Algorithm Design with the Selection Monad

Johannes Hartmann and Jeremy Gibbons

Department of Computer Science, University of Oxford, UK
firstname.lastname@cs.ox.ac.uk

Abstract. The selection monad has proven useful for modelling exhaustive search algorithms. It is well studied in the area of game theory as an elegant way of expressing algorithms that calculate optimal plays for sequential games with perfect information; composition of moves is modeled as a 'product' of selection functions. This paper aims to expand the application of the selection monad to other classes of algorithms. The structure used to describe exhaustive search problems can easily be applied to greedy algorithms; with some changes to the product function, the behaviour of the selection monad can be changed from an exhaustive search behaviour to a greedy one. This enables an algorithm design framework in which the behaviour of the algorithm can be exchanged modularly by using different product functions.

Keywords: Selection monad · Functional programming · Algorithm design · Greedy algorithms · Monads.

1 Introduction

In 2010 Martín Escardó and Paulo Oliva first describe the selection monad in their paper *Selection Functions, Bar Recursion, and Backward Induction* [2], where they explain bar recursion in terms of sequential games. They use the selection monad to model bar recursion and further show how sequential games can be solved using the selection monad. In subsequent work [3], they relate the selection monad to several different applications, such as Double-Negation Shift in the field of logic and proof theory and the Tychonoff Theorem in the field of topology.

Selecting a candidate for a solution out of a collection of options, based on some property function that tells us how good a candidate is, is a recurring pattern in computer science. We can use selection functions of type `(A -> R) -> A` to model this decision process. For example, when playing a sequential game, making a move means deciding for a particular move out of all possible moves. We can imagine a property function that somehow is able to compute how good each move is, and we then select the best one. The general idea when using selection functions is to model a problem in terms of a list of selection functions `[(A -> R) -> A]`. These selection functions then can be combined into a single selection function `([A] -> R) -> [A]` with the help of the product for selection functions. Further, it turns out that these selection functions form a monad, and

that the product for selection functions is given by the monadic `sequence` function. The product for selection functions hereby models an exhaustive search algorithm, that tries out every possible solution to the problem and then selecting the overall best one. This can be applied to find optimal strategies for sequential games, where each individual selection function is modeling the choice of which move to play next, and the sequential composition of all this individual choices computes an optimal strategy for this game.

In this paper we describe alternative product implementations for selection functions, providing different computational behaviour. With these different product implementations we are able to model both greedy algorithms and limited-lookahead search algorithms. This enables us to extend the application of selection functions to a wider range of problems.

In Section 2 we introduce selection functions in greater detail and provide an intuition for and examples of the product for selection functions. Then in Section 3 we provide an alternative product implementation that models a greedy behaviour, and in Section 4 we will have a look at some examples of greedy algorithms that are modeled with the greedy product for selection functions. In Section 6 we introduce another variant of the product of selection function that limits the number of steps it looks into the future. We conclude this paper and discuss potential future work in Section 8.

2 Selection functions

Selection functions summarise the process of selecting an individual object out of a collection of objects. Selecting something means making a choice, deciding in favour of one object over all the other objects of the collection we are choosing from. So in order to make a choice we need to be able to judge these objects, and based on that judgement, decide on a particular object out of a set of all possibilities.

As an example, let's say we want to buy a car. Therefore we have a collection of all available cars on the market, and we want to buy the best one. "Best" in this case depends heavily on our personal perspective on cars, so let's define "best" as "fastest". Then our personal selection process would be: look up top speeds of every car in our collection of all available cars, and select the one with the highest top speed. We can model this in the functional programming language Haskell as follows:

```
myChoice :: Car
myChoice = maxWith allCars topSpeed

maxWith :: Ord b => [a] -> (a -> b) -> a
topSpeed :: Car -> Int
allCars :: [Car]
```

Here, `maxWith` is the function that selects an element from the given list to maximise the given property function. Of course, top speed is not the only property

one might take into account when buying a car; `maxWith` abstracts from the property used for judging.

Given elements of type `A` and properties of type `R`, a selection function takes a property function `p :: A -> R` for judging elements, and selects some element of type `A` that is optimal according to the given property function. A good example of such a selection function is the `maxWith` from above, partially applied to a collection of objects, or the complementary `minWith` function. Escardó and Oliva studied these selection functions and connected them to different research areas, including game theory and proof theory [3].

2.1 Pairs of selection functions

As a next step, we want to look at how to combine two selection functions into a new bigger selection function. We will call this *pairing* of selection functions. To be consistent with the notation of previous literature we first define a type synonym for selection functions:

```
type J r a = (a -> r) -> a
```

One way of thinking of a function of type `J R A` is as embodying a nonempty collection of `A` elements: when told how to judge individual elements by `R`-values, the function can deliver an optimal element by that measure.

Now we define the `pair` operator combining two selection functions to make a new one [3]:

```
pair :: J r a -> J r b -> J r (a,b)
pair f g p = (a,b)
    where
        a = f (\x -> p(x,g(\y -> p(x,y))))
        b = g (\y -> p(a,y))
```

This new selection function selects `(A,B)` pairs, and therefore awaits a property function `p :: (A,B) -> R` that judges pairs. Suppose we are given such property function; then an `(A,B)` pair needs to be produced as output. To do so, we need to extract an `A` out of the first selection function `f`; we do so by constructing a new property function that judges `A`s. However, we only have something that judges `(A,B)` pairs. The trick is, for every `x :: A` we would like to judge, we extract a corresponding `y :: B` out of the second selection function and the pair `(x,y)` with the given property function `p`. Once we have found our optimal `a`, we can use it in the same way to select a corresponding `b` out of the selection function `g` and returning both as a pair.

So intuitively, this `pair` operator for selection functions combines two given selection functions in a way that the resulting selection function produces a pair that is optimal for a property function that judges these pairs. The two elements of this pair are extracted from the given selection functions in a way that they are always judged in the context of a full pair. From the perspective of selection functions as embodying a collection of objects, the pair for two selection functions embodies the cartesian product of the two collections.

2.2 Password example

Let's consider the following example, where we use the product of selection functions to crack a secret password. This secret password consists of two different tokens, a secret number between 1–9 and a secret character between a–z:

```
type Password = (Int, Char)
```

We are also given a property function that tells us if a given password is correct. We will treat this property function as a black box and not depend on its implementation details:

```
p :: Password -> Bool
p (a,b) = a == 7 && b == 'p'
```

In order to crack this password we will now define two individual selection functions for the two different parts of the password:

```
selectInt :: Ord r => J r Int
selectInt p = maxWith p [1..9]

selectChar :: Ord r => J r Char
selectChar p = maxWith p ['a'..'z']
```

Note here that both selection functions require a property function `p` to judge `Int`s or `Char`s individually, which we don't have at this stage. However, building the pair with the above defined `pair` function will result in a combined selection function that we can apply our property function to, and it indeed calculates the correct solution:

```
> pair selectInt selectChar p
---> (7,'p')
```

(Recall that Haskell defines `False < True`, so this expression returns a pair that satisfies `p`, provided that any of the given pairs does so.)

Because the only way to judge each individual component is by the given property function that judges complete password pairs, the `pair` function needs to create new property functions for its selection functions that are able to judge individual objects in a broader context.

This example fits nicely with the intuition that each selection function already embodies a set of objects, and that the `pair` function builds the cartesian product of these two sets and judges each pair individually and returns an optimal pair.

2.3 Iterated product of selection functions

The next logical step is to expand this from pairs to n-ary products. Unfortunately the standard Haskell type system is too restrictive to allow arbitrarily typed products of selection functions, and therefore the closest we can get is combining a list of selection functions for a common element type into a single selection function [3]:

```
product :: [J r a] -> J r [a]
product []       _ = []
product (e : es) p = a : as
   where a  = e(\x -> p(x : product es (p . (x:))))
         as = product es (p . (a:))
```

This particular implementation of `product` behaves similarly to the previous `pair` function. Given a property function `p :: [A] -> R` that judges lists, this function iterates through the list of selection functions in order to extract a concrete object out of each of them and therefore building a property function that considers both all previous decisions and all future decisions.

Note that for each recursive call a new property function is created that prepends the current choice of object via `(p . (a:))`. Further, to extract an object out of the current selection function `e`, a property function is created that judges each element of the underlying set in context of all possible future choices by recursively calling `product` within the property function.

2.4 Dependently typed version

Note that in this implementation we are not using the more restrictive type to our advantage. In contrast to the tuples of the above defined `pair` operator, the choice of lists comes with two drawbacks. First, the elements of the list must all be of the same type and second, lists in Haskell can be of arbitrary length while tuples are always of a fixed size.

The above presented `product` implementation does not take advantage of the first drawback, and assumes the second restriction. It constructs property functions that always judge elements in context with previous choices and potential future choices, making the restriction that every element needs to be of the same type irrelevant and assumes that the given property function is able to judge lists of the exact same length as the given list of selection functions.

In a dependently typed language we would be able to type this particular `product` implementation with a more expressive type that allows heterogeneous lists that can contain elements of any type and also ensures that each list we are dealing with has the correct length. An example implementation with this more expressive type in the programming language Idris is not difficult to construct [6].

We will later use this less expressive type to our advantage when we have a look at greedy algorithms, where we omit lookahead into the future.

2.5 Extended password example

The previous password example can now easily be extended to make use of the `product` function. We now want to crack a password of type `String`. We know only that it contains characters from a–z and that its length is 8:

```
type Password = String
```

and we are given a property function as a black box again:

```
p :: String -> Bool
p x@[_,_,_,_,_,_,_,_] = x == "password"
p _                   = undefined
```

Note that our property function is only defined for `String`s of length 8.

We can now utilise the previous `selectChar` and construct a list that contains this selection function exactly 8 times, once for each character of the password:

```
es :: Ord r => [J r Char]
es = replicate 8 selectChar
```

Utilising the previous `product` function will calculate the correct solution:

```
> product es p
---> "password"
```

This example again fits nicely the previously described intuition, where each component of the solution can only be judged in context with the previous selections as well as all possible future selections. This is underlined by the fact that by design, the property function is only able to judge solutions of the correct length, while being undefined for inputs of different lengths. The absence of any exceptions strengthens our intuition, that individual objects are always judged in full context.

2.6 Selection functions form a monad

Further, Escardó and Oliva show [3] that the type of selection functions forms a strong monad. This strengthens the intuition that a selection function embodies a collection of elements.

Selection functions with a fixed property type can be made instance of the monad class with this bind function:

```
(>>=) :: J r a -> (a -> J r b) -> J r b
(>>=) e f = \p -> f (e (p . flip f p)) p
```

and this `return` function:

```
return :: a -> J r a
return a = \_ -> a
```

Intuitively, the monad instance takes care that the underlying set elements are always judged in context. For `e >>= f`, we first want to extract an object of type `A` out of the given `e`, and then use `f` to transform it into a `J R B`. To extract an object of type `A` out of `e` we need to build a new property function that judges each element by how well it produces objects of type `b` by incorporating `f` into the new property function.

With `J R A` being a monad, we can utilise the prelude's monad functions to our advantage. Escardó and Oliva claim [3] that their `product` implementation is equivalent to the monadic `sequence` function from the Haskell prelude. A proof of this is claim is given in the appendix.

2.7 History-dependent product of selection functions

As an extension to the previous product for selection functions, Escardó and Oliva introduced [3] a history-dependent version of the `product` function. It keeps track of previous decisions and allows therefore for more dynamic selection functions, that can base the set from which they select on previous decisions:

```
hProduct :: [a] -> [[a] -> J r a] -> J r [a]
hProduct h []       p = []
hProduct h (e : es) p = a : as
   where a  = (e h) (\x -> p(x : hProduct (h++[x]) es (p . (x:))))
         as = hProduct (h++[a]) es (p . (a:))
```

This history-dependent version proves useful when modeling sequential games, in which the moves available at a given stage of the game typically depend on the previously played moves. The application of the selection monad to sequential games is already widely studied [1–4, 7, 8].

2.8 Efficiency drawbacks of this implementation

The `pair`, `product`, and `hProduct` implementations introduced above combine individual selection functions into a single product selection function. In order to extract the necessary elements out of the individual selection functions, the required property functions are created so as to always consider previous choices as well as potential future choices. This models an exhaustive search behaviour, where every possible combination of objects is judged by a given property function and the overall favorable combination of objects is then chosen. This results in an exponential runtime with respect to the number of selection functions to be combined. This exponential runtime makes the application of the selection monad infeasible for computing solutions for many problems.

Several methods to cope with this runtime have been explored in the past. There are several approaches to avoid unnecessary computations. Firstly, the individual selection functions can be neatly constructed such that they prune the inspection of their elements in unnecessary cases. Concretely, if the cost value `R` by which each object is judged has an upper or lower bound, the search can be stopped once a solution reaches one of these bounds [5]. Additionally, one should try to make as little as possible use of the property function, as it always constructs all possible future objects to judge the current element in context. One concrete example for this can be to avoid the use of the property function completely if there is only one element to choose from.

Secondly, the different `product` variants themselves leave room for improvement. For example, for minimax algorithms [3], a different `hProduct` function could implement alpha-beta pruning, which is used in game theory to reduce the search space when calculating perfect plays for sequential games.

One final flaw with the current implementations of `product` and `hProduct` variants is that they perform a lot of redundant calculations. In particular, each starts with the first selection function in the list and calculates all possible

future choices for each internal element. Based on this information it chooses the object that leads to the best possible future outcome. However, continuing the recursion, it forgets that it already explored all possibilities from there and starts the computation all over again, leading to a lot of redundant computations.

For the remainder of this paper, we will focus on different implementations for the iterated product of selection functions, which will have different computational behaviour. This will enable us to efficiently express greedy algorithms and other algorithm classes as products of selection function, and further enable us to abstract the behaviour of the different algorithms into the `product` functions.

3 Greedy algorithms

The product of selection functions as we have seen so far models an exhaustive search algorithm, exploring all possible combination of objects in the underlying sets of the selection functions. In this section, we explore a different product for selection functions. By using the less expressive type of the `product` function to our advantage, we are able to modify it in a way that allows us to model a greedy algorithm behaviour as a product of selection functions.

We previously identified that by choosing lists as container type for the selection functions, we restrict all the selection functions in the container to be of the same type. Further, the Haskell list type does not impose any length restrictions on the input list of property function. In order to model a greedy behaviour we can use this more general type to our advantage.

Once we require the property function to be defined for partial solutions as well, we can define a new `greedyProduct` function that judges the underlying objects of the selection functions only in the context of previous choices without caring about potential future choices:

```
greedyProduct :: [J r a] -> J r [a]
greedyProduct []       p = []
greedyProduct (e : es) p = a : as
   where a  = e (\x -> p [x])
         as = greedyProduct es (p . (a:))
```

So iterating through the list of selection functions, we extract a value out of each selection function by building a new property function of type `A -> R` by converting each `A` into a singleton list and then applying our property function to it. In the recursive call we build a new property function that keeps track of the previous choices. Note that this definition differs from the `product` definition by omitting the recursive call inside the new property function that calculates all possible future choices. Further, we can also define the corresponding history-dependent version:

```
greedyHProduct :: [a] -> [[a] -> J r a] -> J r [a]
greedyHProduct h []       p = []
greedyHProduct h (e : es) p = a : as
```

```
  where a  = (e h) (\x -> p [x])
        as = greedyHProduct (h++[a]) es (p . (a:))
```

Judging the individual objects outside of a global context locally captures the essence of greedy algorithms. In general, algorithms are called "greedy" when they perform only a local optimisation at each step. We can now utilise these new products for selection functions to implement greedy algorithms. The general idea is to define each individual local step of the greedy algorithm as a selection function; then the greedy algorithm arises from building the greedy product of these selection functions. The greedy behaviour is thereby abstracted away into the product function, enabling us to describe greedy algorithms in a similar way form to exhaustive search algorithms.

4 Examples of greedy algorithms

In this section we look at some examples applying the new `greedyProduct` variants to solve problems in a greedy way.

4.1 Password example

To continue the password example from above in a greedy way, we require a more sophisticated property function, that is able to judge partial solutions. So we are now given the following property function, that returns the number of correctly guessed characters of the password, instead of simply a boolean:

```
p :: String -> Int
p = length . filter id . zipWith (==) "password"
```

We can reuse our previously defined selection functions:

```
es :: Ord r => [J r Char]
es = replicate 8 selectChar
```

And with the new property function we can utilise the `greedyProduct` which will calculate the correct solution:

```
> greedyProduct es p
--> "password"
```

With the new property function, we don't need to know all future possibilities when determining the correct character at the next position of the password. We just need to be aware of our previous choices and can then decide greedily for the character that maximises the property function.

4.2 Prim's algorithm

A textbook example for a greedy algorithm is Prim's algorithm for finding a minimal spanning tree for a graph [9]. Given a weighted, undirected graph, a minimal spanning tree is a subset of edges of the given graph that forms a tree and is minimal in its weight. The general idea of Prim's algorithm is to grow a tree by repeatedly greedily selecting the lightest edge extending the tree (that is, the lightest edge connected to the tree and not creating a cycle).

To implement Prim's algorithm in terms of selection functions, we first define a graph as a list of edges, and an edge as a triple of integers with the first two elements being the two nodes of the edge and the third element the weight of the edge:

```
type Node   = Int
type Weight = Int
type Edge   = (Node, Node, Weight)
type Graph  = [Edge]
```

Further, we then define a helper function that calculates if a node is part of a given graph:

```
nodeOf :: Graph -> Node -> Bool
nodeOf [] _           = False
nodeOf ((x,y,_):xs) n = n == x || n == y || nodeOf xs n
```

We can then define a function calculating the total weight of a given collection of edges:

```
p :: [Edge] -> Weight
p [] = 0
p ((_,_,x):xs) = x + p xs
```

Further, we then need a function that calculates all edges that can be added to the tree such that it stays a tree. An edge of the graph is a candidate exactly if one of its ends is already in the tree, and the other is not. In the initial case, where there is no tree existing, all edges are potential candidates.

```
getCandidates :: Graph -> [Edge] -> [Edge]
getCandidates g [] = g
getCandidates g h  = filter f g
  where
    f (x,y,_) = nodeOf h x && not (nodeOf h y) ||
                not (nodeOf h x) && nodeOf h y
```

To build our list of selection functions for a given graph, we want to select the lightest edge of all possible edges according to our property function. We also need exactly one fewer selection functions than there are nodes in the original graph. (where nodes is a function that returns all nodes of a given graph)

```
selectEdge :: Ord r => Graph -> [[Edge] -> J r Edge]
selectEdge g = replicate (length (nodes g) - 1) f
 where
  f x p = minWith p (getCandidates g x)
```

Now considering the following example graph:

```
exampleGraph :: Graph
exampleGraph = [(1,2,1),(2,3,5),(2,4,9),(4,5,20),(3,5,1)]
```

we can form the product of our selection functions with the `greedyProduct` function. Applying this to our property function we get a minimal spanning tree as a result:

```
greedyHProduct [] (selectEdge exampleGraph) p
---> [(1,2,1),(2,3,5),(3,5,1),(2,4,9)]
```

4.3 Greedy graph walking

In this example, we are given a directed weighted graph and a start node, and we want to walk a given number of steps in the graph and thereby minimise the cost of the path we walked. This example will illustrate the different computational behaviour of the greedy product in comparison to the normal product. We use the same representation of graphs and edges from the previous example, together with the property function that can also be used to calculate the cost of a path. We now define an `getCandidates` function that, given a graph and the path we already walked, calculates the possible next edges:

```
getCandidates :: Graph -> [Edge] -> [Edge]
getCandidates g []        = undefined
getCandidates g [(_,n,_)] = filter (\(x,_,_) -> x == n) g
getCandidates g (_:xs)    = getCandidates g xs
```

Note, that the `getCandidates` function is undefined for the empty path. Therefore we are required to have the start edge in the initial path. Next, we define the list of selection functions. Each selection function takes a history describing the path that was already walked on the graph, calculating all possible next edges and selecting the edge with the minimum cost according to the property function. For our example, we want to walk 3 steps on the graph and therefore replicate the selection function 3 times.

```
es :: Ord r => Graph -> [[Edge] -> J r Edge]
es g = replicate 3 f
    where f x p = minWith p (getCandidates g x)
```

Now consider the following graph, as shown in Figure 1.

```
exampleGraph :: Graph
exampleGraph = [(1,2,1),(2,3,1),(2,4,5),(3,5,10),
                (4,6,1), (6,7,2), (5,7,1)]
```

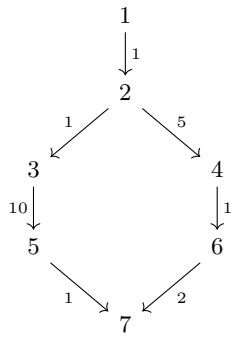

Fig. 1. Example Graph

To greedily walk a path on this graph, we can utilise the `greedyHProduct` with the initial edge `(1,2,1)` in the history and our selection functions applied to the `exampleGraph`. However, applying a greedy algorithm on this graph, locally choosing the best available edge at each stage, we won't get an optimal result:

```
greedyHProduct [(1,2,1)] (es exampleGraph) p
---> [(1,2,1),(2,3,1),(3,4,10),(5,7,1)]
```

In contrast, when using the normal product, we do achieve the optimal result (total cost 9 rather than 13):

```
hProduct [(1,2,1)] (es exampleGraph) p
---> [(1,2,1),(2,4,5),(4,6,1),(6,7,2)]
```

This example illustrates that for a graph walking algorithm to work, some insight about future edges is needed in order to calculate the correct result. When choosing the edge going out of node 2, we need to look further than the current edge to detect that going from node 2 to node 3 would lead to an overall worse outcome. That is, a greedy algorithm doesn't work.

5 Correctness

While the idea of a greedy algorithm is easy to grasp, proving that a greedy algorithm solves a given problem turns out to be quite difficult. If we view greedy algorithms in terms of selection functions, we can state that a greedy algorithm works if both the `greedyProduct` and the `product` functions calculate a result with the same cost:

```
p (greedyProduct selectFunc p) = p (product selectFunc p)    (1)
```

and further with the history dependent version with a initial history `h0`:

```
p (greedyHProduct h0 selectFunc p) = p (hProduct h0 selectFunc p)
                                                              (2)
```

In the case of the graph walking example, we can construct a counter example for which this property does not hold:

```
p (greedyHProduct [(1,2,1)] (es exampleGraph) p) ==
p (hProduct [(1,2,1)] (es exampleGraph) p)
---> False
p (hProduct [(1,2,1)] (es exampleGraph) p)
---> 9
p (greedyHProduct [(1,2,1)] (es exampleGraph) p)
---> 13
```

6 Limited lookahead

While greedy algorithms base their decision on the currently available options without considering the future, there are use cases where a limited lookahead into the future improves the result of an algorithm, without needing to go as far as exhaustive search. We can alter the product further to represent such a limited lookahead. To do so we introduce a limiting parameter to the product and distinguish the behaviour of the product depending on whether we reached the maximum lookahead depth. When building the property function for judging the individual underlying elements of a selection function, we decrement the lookahead depth in the recursive call.

```
limProduct :: Int -> [J r a]  -> J r [a]
limProduct i [] p              = []
limProduct i (e:es) p | i >  0 = a : as
                      | i <= 0 = [a']
    where
        a   = e (\x -> p (x : limProduct (i-1) es (p . (x:))))
        as  = limProduct i es (p . (a:))
        a'  = e (\x -> p [x])
```

Further, we can also introduce a history-dependent version with limited lookahead:

```
limHProduct :: Int -> [a] -> [[a] -> J r a] -> J r [a]
limHProduct i h [] p              = []
limHProduct i h (e:es) p | i >  0 = a : as
                         | i <= 0 = [a']
   where
       a  = e h (\x -> p (x : limHProduct (i-1) (h++[x]) es (p . (x:))))
       as = limHProduct i (h++[a]) es (p . (a:))
       a' = e h (\x -> p [x])
```

6.1 Graph Example

Recalling the greedy graph walking example from Section 4.3, the greedy algorithm is not able to produce optimal results for the given example graph in

Figure 1. The greedy algorithm is limited to local decisions, and therefore unable to detect an upcoming costly edge. Utilising the limited lookahead product `limHProduct` now with a lookahead of 1, we are able to detect the upcoming costly edge $3 \rightarrow 5$ and are able to calculate an optimal solution (total cost of 9) for this example graph:

```
shortestPathLimited = limHProduct 1 [(1,2,1)] (es exampleGraph) p
---> [(1,2,1),(2,4,5),(4,6,1),(6,7,2)]
```

This might work for this particular graph, there is no guaranty that limited lookahead can be utilised for arbitrary complex graphs. However, considering graphs where every split eventually converges again into a single node after at most n steps, a n-step lookahead is sufficient for our graph walking example. Such graph might look similar to Figure 2.

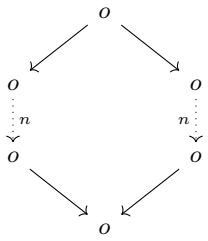

Fig. 2. Example limited lookahead graph

This example shows that having deeper insights about your problem can enable programmers to utilise limited lookahead algorithms. Another possible application of limited lookahead algorithms can be in the area of game theory. When, for example, implementing an AI opponent for chess, it is not computationally feasible to calculate a perfect game of chess. At some point, there needs to be a cutoff of the search space to ensure reasonably runtimes.

7 Iterated Product

A common pattern we can observe in the previous examples, is that we are always building a list of a fixed length containing copies of the same selection functions. Although not presented in this paper, there are applications for building products of different selection functions, one being the minimax algorithm for calculating solutions to sequential games. However, another different product for selection functions can be an iterated product where we iterate a given single selection function until we arrive at the desired result. Therefore given a predicate `pred :: [x] -> Bool` that tells us whether a given history is a final solution to our problem, we extract objects out of the given selection function until the predicate is satisfied:

```
iterate :: ([a] -> Bool) -> [a] -> J r a -> J r [a]
iterate pred h e p | not (pred h) = a : as
                   | otherwise    = []
    where
        a  = e (\x -> p(x : iterate pred (h++[x]) e (p . (x:))))
        as = iterate pred (h++[a]) e (p . (a:))
```

Further, we can also define iterated product for the history dependent product versions as well as the greedy product versions:

```
hIterate :: ([a] -> Bool) -> [a] -> ([a] -> J r a) -> J r [a]
hIterate pred h e p | not (pred h) = a : as
                    | otherwise    = []
    where
        a  = (e h) (\x -> p(x : hIterate pred (h++[x]) e (p . (x:))))
        as = hIterate pred (h++[a]) e (p . (a:))

greedyIterate :: ([a] -> Bool) -> [a] -> J r a -> J r [a]
greedyIterate pred h e p | not (pred h) = a : as
                         | otherwise    = []
    where
        a  = e (\x -> p [x])
        as = greedyIterate pred (h++[a]) e (p . (a:))

greedyHIterate :: ([a] -> Bool) -> [a] -> ([a] -> J r a) -> J r [a]
greedyHIterate pred h e p | not (pred h) = a : as
                          | otherwise    = []
    where
        a  = (e h) (\x -> p [x])
        as = greedyHIterate pred (h++[a]) e (p . (a:))
```

We can easily utilise this new iterated products to work with our previous examples:

7.1 Examples

Password Example Given that we already know that our password has a length of 8 characters, we can use the iterated product with a simple predicate that checks weather a given history reached the length of 8. Our selection function then is simply just the `selectChar` selection function:

```
iterate (\x -> length x == 8) [] selectChar p
---> "password"
greedyIterate (\x -> length x == 8) [] selectChar p
---> "password"
```

Graph walking Similar, we can utilise the iterated history dependent product to solve our graph walking problem. We first need to define our single selection function and then obtain the result by utilising the iterated product:

```
selectionFunction :: [Edge] -> J r Edge
selectionFunction x p = minWith p (getCandidates exampleGraph x)

hIterate (\h -> length h == 3) [(1,2,1)] selectionFunction p
---> [(1,2,1),(2,4,5),(4,6,1),(6,7,2)]
greedyHIterate (\h -> length h == 3) [(1,2,1)] selectionFunction p
---> [(1,2,1),(2,3,1),(3,4,10),(5,7,1)]
```

Prims Algorithm And similar to the graph walking, we can implement Prims algorithm with the iterated product:

```
selectionFunction :: [Edge] -> J r Edge
selectionFunction x p = minWith p (getCandidates exampleGraph x)

pred :: [Edge] -> Bool
pred h =  length h == (length exampleGraph - 1)
greedyHIterate pred [] selectionFunction p
---> [(1,2,1),(2,3,5),(3,5,1),(2,4,9)]
```

While these examples are straightforward, the true strength of the iterated product is that it can deal with solutions of different lengths. One example of this can be sequential games. There are many sequential games that have a specific ending condition rather than ending after a certain set of moves. This iterated product has the advantage of being able to deal with solutions of different lengths, while loosing the flexibility of building the product of a list of different selection functions. To solve this shortcoming we can build a iterated product that can deal with infinite lists:

```
infIterate :: ([a] -> Bool) -> [a] -> [J r a] -> J r [a]
infIterate pred h [] p                    = []
infIterate pred h (e:es) p | not (pred h) = a : as
                           | otherwise    = []
    where
        a  = e (\x -> p(x : infIterate pred (h++[x]) es (p . (x:))))
        as = infIterate pred (h++[a]) es (p . (a:))
```

One can easily adapt this product to also work history dependent or in a greedy way.

8 Conclusion and future work

We have seen that in addition to the already known monadic product for selection functions, there are other product implementations for selection functions, each capturing different computational behaviour. In particular with the above presented variations, the idea of describing problems as collections of selection functions can now be applied to problems that are solvable by greedy algorithms and limited lookahead algorithms. Moreover, the computational behaviour of an algorithm can be changed modularly by using different products for selection functions, while the problem description stays the same.

In addition to greedy products and limited lookahead products, there is the potential for more product implementations that behave differently. One example for this might be a product that is able to perform Alpha-Beta pruning minimax algorithms.

References

1. Bolt, J., Hedges, J., Zahn, P.: Sequential games and nondeterministic selection functions. arXiv preprint arXiv:1811.06810 (2018)
2. Escardó, M., Oliva, P.: Selection functions, bar recursion and backward induction. Math. Struct. Comput. Sci. **20**(2), 127–168 (2010)
3. Escardó, M., Oliva, P.: What sequential games, the tychonoff theorem and the double-negation shift have in common. In: Proceedings of the third ACM SIGPLAN workshop on Mathematically structured functional programming. pp. 21–32 (2010)
4. Escardó, M., Oliva, P.: Sequential games and optimal strategies. Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences **467**(2130), 1519–1545 (2011)
5. Hartmann, J.: Finding optimal strategies in sequential games with the novel selection monad. arXiv preprint arXiv:2105.12514 (2018)
6. Hartmann, J.: Dependently Typed Selection Monad (2022), https://github.com/IncredibleHannes/DependentlyTypedSelectionMonad
7. Hedges, J.: The selection monad as a cps transformation. arXiv preprint arXiv:1503.06061 (2015)
8. Hedges, J.M.: Towards compositional game theory. Ph.D. thesis, Queen Mary University of London (2016)
9. Prim, R.C.: Shortest connection networks and some generalizations. The Bell System Technical Journal **36**(6), 1389–1401 (1957)

Appendix

A Proof that product equals sequence

```
xm >>= \x -> sequence' xms >>= \xs -> return (x : xs)
-- {{ expand >>= definition }}
xm >>= \x -> (\p -> (\xs -> return (x : xs)) ((sequence' xms)
(p . flip (\xs -> return (x : xs)) p)) p)
-- {{ apply lambda }}
xm >>= \x -> (\p -> (return (x : ((sequence' xms)
(p . flip (\xs -> return (x : xs)) p))) ) p)
-- {{ resolve return }}
xm >>= \x -> (\p -> (x : ((sequence' xms) (p . flip (\xs -> return (x : xs)) p))))
-- {{ apply flip }}
xm >>= \x -> (\p -> (x : ((sequence' xms) (p . (\_ xs -> (x : xs)) p))))
-- {{ apply lambda }}
xm >>= \x -> (\p -> (x : ((sequence' xms) (p . (\xs -> (x : xs))))) )
-- {{ resolve function composition }}
xm >>= \x p -> (x : ((sequence' xms) (\xs -> p (x : xs))))
-- {{expand >>= definition  }}
\p' -> ( (\x p -> (x : ((sequence' xms) (\xs -> p (x : xs)))))
((xm) (p' . flip (\x p -> (x : ((sequence' xms) (\xs -> p (x : xs)))))  p')) p' )
-- {{ apply lambda }}
\p' -> ( (\x p -> (x : ((sequence' xms) (\xs -> p (x : xs)))))
(xm (\x -> p' (x : ((sequence' xms) (\xs -> p' (x : xs)))))) p' )
-- {{ apply lambda }}
\p' -> (\x -> (x : ((sequence' xms) (\xs -> p' (x : xs)))))
(xm (\x -> p' (x : ((sequence' xms) (\xs -> p' (x : xs))))))
-- {{ rewrite with where }}
\p' -> z : sequence' xms (\xs -> p' (z : xs))
           where z = xm (\x -> p' (x : ((sequence' xms)
           (\xs -> p' (x : xs)))))
-- {{ rewrite with where }}
\p' -> z : zs
           where z  = xm (\x -> p' (x : sequence' xms (p' . (x:))))
                 zs = sequence' xms (p' . (z:))
```