Improving the performance of the puzzle "Escape from Zurg" by applying pruning methods

Francisco Noya
fnoya2@illinois.edu

Abstract

Reducing the search space is critical for search programs that have to deal with spaces that tend to grow exponentially. Functional programming can produce elegant algorithms that can benefit from features of modern functional languages such as lazy data structures. Using the "Escape from Zurg" puzzle, I converted the original exhaustive search algorithm into a more efficient version by means of two pruning methods based on a lazy data structure called *improving sequence*. After applying these techniques the running time for finding the optimal solution was slashed by over 20 times and the amount of memory used was reduced by 95%.

I. Overview

The Haskell implementation presented in the original "Escape from Zurg" paper [2] constructs an exhaustive search space with all possible configurations of the elements of the problem. This is done by extending the search tree with a function that generates every alternative from each intermediate node. Each node (or state) is paired with the list of moves that leads to it. The state itself is represented by a tuple containing the position of the flashlight (L or R) and the elements (toys) that are still on the left side of the bridge (where Zurg is).

The construction of the search space takes advantage of Haskell list comprehension and lazy evaluation features to terminate without an explicit terminating condition. This space is subsequently filtered with the isSolution predicate to select those paths that lead to the goal within the given constraint ($duration \leq 60$).

This approach of solving the problem is clear and concise but it is also very inefficient. For example, with the original implementation and setup 108 paths are generated and evaluated but only two of them satisfy the duration constraint. To circumvent this problem, I decided to include a technique that would early prune those paths that will not lead to the goal because their partial durations are already over the threshold.

The objective of search problems of this kind is to find the least cost to reach the goal from an initial state. A common characteristic of these problems is that the cost function grows monotonically along the depth of the search tree. This property can be use for pruning. The cost of a certain path can be viewed as a function that by approximation reaches its final value. In many everyday tasks these approximating values are usually generated but rarely used. For example, to obtain the length of an array, the function uses an accumulator that gradually approaches the final result. If the intent of the programmer was to evaluate the expression 1<length[1..100] the length will inefficiently iterate one hundred times before returning the value 100.

Improving sequences are lazy data structures that consist of a monotonical sequence of approximating values that are gradually improved on the basis of some ordering relation as they approach the final result [3]. The idea behind *improving sequences* is that if an intermediate value of some expression has sufficient information to yield the result of an outer expression, further unnecessary computations are omitted. In the length example, only two iterations will be needed before returning the result of 1<length[1..100], instead of 100.

I applied this idea to the "Escape from Zurg" problem to prune the branches of the search tree whose intermediate durations were already over a threshold. In fact, I adapted the algorithm to find the paths of minimum duration or cost that solve the problem. The implementation was based on Morimoto et al. [4] and is summarized in the following section.

II. Implementation

Improving sequences are defined as the following sum data type in Haskell:

```
data IS a = a :? IS a | E deriving (Eq, Show)
```

The sequence consists of gradually improving values. The sequence could be infinite but, if finite, E represents the terminal symbol that indicates that the current value cannot be further improved. For example, $1 :? 2 :? 3 :? \mathsf{E}$ denotes a sequence in which 3 is the most improved value.

For example, a redefined length function that makes use of an improving sequence will have the following definition.

```
length :: [a] -> a -> IS a
length [] n = n :? E
length (x:xs) n = n :? IS.length xs (n+1)
```

Special binary relations are also needed to make the IS type useful.

```
(.<) :: Ord a => a -> IS a -> Bool
n .< E = False
n .< (x :? xs) = (n < x) || (n .< xs)
```

The advantages in efficiency of IS are obvious when measuring the time and memory needed to compare the length of a large list against 1.

```
*Zurg> s=[1..10000000000]

*Zurg> 1 < Prelude.length s
True
(162.96 secs, 720,000,052,040 bytes)

*Zurg> 1 .< IS.length s 0
True
(0.01 secs, 53,272 bytes)
```

Before applying IS to the "Escape from Zurg" puzzle I defined a new search function whose purpose is to find the shortest time (i.e. minimum duration) to achieve the goal. Without using IS this function has the following definition.

```
search :: ([m],s) -> Int
search (ms,state)
         | isGoal (ms,state) = cost (ms,state)
         | otherwise = minimum [search (moves : ms,ss) | (moves,ss) <- trans
            state ]
```

The cost function is just the cumulative duration of the moves required to attain the given state. The isGoal function is just the result of comparing the given state with the desired final state $(\mathsf{R}, [])$.

To use IS in this problem, we first need to define a new minimum function based on IS.

```
minimum :: Ord a => [IS a] -> IS a
minimum = foldl1 minB

minB :: Ord a => IS a -> IS a -> IS a
```

```
minB E ys = E
minB xs E = E
minB xxs@(x :? xs) yys@(y :? ys)
      | x == y = x :? minB xs ys
      | x < y = x :? minB xs yys
      | otherwise = y :? minB xxs ys
```

The new searchIS function based on IS has the following form.

```
searchIS :: ([m],s) -> IS Int
searchIS (ms,state)
         | isGoal (ms,state)= cost (ms,state) :? E
         | otherwise = cost (ms,state) :? IS.minimum [searchIS (moves : ms, ss)
            | (moves,ss) <- trans state]
```

When evaluating the minimum cost between two search tree branches, the computations are pruned when the end of the IS with the minimal value is reached. States further along a branch that has an intermediate cost already higher than the final value of another path are not even explored saving time and resources. This turns out to be an implementation of Dijkstra's single-source shortest-path search algorithm [1].

In this implementation computations are only terminated when the end of an IS is reached. However, when the intermediate value of a branch is already higher than the current global minimum, we know that that branch can be discarded because it is not going to lead to the optimum. In this case, efficiency can be gained if the exploration is terminated earlier. By introducing an upper bound to the search function and rewriting the minimum function as a continuation, we can achieve this improvement.

```
searchD :: ([m],s) -> IS Int
searchD (ms, state) = searchD' (ms,state) (maxBound :? E)
  where
    searchD' (ms,state) u
             | isGoal (ms,state) = cost (ms,state) :? E
             | otherwise = cost (ms,state) :? IS.minimumD u [searchD' (moves :
                ms, ss) | (moves,ss) <- trans state]

minimumD :: Ord a => IS a -> [IS a -> IS a] -> IS a
minimumD = foldl dfbb

dfbb :: Ord a => IS a -> (IS a -> IS a) -> IS a
dfbb u f = finalize (minB u (f u))

finalize :: Ord a => IS a -> IS a
finalize (x :? E) = x :? E
finalize (x :? xs) = finalize xs
```

The finalize function just returns the most improved value of the IS. The global minimum is wrapped in a IS of the form x :?E where x is the current global minimum and as such is fed into minB for further comparisons.

This is an implementation of the Depth-First Branch-and-Bound search algorithm.

III. Tests

The three search algorithms (search, searchIS, and searchD) were applied to solve the original puzzle with 4 toys as well as with more difficult versions with 5 to 8 toys. The Table I summarizes the definitions for each toy: name and time that takes to cross the bridge.

The time and memory consumed on each run for each algorithm are presented in Table II.

TABLE I
DEFINITION OF TOYS USED IN THE EXPERIMENTS

Toy name	Time to cross
Buzz	5
Woody	10
Rex	20
Hamm	25
T1	11
T2	13
T3	14
T4	15

TABLE II
TIME AND MEMORY CONSUMED ON EACH RUN

No. of toys	search	searchIS	searchD
4	0.02s 0.6 Mb	0.01s 0.5 Mb	0.01s 0.5 Mb
5	0.16s 24 Mb	0.08s 12 Mb	0.05s 12 Mb
6	7.8s 2 Gb	1.6s 0.4 Gb	1.3s 0.4 Gb
7	1011s 288 Gb	115s 22 Gb	60s 23 Gb
8	ND	ND	5874s 1.582 Gb

ND: Not done.

IV. CONCLUSIONS

The use of *improving sequences* has shown to produce much better solutions for the original puzzle in terms of overall efficiency. This comes directly from the ability to prune branches of the search tree that are incapable of producing the goal at a lower cost. Functional programming produce concise and elegant code to represent these algorithms. In addition, Haskell lazy evaluation and pattern matching provide additional computation efficiency and code clarity.

V. CODE REPOSITORY

- https://github.com/fnoya/cs421-project

REFERENCES

[1] Dijkstra, E. W.. (1959). "A note on two problems in connexion with graphs". *Numerische Mathematik*, 1(1), 269–271. https://doi.org/10.1007/bf01386390

[2] Erwing, M. (2004). "Escape from Zurg: An Exercise in Logic Programming". *Journal of Functional Programming*, 14(3), 253-261.

[3] Iwasaki, H., Morimoto, T., & Takano, Y. (2011). "Pruning with improving sequences in lazy functional programs". *Higher-order and Symbolic Computation* (formerly LISP and Symbolic Computation), 24(4), 281–309. https://doi.org/10.1007/s10990-012-9086-3

[4] Morimoto, T., Takano, Y., & Iwasaki, H. (2006). "Instantly Turning a Naïve Exhaustive Search into Three Efficient Searches with Pruning". *In* Lecture Notes in Computer Science (pp. 65–79). https://doi.org/10.1007/978-3-540-69611-7_4