Comparing Functional and Imperative Programming Languages on Classical Planning Problems

DE-AN HUANG (DAHUANG), Stanford University

ACM Reference Format:
De-An Huang (dahuang). 2017. Comparing Functional and Imperative Programming Languages on Classical Planning Problems. 1, 1 (December 2017), 5 pages. https://doi.org/0000001.0000001_2

1 SUMMARY

In this project, we compare imperative (Python) and functional (Haskell) programming languages on solving the automated planning problems. Functional language is useful since the classical planning problem can be reduced to boolean satisfiability problem (SAT), where functional language and recursion provide a natural way of implementation. On the other hand, large-scale planning problems also suffer from the curse of dimensionality, where the efficiency of implementation plays a more important role. In this case, we might prefer imperative language. The goal of the project is thus to investigate the tradeoffs of functional and imperative programming on planning problems of different scales. Our results show that: (i) Recursion is a more natural way for classical planning, and thus functional programming is preferred qualitatively for implementation. (ii) On the other hand, iterative implementation is required for large-scale planning problem, where imperative language is preferred in terms of quantitative results. Furthermore, we explore the use of planning heuristics to alleviate the curse of dimensionality at scale and enable the functional language implementation to deal with large-scale planning.

2 BACKGROUND

In this section, I discuss the relevant background to support the motivation of the project. First, I introduce the definition of classical planning problem and its relation to boolean satisfiability problem (SAT). This provides background for understanding the usefulness of functional language in this context. Next, I discuss the curse of dimensionality in large-scale planning problem to support the importance of efficiency.

In addition to the programming languages that implement the *planning algorithms* (Python and Haskell), it is important to note that the *planning problems* are also specified in programming languages. More specifically, Planning Domain Definition Language (PDDL) is a standard programming language for specifying the classical planning problems. All the planning problems in this project are defined in PDDL. We use PDDL parsers in Python and Haskell to obtain the same set of planning problems for a fair comparison. We then apply planning algorithms in Python and Haskell to solve the parsed planning problems. We thus provide a brief description of PDDL in this section.

Author's address: De-An Huang (dahuang), Stanford University, dahuang@stanford.edu.

2.1 Classical Planning Problem

We follow the definition in [Ghallab et al. 2004]. A *classical planning problem* is defined by

- s_0: the initial state.
- g: the goal.
- $O = \{o\}$: the set of operators.

Each operator $o \in O$ is a triple defined by:

$$o = (name(o), precondition(o), postcondition(o)).$$

When the precondition of operator o is satisfied in the current state s, then we can apply the operator to the state, which will bring us to a new state s' depending on the postcondition/effect of o. The goal of planning is thus to find a valid sequence of operators o that can transform the initial state s_0 to a state s_g that satisfy the goal g.

2.2 Planning as Satisfiability.

This discussion is mainly to connect classical planning problem to the SAT we learned from Assignment 5, and thus provide the context why functional programming can be preferable for solving planning problems. As shown in [Kautz et al. 1992], the above planning problem can be reduced to a SAT problem. The key idea is to represent states as propositional variables the relation between two consecutive states as a propositional formula. In this way, we can synthesize a propositional formula that is satisfiable if and only if there is a plan. From this point of view, the classical planning problem is similar to SAT and Datalog query problem we learned from the class, both of which we have learned that functional programming provides a more elegant way of implementation.

2.3 Curse of Dimensionality of Planning

While the classical planning problem is general, its set of states induce a state space that has a size that is exponential to the number of elements in the set. And the complexity can be high for solving these larger scale problems. In this case, the efficiency is an important aspect for the algorithm execution. While functional programming provides a more natural way of analytically solving the classical planning problem, imperative programming provides more efficiency and flexibility in implementation. The goal of the project is thus to explore the tradeoffs between the two programming paradigms on varying complexities of planning problems.

2.4 Planning Domain Definition Language (PDDL)

While the main goal of the project is comparing programming languages for *planning algorithms*, the *planning problems* we are solving are themselves specified by programming language. Essentially, PDDL provides a unified way of writing the states (used for initial

ALGORITHM 1: Forward State Space Search - Recursive

```
Input: Initial state $s_0$ and goal $g$
Output: Plan $\pi$ achieving $g$ from $s_0$, or $NULL$ if infeasible
Heap fringe= Heapify([(Heuristic($s_0$), $s_0$)]) ;
Set seen = $\emptyset$ ;
return ForwardSearch(g, fringe, seen)
function ForwardSearch (g, fringe, seen)
    $h_s$, $s$ = fringe.heappop() ;
    if $s$ is NULL then
        return NULL
    else if $s$ satisfy $g$ then
        return $\pi(s)$
    else if $s \in$ seen then
        return ForwardSearch(g, fringe, seen)
    else
        seen.add($s$) ;
        for $s' \in$ Successor(s) do
            if $s' \notin$ seen then
                fringe.heappush((Heuristic($s'$), $s'$))
            end
        end
        return ForwardSearch(g, fringe, seen)
    end
end
```

state and goal), and also the conditions in pre/post-conditions of the operator. In PDDL, the classical planning problem is split into two files: *domain* file and *problem* file. The domain file contains $O = \{o\}$ the set of operators, and the problem file contains the initial state s_0 and the goal g. In this case, the operators in the domain file can be easily reused for different pairs of initial state and goal. More details will be discussed in Section 3.2

3 APPROACH

The goal of the project is to compare imperative and functional languages on solving classical planning problems at various scales. This includes two components: (i) planning algorithm implementation, (ii) planning problem specification. For a fair comparison, we need to make sure these two components are identical in both programming languages. For the planning algorithm, we implement the forward state space search algorithm (details in Section 3.1) in both Python and Haskell. For the second part, we specified the planning problems of various scales in PDDL and implement PDDL parsers in both Python and Haskell to ensure that they are solving the same planning problems. In addition, we discuss the quantitative evaluation metrics and their implementation for comparing these two programming languages.

3.1 Planning Algorithm - Forward State Space Search

We implement the forward state space search [Ghallab et al. 2004] algorithm that serves as the backbone for many state-of-the-art algorithms. The outline is shown in Algorithm 1. The key component is the **ForwardSearch** function, which is recursively called to continue the search. We can see how functional language can be a natural way for its implementation. In addition, the **Successor** function check the applicable operators and apply the operator to the current state to get the successors. This function is similar to the `new_facts_for_rule` of DataLog in Assignment 5, and we have learned from Assignment 5 that functional programming is also helpful for implementing it. Examples of my implementation of forward state space search in both Python and Haskell will be shown in Section 4 for qualitative discussion of implementing with both languages.

Another important function to note is the **Heuristic**() function, which is where state-of-the-art planning algorithms improve upon this backbone searching algorithm. The baseline heuristic in our main results is uniform (same for all states s), which provides no preference on the order of the state to continue the search. On the other hand, if we have a good heuristic that can estimate how close a state s is to the goal g, then we can prioritize them with **Heuristic**() in the heap, and therefore achieve better planning efficiency. We explore the use of heuristics to improve the efficiency of functional language in Section 5.

3.2 Planning Problem - Blocks World

Another important aspect of this project is to provide identical planning problems at various scales to compare the performance of functional and imperative language. Without enough scale of the problem, it is hard to really understand the requirement for efficiency. More specifically, we select the *BlocksWorld* planning problem that has been widely used as an example for automated planning [Russell et al. 1995]. The goal of BlocksWorld is to pick-and-place the blocks on each other or on the table to achieve the desired configuration. In addition, one property of BlocksWorld that is important to the project is that the size of its state space grows exponentially with respect to the number of blocks. This allows us to generate large-scale planning problems suffering the curse of dimensionality to compare the programming languages.

For implementation, we use Planning Domain Definition Language (PDDL) to specify the BlocksWorld problems ranging from 3 blocks to 9 blocks generated automatically using the Haskell source code of the Hierarchical PDDL work [Alford et al. 2009]. As mentioned in Section 2.4, the PDDL specification of planning problem is separated into two files: domain file and problem file. Here is an example of part of the blocks world PDDL domain file:

```
(define (domain blocks)
  (:requirements :strips :disjunctive-preconditions)
  (:types BLOCK)
  (:predicates
    (hand-empty)
    (clear ?b - BLOCK)
    (holding ?b - BLOCK)
    (on ?top - BLOCK ?bottom - BLOCK)
    (on-table ?b - BLOCK))
  (:action putdown
    :parameters (?b - BLOCK)
    :precondition (holding ?b)
    :effect (and
      (hand-empty) (not (holding ?b))
      (on-table ?b) (clear ?b)))
```

It can be seen that it specifies the actions/operators that are reusable for all our BlocksWorld planning problem, and this file is thus shared across all the planning problems we use in this project.

On the other hand, the problem file specifies the initial state and goal specific to the planning problem. Example of blocks world PDDL problem file with 2 blocks:

```
(define
 (problem pfile_2)
 (:domain blocks)
 (:objects b1 b2 - BLOCK)
(:init (hand-empty) (clear b2) (on-table b1) (on b2 b1))
 (:goal (and (clear b2) (on-table b1) (on b2 b1))))
```

We use PDDL parser in both Python and Haskell to ensure that they are solving the same set of planning problems for a fair comparison.

3.3 Starter Codes

The main starter codes I used are for the PDDL parser.

- Python: I used the pddl-lib https://pypi.python.org/pypi/pddlpy/0.1.9 in Python to parse the PDDL planning domain and problem. However, their parsed objects are not directly comparable for me to apply the forward state space search algorithm, so I implemented a wrapper to transform their results to a graph that is feasible for forward state space search and comparison to the Haskell implementation.
- Haskell: The closest I can find to a PDDL parser in Haskell is the huff package https://hackage.haskell.org/package/huff-0.1.0.1 where they parse a PDDL-like script in part of the code. As their PDDL-like language is equivalent to PDDL for the BlocksWorld, I implemented a translator from PDDL to their PDDL-like script in Python so the huff package in Haskell can parse the same planning problem as the ones in Python.

For the backbone of planning algorithm implementation I used the following reference code, but have to rewrite the core planning algorithm part for comparing Haskell and Python:

- Python: I referred to https://github.com/garydoranjr/pyddl when implementing the iterative version of the forward state space search algorithm (more details in Section 4.3). For the recursive version, I just implement Algorithm 1 on the output of pddl-lib.
- Haskell: The huff package already includes its own planner (with Fast-Forward Algorithm). However, as it is not the same and not comparable to the forward state space search in Python, I rewrote the core part of the planning algorithm, while using the code outline of the huff package.

As mentioned in Section 3.2, I used the source code of the Hierarchical PDDL work [Alford et al. 2009] (https://github.com/ronwalf/HTN-Translation) to generate the BlocksWorld problems of different number of blocks.

4 RESULTS

Our goal is to compare imperative and functional language on the forward state space search algorithm at different scales. While we have seen that the functional programming language provides a more natural way to implement the algorithm, we want to see what is the tradeoffs we have at large scale. We observe as expected that Haskell is a better option for the recursive implementation both quantitatively and qualitatively. However, iterative search actually has a better performance at large-scale, and imperative programming with the program state is easier for iterative search implementation.

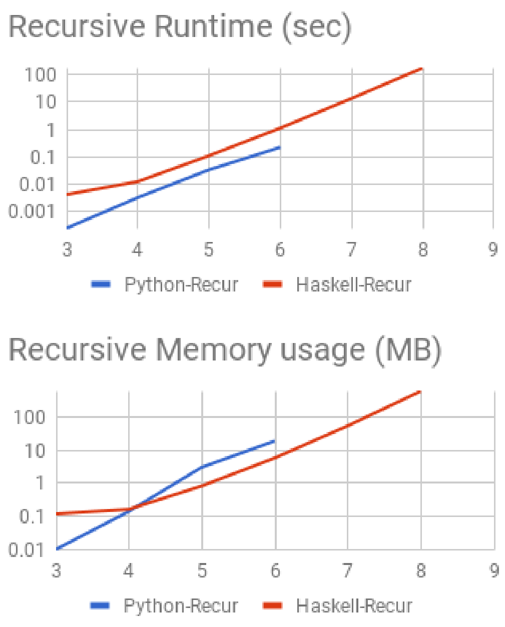


Fig. 1. Runtime and memory usage comparison of Python and Haskell recursion forward state space search algorithm. Python cannot finish planning problem beyond 6 blocks because of stack overflow, and Haskell cannot finish 9 blocks problem due to too long runtime.

4.1 Evaluation Metrics

For quantitative comparison of the programming languages, we use two metrics: (i) runtime and (ii) memory usage. Both of which are standard metrics that we can get by profiling tools. More specifically, we use `memory_profiler` for Python and RTS option for Haskell for profiling. We did not use lines of code for quantitative comparison. Instead, we provide qualitative discussion on the languages. In addition, as we will see in Section 4.2, Python and Haskell treat recursion very differently at scale.

4.2 Comparing Recursive Foward Search

Now we compare quantitatively the implementation of Forward State Space Search in Algorithm 1 in Python and Haskell. The memory usage and runtime comparison are shown in Figure 1. The horizontal axis is the number of blocks and the vertical axis is in log-scale. It can be seen that the two have similar memory usage, but Python is more efficient when the number of blocks is small. However, it is important to note that the Python implementation cannot deal with more than 6 blocks because the required levels of

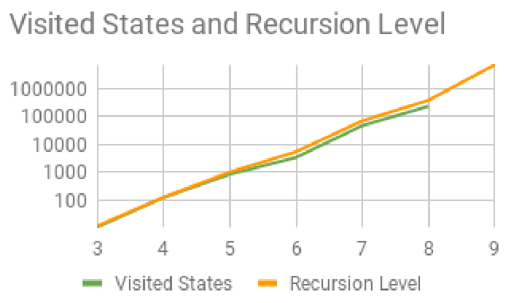


Fig. 2. Number of visited states and the levels of recursion required to find the plan for different numbers of blocks with uniform forward state space search. We can see that there indeed exists the challenge of curse of dimensionality.

recursion will lead to stack overflow. It is interesting to find that Python doesn't optimize tail recursion, and unbridled recursion causes stack overflows easily. Note that I already altered the system recursion limit of Python to enable results of 5 and 6 blocks. In addition, the Haskell implementation also fails to complete the 9 blocks problem (taking too long). We can see that both the runtime and the memory usage for both languages follow the exponential growth. The shows that curse of dimensionality indeed exists for the BlocksWorld planning problem. We further verify this by showing the number of visited states and recursion levels required to find the plan in Figure 2. With 5 blocks, the levels of recursion already exceed the default recursion limit value (1000) in Python.

Based on the results in this section, we can see that Haskell is indeed much better for recursive implementation, while Python is not really designed to handle large-scale recursion. On the other hand, Python recursion offers slightly better efficiency when the levels of recursion do not lead to stack overflow.

Overall, using Forward State Space Search with uniform heuristic function, we can see that both of the functional and imperative languages suffer from the curse of dimensionality and cannot even finish running all of the planning problems. Although we do see that functional programming is a better fit for recursive implementation. Next, we explore ways of improving the efficiency at large-scale to overcome the curse of dimensionality in both the imperative language and the functional language

4.3 Improving Imperative Language with Iterative Search

One important observation about the forward state space search in Algorithm 1 is that it is not required to be implemented recursively. While recursion is a more intuitive way of implementing this algorithm, we can also implement it iteratively. The pseudo-code is shown in Algorithm 2. While the main body of the algorithm is the same as the recursive search, we replace the recursion with a `while True` iteration that updates the state of the heap "fringe" and the set of seen states. In this case, it is not required to use recursion in imperative language, which Python is not really optimized for.

The memory usage and runtime comparing to the recursive implementation of Python and Haskell are shown in Figure 3. We can see

ALGORITHM 2: Forward State Space Search - Iterative

```
Input: Initial state $s_0$ and goal $g$
Output: Plan $\pi$ achieving $g$ from $s_0$, or $NULL$ if infeasible
Heap fringe= Heapify([(Heuristic($s_0$), $s_0$)]) ;
Set seen = $\emptyset$ ;
while True do
    $h_s$, $s$ = fringe.heappop() ;
    if $s$ is NULL then
        return NULL
    end
    if $s$ satisfy $g$ then
        return $\pi(s)$
    end
    if $s \notin$ seen then
        seen.add($s$) ;
        for $s' \in$ Successor($s$) do
            if $s' \notin$ seen then
                fringe.heappush((Heuristic($s'$), $s'$))
            end
        end
    end
end
```

that this significantly improve both the runtime and the memory usage compared to both recursive implementations. More importantly, now the planner is able to finish the 9 blocks planning problem. On the other hand, one interesting observation is that the recursive implementation in Haskell actually has good performance on memory usage and is comparable to the iterative search in Python. This is in contrast to recursive implementation in Python, which can only deal with 6 blocks, and uses more memory compared to iterative implementation.

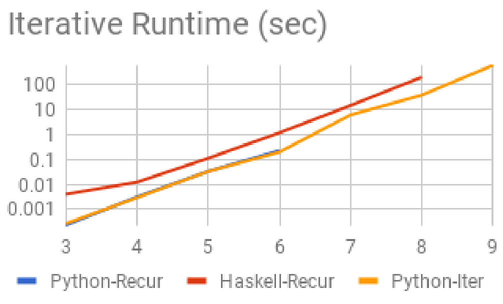
It is important to note that it is less straightforward to implement iterative search in Haskell and functional language in general. The iterative forward state space search requires the program to maintain the "state" of the heap "fringe" and the set of seen states, and continue to update the state of the program for each iteration. On the other hand, functional language like Haskell is naturally stateless, and the `while True` operation is harder to implement. Actually one way of implementing the while loop in Haskell is through recursion[1]. However, this defeats our purpose of avoiding recursion to deal with large-scale planning problem.

4.4 Qualitative Comparison

We provide additional qualitative discussion on implementing recursive forward state space search algorithm in Python and Haskell. The Python implementation of **ForwardSearch** is essentially the same as the one in Algorithm 1. On the other hand, Haskell allows the implementation to be more concise. First, we are able to use pattern matching to easily consider all the possible cases of s popped out from the heap to simplify the recursion implementation. Second, the most complicated case, where we have to check all the successors s' of the state s can be more easily written as:

```
ForwardSearch g (HS.insert s seen)
```

[1]https://stackoverflow.com/questions/17719620/while-loop-in-haskell-with-a-condition



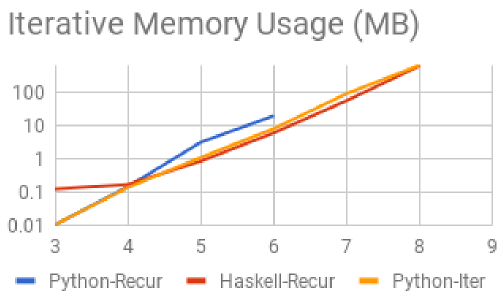


Fig. 3. Runtime and memory usage comparison of iterative forward search in Python. Iterative search significantly improve the performance of Python. Note that iterative implementation is not really appliable to Haskell.

```
(foldr Heap.insert fringe unseen_successors)
```

5 EXPLORING EFFECT OF PLANNING HEURISTICS

We have shown that iterative forward search can improve the performance of imperative language at large-scale. Now we explore other approaches for improving the performance at scale. As discussed in Section 3.1, designing stronger heuristic can improve the performance of forward state space search algorithm. We thus explore the effect of adding in the Relaxed Graphplan heuristic [Hoffmann and Nebel 2001] to our forward state space search algorithm. Given a state *s*, the Relaxed Graphplan heuristic uses a procedure similar to the original forward state space search, but when applying the operators to the states, it does not apply the negative effect. In this case, the Relaxed Graphplan is guaranteed to reach the goal in polynomial time to serve as a good heuristic.

The runtime of adding in the Relaxed Graphplan heuristic to our Haskell implementation of recursive forward state space search is shown in Figure 4. It can be seen that this allow the Haskell implementation to have a closer performance to the Python implementation. Note that we used existing Relaxed Graphplan implementation in Haskell for this part, and did not implement the Relaxed Graphplan heuristic in Python.

6 CONCLUSION

We have compared functional (Haskell) and imperative (Python) programming languages on solving the classical planning problems. We showed that functional programming is better for implementing the recursive search algorithm that naturally represents the planning problem. However, for the large-scale planning problems that suffer the curse of dimensionality, iterative implementation by imperative language has better performance. Finally, we explore the use of Relaxed Graphplan heuristic in the recursive forward state space search algorithm and show that it is able to have comparable performance with the iterative search implementation in Python.

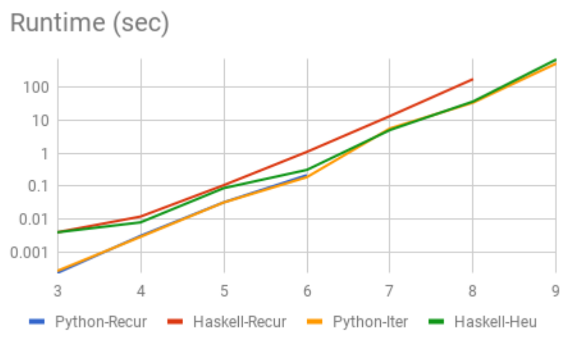


Fig. 4. Runtime comparison of different forward state space search implementation. The iterative search in Python (Python-Iter) has the best performance. Haskell implementation with Relaxed Graphplan Heuristic (Haskell-Heu) also achieved comparable performance with recursive search at large-scale

7 WORK DISTRIBUTION

I am the only one on the project, so I did all the work.

REFERENCES

Ronald Alford, Ugur Kuter, and Dana S Nau. 2009. Translating HTNs to PDDL: A Small Amount of Domain Knowledge Can Go a Long Way.. In *IJCAI*. 1629–1634.

Malik Ghallab, Dana Nau, and Paolo Traverso. 2004. *Automated Planning: theory and practice.* Elsevier.

Jörg Hoffmann and Bernhard Nebel. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research* 14 (2001), 253–302.

Henry A Kautz, Bart Selman, et al. 1992. Planning as Satisfiability.. In *ECAI*, Vol. 92. 359–363.

Stuart Russell, Peter Norvig, and Artificial Intelligence. 1995. A modern approach. *Artificial Intelligence. Prentice-Hall, Egnlewood Cliffs* 25 (1995), 27.