at the end, this is the cost of the action, i.e., if the action cost had been 30 then the string at the end would be *c30*.

The states and the actions that lead to the solution are clearly identifiable and since the cost of the actions does not decrease and the algorithm is BFS, the given solution is optimal.

Finally, a small appreciation. The final node in the solution path has a dotted style, which means that it was in the fringe when the solution was found, thereby we can deduce that the “goal test” is performed when generating the successors. In other algorithms the final node may have a dashed style, which will mean that the solution is found when extracting the node from the fringe.

# Quick-start example of a custom problem

Even though the tool is able to load Python 3 code, it must follow a couple of structural rules so that it can be understood by the underlying execution engine. As a gentle introduction to some of the components to implement custom problems, we will see the implementation of the Jars problem.

The first step is `import` classes and functions, provided by the tool, to create a new Python class for our problem. This elements can be imported from *`hlogedu.search.problem`* and the basic ones you'll need are:

- `Problem`
- `Heuristic`
- `DRange / DDRange`
- `DInterval / DDInterval`
- `Categorical`
- `action`

Let us now define the initial structure of our Jars problem implementation.

```
from hlogedu.search.problem import action, Heuristic, Problem, DDRange

class JarsProblem(Problem):
  NAME = "Jars"
```

The first line imports all the structural elements that we will use to define the problem and then, we define our problem class *JarsProblem* that inherits from *Problem*. By default the tool will use the class name *JarsProblem* on the command line interface, to override this behavior use the *NAME* variable.

In our implementation of the Jars problem we will have two jars with capacities 5 l and 3 l, we can define this in the *__init__* of our class, as follows.

```
def __init__(self):
  super().__init__()
  self.capacities = (5, 3)
  self.n_jars = len(self.capacities)
```

Now that we have our problem initialized we **must** implement three methods used by the execution engine to start, end and validate the search states. These are, *get_start_states*, *is_goal_states* and *is_valid_state*.

```
def get_start_states(self):
  return [(0,) * self.n_jars]
```

This function returns a list with all the initial states, in our case we have only one start state that we represent with a tuple of integers, each integer represents how fill each jar is. In the code above we return a list with a tuple of *`self.n_jars`* zeros.
Note: if *`self.n_jars`* is 4 we return the list *[(0, 0, 0, 0)]*

```
def is_goal_state(self, state):
  return state[0] == 4
```

This one, as its name indicates, tests whether or not the given state is the goal state and returns True or False accordingly, in our case we consider as end any state that has 4 l in the first jar.

```
def is_valid_state(self, state):
  for i in range(self.n_jars):
    if not 0 <= state[0] <= self.capacities[0]:
      return False
  return True
```

Finally, this function tests if the provided state is valid. In our particular implementation it just tests that each jar is in the range [0, max capacity]. Later, we will see that if the actions are sound, this function can return True without evaluating the correctness of the state.

Now that we defined the basics of our problem, we can move to implementing the last piece, the actions. In this implementation we will have 3 actions: *fill*, *empty* and *pour*. The first two will fill/empty the jar passed as parameter and the last one will pour the content of one jar into the other, until either destination is full or the source is empty.

```
@action(DDRange(0, 'n_jars'), cost=1)
def fill(self, state, jar):
  if state[jar] == self.capacities[jar]:
    return None
  n_state = list(state)
  n_state[jar] = self.capacities[jar]
  return tuple(n_state)

@action(DDRange(0, 'n_jars'), cost=1)
def empty(self, state, jar):
  if state[jar] == 0:
    return None
  n_state = list(state)
  n_state[jar] = 0
  return tuple(n_state)

@action(DDRange(0, 'n_jars'), DDRange(0, 'n_jars'), cost=1)
def pour(self, state, jar_s, jar_d):
    cap_d = self.capacities[jar_d]
    if jar_s == jar_d or state[jar_s] == 0 or state[jar_d] == cap_d:
        return None
    to_pour = min(state[jar_s], cap_d – state[jar_d])
    n_state = list(state)
    n_state[jar_s] -= to_pour
    n_state[jar_d] += to_pour
    return tuple(n_state)
```

As you can appreciate in the methods above, all of them are decorated with `@action,` the first parameter of each function (ignoring the standard *self*) is the state to which the action is applied and all return new states. In addition to the state, each action can have as many parameters as necessary and their possible values must be specified in the `@action` decorator. In this example, all parameters represent jar indexes (the jar to fill/empty or source and destination jar when pouring) and their values are in the range [0, num. Jars].

The tool offers three different types to define the values of the action parameters, ranges, intervals and list of categorical values. The types that implement them are DRange/DDRange, DInterval/DDInterval and Categorical. In the example above, the parameters are represented with DDRanges, which start at 0 and end at `n_jars`, notice that the text `n_jars` matches the variable name `self.n_jars.` This is not accidental, DDRange and DDInterval replace the textual value with the value of a variable with the same name defined in `self`.

The last, non-mandatory, parameter of the `@action` decorator is the `cost` of the action, in our example we consider this cost to be always `1` for all the actions. Nonetheless, in some situations the cost of an action is affected by the state or its parameters, if this happens, the `cost` parameter have to be omitted and instead of just returning a new state, the method that implements the action must return a tuple *(cost, new state)*.

**jars.py**

```python
# -*- coding: utf-8 -*-

from hlogedu.search.problem import action, Heuristic, Problem, DDRange

class JarsProblem(Problem):
    """Jars problem
    This class implements the Jars problem. In this problem
    we have two or more jars with different capacities. At
    the beginning each jar is empty and the objective is to
    have a desired amount of liquid in one of them.
    The possible actions are:
        - fill one jar to its maximum capacity
        - empty one jar
        - pour from one jar into another, until the first is
          empty or the latter is full.
    """
    NAME = "Jars"

    def __init__(self):
        super().__init__()
        self.capacities = (5, 3)
        self.n_jars = len(self.capacities)

    def get_start_states(self):
        return [(0,) * self.n_jars]

    def is_goal_state(self, state):
        return state[0] == 4

    def is_valid_state(self, state):
        for i in range(self.n_jars):
            if not 0 <= state[0] <= self.capacities[0]:
                return False
        return True

    @action(DDRange(0, 'n_jars'), cost=1)
    def fill(self, state, jar):
        if state[jar] == self.capacities[jar]:
            return None
        n_state = list(state)
        n_state[jar] = self.capacities[jar]
        return tuple(n_state)

    @action(DDRange(0, 'n_jars'), cost=1)
    def empty(self, state, jar):
        if state[jar] == 0:
            return None
        n_state = list(state)
        n_state[jar] = 0
        return tuple(n_state)

    @action(DDRange(0, 'n_jars'), DDRange(0, 'n_jars'), cost=1)
    def pour(self, state, jar_s, jar_d):
        cap_d = self.capacities[jar_d]
        if jar_s == jar_d or state[jar_s] == 0 or state[jar_d] == cap_d:
            return None
        to_pour = min(state[jar_s], cap_d - state[jar_d])
        n_state = list(state)
        n_state[jar_s] -= to_pour
        n_state[jar_d] += to_pour
        return tuple(n_state)
```

# Actions

As seen in the previous example, actions are just Python methods that receive a state as the first parameter, ignoring the Python `self` idiom, decorated with `@action`, and return a new state or **`None`** if the action does not apply. This decorator helps the tool identify the actions and contains meta-information necessary to properly execute the search procedure. The most important meta-information are the possible values of any parameter that comes after the state, in the previous example we have seen some actions that require the index of the Jar to operate, but actions are not restricted to work with integer ranges. The next section will present the supported parameter types.

Another meta-information associated with an action is its cost, in the example above we see the cost specified at the end of the decorator via the `cost` keyword, but this is optional. In reality an action must return a pair where the first element is the cost and the second the new state. The thing is, many actions have constant `cost` and the `@action` decorator let us hide this implementation detail if the `cost` keyword is provided. Let us rewrite the `fill` action from the Jars example without the `cost` keyword:

```
@action(DDRange(0, 'n_jars'))
def fill(self, state, jar):
    if state[jar] == self.capacities[jar]:
        return None
    n_state = list(state)
    n_state[jar] = self.capacities[jar]
    return 1, tuple(n_state)
```

## More action parameter types

In the previous example we have seen actions with parameters that can be defined using DDRange. For other problems different parameters or combinations of them may be necessary, to cover different situations the tool provides the following types:

- **`DRange`**`(stop: int)` or **`DRange`**`(start: int, stop: int)`

  This kind of range mimics the built-in Python range function and it can be constructed with one or two integers. The version with one integer starts at *0* and ends at *stop-1* and the version with two integers starts at *start* and ends at *stop-1*. A parameter with this type will take values in the range [*start*, *stop*)

- **`DDRange`**`(stop: int/str)` or **`DDRange`**`(start: int/str, stop: int/str)`

  Basically the same as DRange but, in addition to integers it lets *start* and *stop* take textual values, like `'n_jars'`, which will be replaced by the value of the variable with the same name defined in the problem instance, i.e., `self.`

- **`DInterval`**`(stop: int)` or **`DInterval`**`(start: int, stop: int)`

  Like a DRange but it defines the range [*start*, *stop*]

- **`DDInterval`**`(stop: int/str)` or **`DDInterval`**`(start: int/str, stop: int/str)`

  A DInterval with the same support for textual values that has DDRange.

- **`Categorical`**`(values: Iterable)`

  A list of values with no specific order, as an example, a parameter that takes temperature values as `'freezing'`, `'cold'`, `'mild'` and `'hot'`, can be defined as:

  ```
  Categorical(['freezing', 'cold', 'mild', 'hot'])
  ```

# Built-in algorithms

To perform the search on any given problem, this tool comes with some built-in, blind and informed, search algorithms. Additionally, all the algorithms are implemented following two schemes: graph and tree search. The name used in the command-line interface to refer to them makes it clear which algorithm and scheme the different implementations refer to.

- graph-bfs
- tree-bfs
- graph-dfs
- tree-dfs
- graph-ucs
- tree-ucs
- graph-best-h
- tree-best-h
- graph-astar
- tree-astar

Two important aspects of these implementations are, the goal test position within the algorithm and the order in which the nodes are extracted form the fringe.

- In all the built-in algorithms, the goal test is performed when generating the successors on the blind search algorithms, and after extracting the node from the fringe on the informed search algorithms.
- In all the built-in algorithms, nodes are extracted from the fringe in lexicographical order. Bear in mind, this lexicographical order is affected by the state representation and can be modified by overriding the Python operators “<”, “<=”, “>”, “>=” and “==”.