SD314: Miniproject – FP in Python

Author : Christophe Garion <christophe.garion@isae-supaero.fr>
Audience : SUPAERO 3A SD/ARO
Date : 07/01/16

Abstract

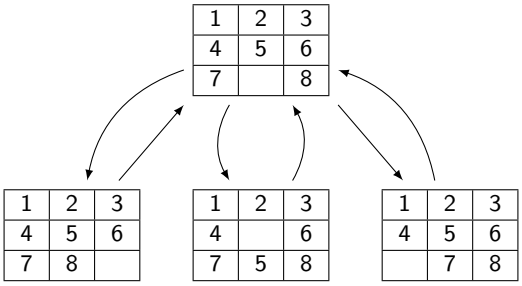
The goals of this mini-project is to understand how the functional programming paradigm allows to easily solve complex problems and to implement in Python (in functional programming style) a mini solver when space search can be modelled by a graph.

Acknowledgements: this mini-project is an adaptation of the first part of the final project proposed by a MOOC on functional programming with OCaml by Université Paris Diderot [1]. If you are interested in functional programming, I strongly recommend to attend this MOOC.

1 Presentation

When you want to write a program that solves a game, the first idea is to build the search space corresponding to the game. Some games have simple search space, because for instance they do not have many possible moves and because the space search can be reduced to a tree. Take for instance Tic-Tac-Toe: when you have put a "X" or a "O" in a case, you cannot undo your move.

Some games are a little bit difficult though. Take Jigsaw for instance. When you have deplaced a piece, you can undo your move on the next step. The search space is modelled by a graph. Here is a simple example with a Jigsaw 3×3 puzzle:



The goal of this mini-project is to create a simple solver when the search space can be modelled by a graph.

2 Questions

Your code must only use functional programming concepts: recursion, higher-order functions etc. In particular, you are not allowed to use **`for`** loops. Try also to use **`map`**, **`reduce`** and **`filter`** functions as often as possible.

1. write a function `loop` such that `loop(p, f, x)` returns `x` when `p(x)` is `True` and `loop(p, f, x)` returns `loop(p, f, f(x))` otherwise.
2. write a function `exists` such that `exists(p, l)` returns `True` when there is an element of list `l` such that `p(l)` is true.
3. write a function `find` such that `find(p, l)` returns `x` if `x` is the first element of `l` such that `p(x)` is `True`. When no such element exists, simply return `None`.
4. we now consider that the possible states of the game we are interested in are represented by a set E. We have a relation $\mathcal{R}$ such that $x\mathcal{R}y$ if and only if the state y is reachable in *one* step from x in the game. As a relation is a subset of $E \times E$, we will consider that $\mathcal{R}$ will be implemented in Python by a function `rel` that takes an element and returns a list of elements of the same type such that for all `y` in `rel(x)` $x\mathcal{R}y$ holds.

   We will consider here a simple relation on integers: $x\mathcal{N}y$ holds if and only if the difference between x and y is at most 2. This relation will be used to illustrate the concepts in the following. You can also use this relation for testing purposes for instance.

   Implement $\mathcal{N}$ by a `near` function. For instance `near(2)` should return the list `[0, 1, 2, 3, 4]`.

5. a relation $\mathcal{R}$ allows to compute the states reachable from a state in one step. We want now to be able to compute the states reachable from a list of states in one step.

   Write a function `flat_map` such that `flat_map(rel, l)` where `l` is a list of possible states returns the list of all possible states reachable in one step from the states of `l`. For instance, `flat_map(near, [2, 3, 4])` should return `[0, 1, 2, 3, 4, 1, 2, 3, 4, 5, 2, 3, 4, 5, 6]`.

6. write a function `iter(rel, n)` that iterates the relation `rel` `n` times. What should this function return?

7. the next step in solving the game is to compute the transitive closure of $\mathcal{R}$: you will obtain all the possible states from a given state. Of course, you will not compute the transitive closure of $\mathcal{R}$ as it may be infinite: we want here to iterate $\mathcal{R}$ starting from a given state and stop when we have a reachable state that verifies a given property (for instance to be the winning state...).

   Write a function `solve(rel, p, x)` such that it computes the iterations of `rel` starting initially from `x` until it reaches an element `y` such that `p(y)` is `True`. It then returns `y`.

   For instance, `solve(near, lambda x: x == 12, 0)` should return 12.

8. we not only want to have `y`, but also the complete path starting from `x` that goes to `y`. Write a function `solve_path(rel, p, x)` that returns the list of intermediate elements starting from `x` and finishing to `y`. For instance, `solve_path(near, lambda x: x == 12, 0)` should return `[0, 2, 4, 6, 8, 10, 12]`. You should use `solve` with the correct arguments to write `solve_path`.

At this point, your solver is not really efficient as you introduce a lot of redundancies in the search process. You may continue the mini-project by using sets instead of lists in a first time.

3 Deliverables

You should send for **01/29/16** a Python source file answering the previous questions. **The Python source file must be named `solve-bX.sql` where `bX` is your binom number.** Do not write a separate report, simply put your comments or detailed answers as Python comments in the source file.

The file will be send by email to christophe.garion@isae-supaero.fr and the email subject must be **`[SD314] Python project`**.

References

[1] Roberto Di Cosmo, Yann Regis-Gianas, and Ralf Treinen. *Introduction to functional programming in OCaml*. 2015. URL: https://www.france-universite-numerique-mooc.fr/courses/parisdiderot/56002/session01/about.

License CC BY-NC-SA 3.0