American University in Bulgaria

COS498: Artificial Intelligence and Haskell Programming

Final Report

Written by: Galina Endarova

Supervisor: Prof. Volin Karagiozov

# Contents:

Reports……………………………………………………………………….……..…………..3

Implementation of course algorithms………………………………………..………..…13

Missionaries and Cannibals…………………………………………………….….………..13

8 Queens………………………………………………………………………………………15

Travelling Salesman………………………………………………………………………….15

Functional programming assignment…………………………………………………....16

Code repository example in Haskell……………………………………………………...19

# Reports:

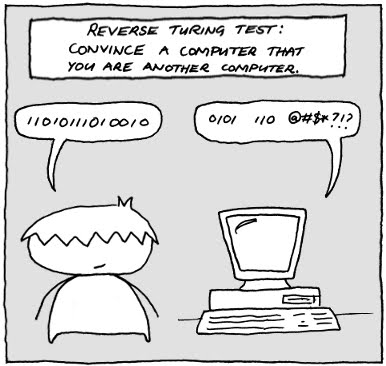
## Report 1:

Artificial Intelligence and its agents

Through the lectures provided for the Artificial Intelligence course here in AUBG and through separate readings I have gathered the following regarding the field:

History and current uses:

Artificial intelligence as a field started in 1950s, with attempts to model human intelligence in the emerging computer field. Problems with this goal started from the fact that despite human intelligence being a field to explore from antiquity, schools of thought do not agree on a single definition as to what intelligence means and how to model our thought processes. Currently the most popular one is that a rational person/agent can make decisions given a set of rules and initial data and make new rules out of the data observed in the real world i.e. has the capacity to learn.



In the field nowadays, a lot of AI is added to currently existing applications, but it is not called that, because too many applications would have the label and thus it would turn meaningless. Instead, AI is usually reserved for systems who can think independently about a number of subjects with no additional input. Examples of where AI finds application are health care systems, planning software, vision, translation machines, self –driving cars, etc. It is worth –noting that the field is booming and lucrative.[[1]](#footnote-1)

Agents:

At the core of the field is the idea of the agent – a rational one who can use his knowledge, logic and input from the outside to achieve his (externally – placed) goals. It has to be autonomous, able to interact with the surrounding environment and designed to be flexible, so that there are no mishaps like the crash of a Google-driven SUV into a bus in February[[2]](#footnote-2) or the racist Microsoft bot. [[3]](#footnote-3)

This provides several problems:

* How to break down the goal into actionable steps
* Which rules to put into the agent in order to help him complete the steps
* What data does the agent need and how to get it from the environment
* How do we measure whether the agent is performing correctly

There are several agent types: [[4]](#footnote-4)

* Simple – reflex – observe the environment and have a rule ready for that state which tells it what to do
* Model – based – keeps a model of the environment, how the agent’s actions would affect it and the rules for what the actions should be
* Goal – based – same as model based, only it has goals (desirable states of the environment) instead of rules, making it more flexible
* Utility – based – same as goal-based, only judges based on how useful the action would be to the agents, instead of on goals

There are also several types of environment:

* Discrete / continuous – chess vs. driving
* Fully observable / partially observable
* Static / dynamic
* Single agent / multiple agents
* Accessible / non-accessible
* Deterministic / non-deterministic – whether the next state is solely the responsibility of the agent or not
* Episodic / non-episodic – whether this state determines the next one

## Report 2:

Overview of Haskell

1. History

In the two years following the release of the first lazy functional language, Miranda, interest in lazy programming grew. As a result there were a dozen different such languages and no widely accepted standard for them. In 1987 at the conference on Functional Programming Languages participants agreed that there needs to be such an open standard and formed a committee to create it. As a result of this committee’s efforts Haskell emerged in 1990.

1. Type

Haskell is a strongly typed functional language with lazy evaluation – meaning that evaluation of an expression is done only when the expression is needed and hash consing is used.

1. Functionality

A lot could be said about how Haskell functions. A lot will be said in the next few reports, considering this is the language we have chosen to examine. This is why here I will stop on just a couple of the characteristics that made the biggest impression on me:

* Patterns – Haskell uses pattern matching in its functions
* factorial **::** ([**Integral**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#t:Integral) a) **=>** a **->** a
* factorial 0 **=** 1
* factorial n **=** n **\*** factorial (n **-** 1)
* Recursion everywhere – recursion is the standard way to resolve complex problems, which for me, coming from a background of imperative languages, was startling
* [**zip**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:zip) **::** [a] **->** [b] **->** [(a,b)]
* [**zip**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:zip) \_ [] **=** []
* [**zip**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:zip) [] \_ **=** []
* [**zip**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:zip) (x**:**xs) (y**:**ys) **=** (x,y)**:**[**zip**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:zip) xs ys
* How much easier certain algorithms are to implement
* quicksort **::** ([**Ord**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#t:Ord) a) **=>** [a] **->** [a]
* quicksort [] **=** []
* quicksort (x**:**xs) **=**
* **let** smallerSorted **=** quicksort [a **|** a **<-** xs, a **<=** x]
* biggerSorted **=** quicksort [a **|** a **<-** xs, a **>** x]
* **in**  smallerSorted **++** [x] **++** biggerSorted

1. Sources used

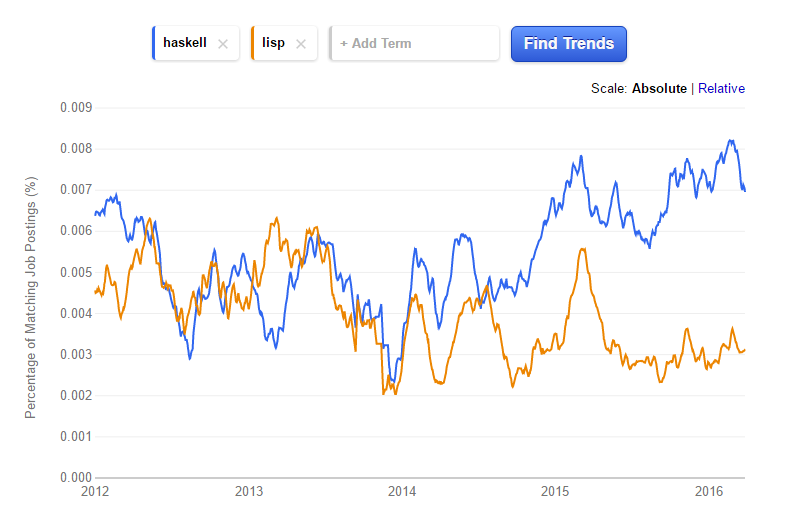
I have used a couple of sources to understand Haskell better:

* <http://learnyouahaskell.com/> - written by an informative Slovenian, this website was the easiest to help me get my bearings when first looking at Haskell
* <http://book.realworldhaskell.org/read/> - I did not manage to go through the whole website, but what I have seen has helped my general understanding

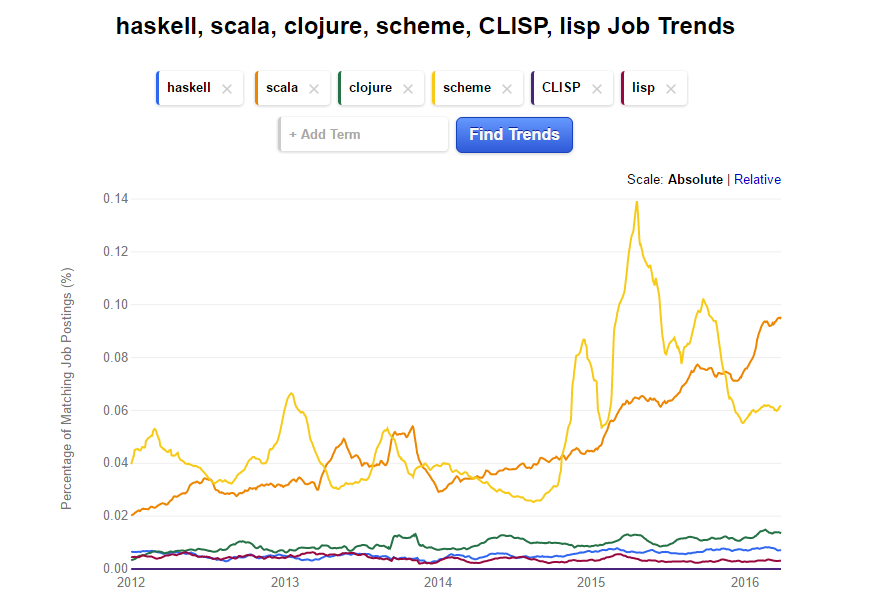
1. Comparison with LISP

Since this is just an intro into AI, not even into one of the languages, I cannot really do them justice by comparing them. However, as it is interesting to delve into the differences and read how programmers squabble over which is better, I did a little bit of research.

From that emerged that in the past 2 years jobs with marker Haskell have been slightly more common than jobs with marker LISP. [[5]](#footnote-5)



However, adding a couple of other languages/dialects, shows a more nuanced situation:



Clojure and Scheme are dialects of LISP, while Scala seems to be a wonderful multifunctional language. This being said I do not see the need to study Haskell instead of LISP, though certainly both are useful (especially for a novice in the world of functional languages).

I also found this article: <http://chrisdone.com/posts/haskell-lisp-philosophy-difference> which might be an interesting read.

## Report 3:

Solving problems by searching

Notes on algorithms and searching:

A typical search for a solution to a given problem would have 4 steps:

* Form goal
* Form problem in terms understandable for the agent which will solve it
* Search solution
* Execution phase

In order to solve the problem, the agent needs 4 things to be specified – the states that the environment can have, the steps that the agent can take, the goal test, which checks whether the current state matches the goal and the path cost of a step, which later is added to the total cost of the solution, on order to measure whether it is a good solution or a bad one.

As far as searching strategies go, there are the uninformed - depth-first, bread-first searches, uniform cost, bi-directional, iterative deepening – and informed – best first type and local search type. [[6]](#footnote-6) [[7]](#footnote-7)

Actual representation:

The states of a search algorithm are represented via a graph, with the each node being a state and each path being the operation needed to get to the state. In its essence, solving a problem by searching means following through the nodes in order to reach (and test) the goal state. Reaching a leaf node without finding the goal state means going back to the nearest fork in the tree and choosing the sibling of the node chosen before.

In Haskell it is a little tricky to represent a graph. Trees have their own library, but a normal-working graph with the required nodes and plenty of space to store the states attached to the nodes requires some tinkering with the language. [[8]](#footnote-8)

One of the ways to do it is to take advantage of the matrimonially – sounding concept Tying the knot.[[9]](#footnote-9) In order to represent a graph (a cyclic structure which can be traversed in a number of directions) the following example is given:

1. **data** DList a **=** DLNode (DList a) a (DList a)
3. mkDList **::** [a] **->** DList a
5. mkDList [] **=** [**error**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:error) "must have at least one element"
6. mkDList xs **=** **let** (first,[**last**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:last)) **=** go [**last**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:last) xs first[[10]](#footnote-10)
7. **in**  first
9. **where** go **::** DList a **->** [a] **->** DList a **->** (DList a, DList a)
10. go prev []     next **=** (next,prev)
11. go prev (x**:**xs) next **=** **let** this        **=** DLNode prev x rest
12. (rest,[**last**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:last)) **=** go this xs next
13. **in**  (this,[**last**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:last))
15. takeF **::** [**Integer**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#t:Integer) **->** DList a **->** [a]
16. takeF 0     \_                 **=** []
17. takeF (n**+**1) (DLNode \_ x next) **=** x **:** (takeF n next)
19. takeR **::** [**Show**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#t:Show) a **=>** [**Integer**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#t:Integer) **->** DList a **->** [a]
20. takeR 0     \_                 **=** []
21. takeR (n**+**1) (DLNode prev x \_) **=** x **:** (takeR n prev)

Calling the functions takeF and takeR return elements forward and backwards respectively.

Another way to define a graph is as a table of Vertexes (this is the way I have done it for some exercises:[[11]](#footnote-11)

1. **type** Vertex **=** [**Int**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#t:Int)
2. **type** Table a **=** Array Vertex a
3. **type** Graph e **=** Table [(e, Vertex)]
4. **type** Bounds  **=** (Vertex, Vertex)
5. **type** Edge e **=** (Vertex, e, Vertex)

Now, in this case, the vertex is an Integer. However, it can easily be substituted for a complex data type which can hold the state of an environment.

## Report 4:

Machine Learning

Machine learning is everywhere around us. As it is easier to talk when you have an example to work with, I took a look at health diagnostics.

For a machine to diagnose correctly an illness and learn from its mistakes, it needs three things:

* task on which to improve – correct diagnosis
* data – correct diagnoses with which it works in the beginning, then the diagnosis which it makes and checks as to whether they are correct
* measure of improvement – could be speed, accuracy, details of solution

When encountering a new example the machine would try to diagnose it. For such a machine the trial should be supervised, so that it gives a diagnose and then it receives feedback about what the correct solution is. Potential problems with such a scenario is that in the process of training the correct solution given to the machine might not be correct at all, leading to later mistakes. And, of course, there is no flawless machine, just as there is no flawless human.

This needs to be taken into account when designing the machine.

A machine learns by

* creating a function out of the test data. The function needs to be consistent with the data in order to predict following data.
* Constructing a decision tree – each internal node is an attribute for which we check the state of the environment, while the leafs are the decisions that the machine can reach

Applications:

* Pairing employees and jobs – Woo [[12]](#footnote-12)
* Predicting the next death in Game of Thrones [[13]](#footnote-13)
* Advanced prosthetics – a hand that learns more about how to act better the longer you use it[[14]](#footnote-14)
* Cancer treatment and diagnostics – current algorithms are as good as or better than practitioners[[15]](#footnote-15)

## Report 5:

Game Playing (Haskell)

These past two weeks I have looked at game strategies and game development using Haskell. Even though the creation of games using Haskell (or functional languages in general? ) is a pain because of UI, some studios and individuals have undertaken it with the goal of having more mathematical and robust games.[[16]](#footnote-16) [[17]](#footnote-17)

Keera Studios, for example, have created a helpful library for defining board games – gtk-helpers – which lives on github. Using it a person can create a peg solitaire game and it is easy to make other, more interactive examples for AI.

Neural Networks:

The other topic that I looked at is neural networks. The types examined were the feedforward and recurrent.

Feedforward networks were the first to be developed (middle of last century). There are two types: Single-layer perception and multilayer perception. The single layer consists of a single layer of outputs, which calculate inputs at each node and if a threshold is crossed they fire the result (typically 1). If not, they fire a specified value. They are limited in what can be created with them. For example, a 1969 book shows that they cannot implement the XOR function.[[18]](#footnote-18)

Multilayer networks differ in that there are intermediate layers between the input and the output layer. They have a non-linear activation function for all the nodes, meaning that a linear function cannot cover the weighted inputs of the input nodes. The typical activation functions used are sigmoids, which makes sense considering the need for the value to pass a threshold.

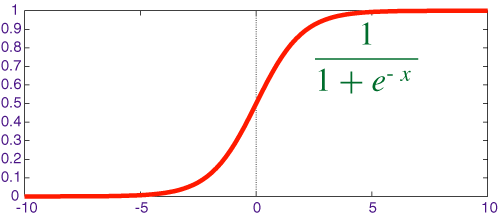


Figure Sigmoid Function

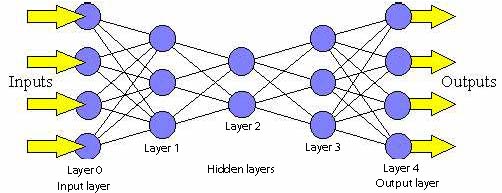


Figure Feedforward neural network

The other type I looked at was recurrent networks, called that because they introduce cycles into the computation. They were first developed in 1980s to address some shortcomings of feedforward networks (for example, recurrent networks can work on speech recognition, because they can go back to trying to solve the problem multiple times).They use internal memory to process inputs in whichever sequence is most convenient. The cycles are introduced to go back to various points in the hidden layer. [[19]](#footnote-19)

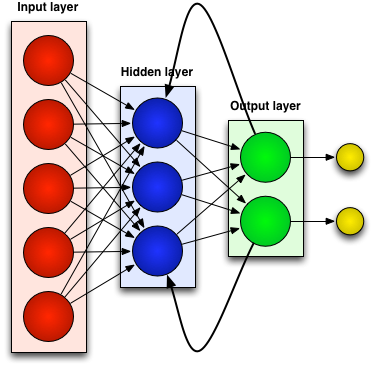


Figure Recurrent neural network

# Application of some of the algorithms:

## Missionaries and Cannibals

2. **module** AIMAMissionaries **where**
3. **import** Data.List(sort, nub, (**\\**))

6. **data** Person **=** Missionary **|** Cannibal
7. **deriving** ([**Ord**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#t:Ord), [**Eq**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#t:Eq), [**Show**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#t:Show))
9. **data** Position **=** LeftSide **|** RightSide
10. **deriving** ([**Eq**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#t:Eq), [**Show**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#t:Show))
12. **data** PState **=** PState {left **::** [Person], right **::** [Person], boat **::** Position}
13. **deriving** ([**Eq**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#t:Eq), [**Show**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#t:Show))
15. beginState **=** PState {left **=** [Missionary, Missionary, Missionary, Cannibal, Cannibal, Cannibal], right **=** [], boat **=** LeftSide}
16. goalState **=**  PState {left **=** [], right **=** [Missionary, Missionary, Missionary, Cannibal, Cannibal, Cannibal], boat **=** RightSide}
18. almostGoalState **=** PState {left **=** [Cannibal], right **=** [Missionary, Missionary, Missionary, Cannibal, Cannibal], boat **=** LeftSide}
19. almostGoalState2 **=** PState {left **=** [Cannibal, Missionary, Missionary], right **=** [Missionary, Cannibal, Cannibal], boat **=** LeftSide}


23. solution **::** [PState]
24. solution **=** idfs beginState 0
26. idfs **::** PState **->** [**Int**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#t:Int) **->** [PState]
27. idfs s n **=** **case** idfs' 0 n False s of
28. [] -> idfs s (n+1)
29. other -> other
30. where
31. idfs' **::** [**Int**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#t:Int) **->** [**Int**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#t:Int) **->** [**Bool**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#t:Bool) **->** PState **->** [PState]
32. idfs' m n True s = [s]
33. idfs' m n False s
34. **|** isGoalState s  **=** idfs' m n True s
35. | m==n           = []
36. | otherwise      = case dropWhile (==[]) $ map (idfs' (m**+**1) n False) (successors s) **of**
37. []     **->** []
38. (x**:**xs) **->** s **:** x
40. -- check if the state is a goal state
41. isGoalState **::** PState **->** [**Bool**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#t:Bool)
42. isGoalState **=** (**==** goalState)
44. -- filter legal states
45. successors **::** PState **->** [PState]
46. successors **=** [**filter**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:filter) isLegalState . allSucc
48. -- generate all states after applying all possible combinations
49. allSucc **::** PState **->** [PState]
50. allSucc s
51. **|** boat s **==** LeftSide **=** [**map**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:map) (updatePStateLeft s) (genCombs (left s))
52. **|** [**otherwise**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:otherwise)          **=** [**map**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:map) (updatePStateRight s) (genCombs (right s))
54. -- move a number of cannibals and missonaries to the right side
55. updatePStateLeft s p **=** **let** oldLeft **=** left s
56. oldRight **=** right s
57. **in** s {left **=** sort **$** oldLeft **\\** p,
58. right **=** sort **$** oldRight **++** p,
59. boat **=** RightSide
60. }
62. -- move a number of cannibals and missonaries to the left side
63. updatePStateRight s p **=** **let** oldLeft **=** left s
64. oldRight **=** right s
65. **in** s {left **=** sort **$** oldLeft **++** p,
66. right **=** sort **$** oldRight **\\** p,
67. boat **=** LeftSide
68. }
70. -- unique combinations
71. genCombs **::** [**Ord**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#t:Ord) a **=>** [a] **->** [[a]]
72. genCombs **=** nub . [**map**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:map) sort . genPerms
74. -- permutations of length 1 and 2
75. genPerms **::** [**Eq**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#t:Eq) a **=>** [a] **->** [[a]]
76. genPerms [] **=** []
77. genPerms (x**:**xs) **=** [x] **:** ([**map**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:map) (**:** [x]) xs) **++** genPerms xs
79. -- legal states are states with the number of cannibals equal or less
80. -- to the number of missionaries on one riverside (or sides with no missionaries)
81. isLegalState **::** PState **->** [**Bool**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#t:Bool)
82. isLegalState s **=** hasNoMoreCannibals (left s) **&&** hasNoMoreCannibals (right s)
83. **where** hasNoMoreCannibals lst **=** **let** lenMiss **=** [**length**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:length) ( [**filter**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:filter) (**==** Missionary) lst)
84. lenCann **=** [**length**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:length) ( [**filter**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:filter) (**==** Cannibal) lst)
85. **in** lenMiss **==** 0 **||** lenMiss **>=** lenCann [[20]](#footnote-20) [[21]](#footnote-21) [[22]](#footnote-22) [[23]](#footnote-23)

## 8 Queens Problem:

1. queens **::** [**Int**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#t:Int) **->** [[[**Int**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#t:Int)]]
2. queens n **=** [**map**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:map) [**reverse**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:reverse) **$** queens' n
3. where queens' 0       **=** [[]]
4. queens' k       = [q:qs | qs <- queens' (k**-**1), q **<-** [1..n], isSafe q qs]
5. isSafe   try qs **=** [**not**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:not) (try `[**elem**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:elem)` qs **||** sameDiag try qs)
6. sameDiag try qs **=** [**any**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:any) (**\**(colDist,q) **->** [**abs**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:abs) (try **-** q) **==** colDist) **$** [**zip**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:zip) [1..] qs

## Travelling salesman problem:

1. coordinates **::** [([**Float**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#t:Float),[**Float**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#t:Float))]
2. coordinates **=** [(0,1),(1,2),(2,3),(2,4),(0,6)]
4. distance **::** ([**Float**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#t:Float), [**Float**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#t:Float)) **->** ([**Float**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#t:Float), [**Float**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#t:Float)) **->** [**Float**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#t:Float)
5. distance (x1,y1) (x2,y2) **=** [**sqrt**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:sqrt) ((x1**-**x2)**^**2 **+** (y1**-**y2)**^**2)
7. tourLength **::** [([**Float**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#t:Float), [**Float**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#t:Float))] **->** [[**Int**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#t:Int)] **->** [**Float**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#t:Float)
8. tourLength pairs tour **=** **let** coords **=** [**map**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:map) (pairs **!!**) tour
9. **in** [**sum**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:sum) **$** [**zipWith**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:zipWith) distance coords ([**tail**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:tail) coords)
11. tourLength' :: [(Float, Float)] -> [Int] -> Float
12. tourLength' pairs tour **=** tourLength pairs (tour **++** ([**head**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:head) tour))
14. main **::** [**IO**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#t:IO) ()
15. main **=** [**print**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:print) **$** tourLength' coordinates [2,1,0,3,4][[24]](#footnote-24)

# Functional Programming Assignment:

1. Problem Description:

In this functional programming assignment I attempted to create an algorithm which counts the number of knights that can be places on a chess board. I attempted this as a modification of the standard 8 queens problem. The problem was a good starting place since it required some knowledge of Haskell and getting familiar with recursion and the language’s pattern matching features.

1. Development:

In the quest for developing the algorithm I browsed over numerous sources. Most of the knowledge gathered about recursion came for Learn you a Haskell for Great Good. Some problem solving techniques were borrowed from the 99 Haskell Problems.

In developing the solution there were two clear areas on which to work:

* Board representation
* Counting algorithm

As far as the board representation goes, I decided to create a custom datatype which holds a counter for the number of knights placed so far, a Matrix type list to hold the number of available places and a tupples list to hold the diagonals of the already place knights.

Algorithm:

1. Creates a board
2. Occupies the board by:
   1. Placing a knight in each consecutive place in the open space list
   2. Deleting that space form the available spaces
   3. Adding 1 to counter
   4. Adding the diagonals of that place to the diagonals list
   5. Proceeding with the next space until there are no more spaces in the list
3. Returns the counter

Code:

1. **import** Data.List
2. **data** Board **=** Board ([**Int**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#t:Int))([[[**Int**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#t:Int)]])([([**Int**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#t:Int),[**Int**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#t:Int))]) **deriving** [**Show**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#t:Show)
4. generate **::** Int**->** Board
5. generate 0 **=** Board (0)([[]])([])
6. generate n **=** Board (0)(replicate n [0..n**-**1])([])
8. --drops the nth element of a list (list starts counting from 1, not zero)
9. smallDrop **::** [**Int**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#t:Int) **->** [a] **->** [a]
10. smallDrop \_ [] **=** []
11. --returns the first n elements + the elements after the nth element
12. smallDrop n a **=** [**take**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:take) n a **++** [**drop**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:drop) (n**+**1) a
14. dropBomb **::** [[[**Int**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#t:Int)]] **->** [**Int**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#t:Int) **->** [**Int**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#t:Int) **->** [[[**Int**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#t:Int)]]
15. dropBomb [] \_ \_ **=** []
16. --first thing to check - if c is the last element of a, then
17. dropBomb a b c **=** **if** c **==** [**length**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:length) a **-**1
18. **then** [**init**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:init) a **++** [(smallDrop b ([**last**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:last) a))]
19. **else** **if** c **==** 0
20. **then** [(smallDrop b ([**head**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:head) a))] **++** [**tail**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:tail) a
21. --if c is in the middle, then it takes the first c-1 elements, drops some elements from the cth element and adds to that the elements after the c+1
22. **else** [**take**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:take) c a **++** [(smallDrop b (a**!!**c))] **++** [**drop**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:drop) (c**+**1) a
24. --placing a knight means:
25. --if there is a knight in the diagonals,
26. --we drop the coordinates from available coordinates
27. --if the diagonals are not in the coordinates,
28. --we add 1 to the number of knights on the board,
29. --we drop the coordinates from available coordinates and
30. --we update the diagonals
31. place **::** Board **->** ([**Int**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#t:Int), [**Int**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#t:Int)) **->** Board
32. place (Board count list diagonals) (b,c) **=**
33. **if** (b**+**c, b**-**c) `[**elem**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:elem)` diagonals
34. **then** Board (count)(dropBomb list b c)(diagonals)
35. **else** Board (count**+**1)(dropBomb list b c)((b**+**c, b**-**c)**:**diagonals)
37. --it needs to set the zise of the board and return how many knights it has
38. knight**::** [**Int**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#t:Int) **->** Board
39. knight n **=** occupy (generate n) 0 0

42. isEmptyAll **::** [[[**Int**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#t:Int)]] **->** [**Bool**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#t:Bool)
43. isEmptyAll a **=** **if** [**null**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:null) ([**last**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:last) a)
44. **then** True
45. **else** False
47. isEmptyRow **::** [[[**Int**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#t:Int)]] **->** [**Int**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#t:Int) **->** [**Bool**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#t:Bool)
48. isEmptyRow a c **=** **if** [**null**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:null) (a**!!**c)
49. **then** True
50. **else** False
52. --it needs to occupy the whole board
53. --if board is empty, it returns the board, so that we can get the counter
54. occupy **::** Board **->**[**Int**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#t:Int) **->** [**Int**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#t:Int) **->** Board
55. occupy (Board count [] diagonals) b c **=** (Board count [] diagonals)
56. occupy (Board count list diagonals) b c
57. **|**b **==** [**length**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:length) list **=** occupy (Board count list diagonals) 0 (c**+**1)
58. **|**c **==** [**length**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:length) list **=** (Board count list diagonals)
59. **|**isEmptyAll list **=** (Board count list diagonals)
60. **|**isEmptyRow list c **=** occupy (Board count list diagonals) 0 (c**+**1)
61. **|**[**otherwise**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:otherwise) **=** occupy (place (Board count list diagonals) (b,c)) (b**+**1) c

# Codebook Repository Example:

Out of the code examples given in the textbook I chose to modify the depth-first search algorithm that I have found in order to make sure it has no cycles. It is a standard searching algorithm and I thought delving into it would give me a better understanding of algorithm construction.

Code:

1. --representation of the graph
2. **type** Table a **=** Array Vertex a
3. **type** Graph **=** Table [Vertex]
4. vertices **::** Graph **->** [Vertex]
5. vertices **=** indices
6. **type** Edge **=** (Vertex,Vertex)
7. edges **::** Graph **->** [Edge]
8. edges g **=** [ (v,w) **|** v **<-** vertices g, w **<-** g**!**v]
10. --function for manipulation of the tables: returns a new table with the function used applied to each of its elements
11. mapT **::** (Vertex **->** a **->** b) **->** Table a **->** Table b
12. -- f is function, t is table
13. mapT f t **=** array (bounds t)
14. [(v, f v (t**!**v)) **|** v**<-**indices t]
16. --for easier work with arrays
17. **type** Bounds **=** (Vertex,Vertex)
19. --builds a table which holds the number of edges leaving each vertex
20. outdegree **::** Graph **->** Table [**Int**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#t:Int)
21. outdegree g **=** mapT numEdges g
22. **where** numEdges v ws **=** [**length**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:length) ws
24. --builds a graph from the list of edges
25. buildG **::** Bounds **->** [Edge] **->** Graph
26. buildG bnds es **=** accumArray ([**flip**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:flip) (**:**)) [] bnds es
28. transposeG **::** Graph **->** Graph
29. transposeG g **=** buildG (bounds g) (reverseE g)
30. reverseE **::** Graph **->** [Edge]
31. reverseE g **=** [ (w,v) **|** (v,w) **<-** edges g]
33. indegree **::** Graph **->** Table [**Int**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#t:Int)
34. indegree g **=** outdegree (transposeG g)
36. --need them later
37. **data** Tree a **=** Node a (Forest a)
38. **type** Forest a **=** [Tree a]
40. --returns all the vertices in the orginal order
41. dff **::** Graph **->** Forest Vertex
42. dff g **=** dfs g (vertices g)

45. preorder **::** Tree a **->** [a]
46. preorder (Node a ts) **=** [a] **++** preorderF ts
47. preorderF **::** Forest a **->** [a]
48. preorderF ts **=** [**concat**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:concat) ([**map**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:map) preorder ts)
50. --depth first order
51. preOrd **::** Graph **->** [Vertex]
52. preOrd g **=** preorderF (dff g)
54. --creates a list with the vertices ziped with numbers
55. tabulate **::** Bounds **->** [Vertex] **->** Table [**Int**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#t:Int)
56. tabulate bnds vs **=** array bnds ([**zip**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:zip) vs [1..])
58. preArr **::** Bounds **->** Forest Vertex **->** Table [**Int**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#t:Int)
59. preArr bnds ts **=** tabulate bnds (preorderF ts)
61. --places the trees in order descendants before ancestors and left before right
62. postorder **::** Tree a **->** [a]
63. postorder (Node a ts) **=** postorderF ts **++** [a]
64. postorderF **::** Forest a **->** [a]
65. postorderF ts **=** [**concat**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:concat) ([**map**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:map) postorder ts)
67. postOrd **::** Graph **->** [Vertex]
68. postOrd g **=** postorderF (dff g)
70. --soring to make sure there are no cycles
71. topSort **::** Graph **->** [Vertex]
72. topSort g **=** [**reverse**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:reverse) (postOrd g)
74. --takes the graph, makes a tree
75. components **::** Graph **->** Forest Vertex
76. components g **=** dff (undirected g)
78. --could duplicate edges
79. undirected **::** Graph **->** Graph
80. undirected g **=** buildG (bounds g) (edges g **++** reverseE g)
82. --double depth first search (Sharir)
83. scc **::** Graph **->** Forest Vertex
84. scc g **=** dfs (transposeG g) ([**reverse**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:reverse) (postOrd g))
86. --reverse the roles of original and transposed graphs
87. scc' :: Graph -> Forest Vertex
88. scc' g **=** dfs g ([**reverse**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:reverse) (postOrd (transposeG g)))
90. --given a graph and a node it bulds a tree of all the vertices reachable from the node
91. generate **::** Graph **->** Vertex **->** Tree Vertex
92. generate g v **=** Node v ([**map**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:map) (generate g) (g**!**v))

95. **type** Set s **=** MutArr s Vertex [**Bool**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#t:Bool)
97. mkEmpty **::** Bounds **->** ST s (Set s)
98. mkEmpty bnds **=** newArr bnds False
100. contains **::** Set s **->** Vertex **->** ST s [**Bool**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#t:Bool)
101. contains m v **=** readArr m v
103. include **::** Set s **->** Vertex **->** ST s ()
104. include m v **=** writeArr m v True
106. prune **::** Bounds **->** Forest Vertex **->** Forest Vertex
107. prune bnds ts **=** runST (mkEmpty bnds `thenST` **\**m **->** chop m ts)
109. --examine the root
110. --if it has been seen before - discard the whole tree
111. --if not, add the vertex to the set
112. --call chop on the descendants of the root
113. --then call chop on the remaining forest
114. chop**::**Set s **->** Forest Vertex**->** ST s (Forest Vertex)
115. chop m [] **=** returnST []
116. chop m (Node v ts **:** us) **=** contains m v `thenST` **\**visited **->** **if** visited
117. **then** chop m us
118. **else** include m v `thenST` **\**\_ **->** chop m ts `thenST` **\as** **->** chop m us `thenST` **\**bs **->** returnST ((Node v **as**) **:** bs)
119. --final function
120. dfs **::** Graph **->** [Vertex] **->** Forest Vertex
121. dfs g vs **=** prune (bounds g) ([**map**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:map) (generate g) vs)
123. reachable **::** Graph **->** Vertex **->** [Vertex]
124. reachable g v **=** preorderF (dfs g [v])
126. --checks for existence of a path between two vertices
127. path **::** Graph **->** Vertex **->** Vertex **->** [**Bool**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#t:Bool)
128. path g v w **=** w `[**elem**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:elem)` (reachable g v)
130. tree **::** Bounds **->** Forest Vertex **->** Graph
131. tree bnds ts **=** buildG bnds ([**concat**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:concat) ([**map**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:map) flat ts))
132. **where** flat (Node v ts) **=** [ (v,w) **|** Node w us **<-** ts] **++** [**concat**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:concat) ([**map**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:map) flat ts)
134. back **::** Graph **->** Table [**Int**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#t:Int) **->** Graph
135. back g post **=** mapT select g
136. **where** select v ws **=** [ w **|** w **<-** ws, post**!**v**<**post**!**w ]
138. cross **::** Graph **->** Table [**Int**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#t:Int) **->** Table [**Int**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#t:Int) **->** Graph
139. cross g pre post **=** mapT select g
140. **where** select v ws **=** [ w **|** w **<-** ws, post**!**v**>**post**!**w, pre**!**v**>**pre**!**w]
142. forward **::** Graph **->** Graph **->** Table [**Int**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#t:Int) **->** Graph
143. forward g tree pre **=** mapT select g
144. **where** select v ws **=** [ w **|** w **<-** ws, pre**!**v**<**pre**!**w] **\\** tree**!**v
146. bcc **::** Graph **->** Forest [Vertex]
147. bcc g **=** ([**concat**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:concat) . [**map**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:map) bicomps . [**map**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:map) (label g dnum)) forest
148. **where** forest **=** dff g
149. dnum **=** preArr (bounds g) forest
151. label **::** Graph **->** Table [**Int**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#t:Int) **->** Tree Vertex **->** Tree (Vertex,[**Int**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#t:Int),[**Int**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#t:Int))
152. label g dnum (Node v ts) **=** Node (v,dnum**!**v,lv) us
153. **where** us **=** [**map**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:map) (label g dnum) ts
154. lv **=** [**minimum**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:minimum) ([dnum**!**v]**++**[ dnum**!**w **|** w **<-** g**!**v] **++**[ lu **|** Node (u,dw,lu) xs **<-** us])
156. bicomps **::** Tree (Vertex,[**Int**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#t:Int),[**Int**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#t:Int)) **->** Forest [Vertex]
157. bicomps (Node (v,dv,lv) ts) **=** [ Node (v**:**vs) us **|** (l, Node vs us) **<-** [**map**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:map) collect ts]
159. collect **::** Tree (Vertex,[**Int**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#t:Int),[**Int**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#t:Int)) **->** ([**Int**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#t:Int), Tree [Vertex])
160. collect (Node (v,dv,lv) ts) **=** (lv, Node (v**:**vs) cs)
161. **where** collected **=** [**map**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:map) collect ts
162. vs **=** [**concat**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:concat) [ ws **|** (lw, Node ws us) **<-** collected, lw**<**dv]
163. cs **=** [**concat**](http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#v:concat) [ **if** lw**<**dv **then** us **else** [Node (v**:**ws) us] **|** (lw, Node ws us) **<-** collected][[25]](#footnote-25)

1. http://venturebeat.com/2016/04/14/artificial-intelligence-startup-digitalgenius-raises-4m-to-make-customer-service-agents-superhuman/ [↑](#footnote-ref-1)
2. http://www.latimes.com/local/lanow/la-me-ln-video-google-car-20160309-story.html [↑](#footnote-ref-2)
3. https://www.theguardian.com/technology/2016/mar/30/microsoft-racist-sexist-chatbot-twitter-drugs [↑](#footnote-ref-3)
4. http://www.tutorialspoint.com/artificial\_intelligence/artificial\_intelligence\_agents\_and\_environments.htm [↑](#footnote-ref-4)
5. http://www.indeed.com/jobtrends/q-haskell-q-lisp.html [↑](#footnote-ref-5)
6. http://www.imada.sdu.dk/~marco/DM828/Slides/dm828-lec2.pdf [↑](#footnote-ref-6)
7. http://www.tutorialspoint.com/artificial\_intelligence/artificial\_intelligence\_popular\_search\_algorithms.htm [↑](#footnote-ref-7)
8. http://web.engr.oregonstate.edu/~erwig/papers/InductiveGraphs\_JFP01.pdf [↑](#footnote-ref-8)
9. https://wiki.haskell.org/Tying\_the\_Knot [↑](#footnote-ref-9)
10. http://stackoverflow.com/questions/5844653/haskell-why-the-convention-to-name-a-helper-function-go [↑](#footnote-ref-10)
11. https://wiki.haskell.org/The\_Monad.Reader/Issue5/Practical\_Graph\_Handling [↑](#footnote-ref-11)
12. http://thenextweb.com/insider/2016/02/09/woo-lets-you-find-a-new-job-on-your-terms-anonymously/ [↑](#footnote-ref-12)
13. http://www.thelocal.de/20160422/munich-students-build-app-to-predict-game-of-thrones-deaths [↑](#footnote-ref-13)
14. http://tech.firstpost.com/news-analysis/scientists-create-machine-learning-robot-hand-313987.html [↑](#footnote-ref-14)
15. https://www.sciencedaily.com/releases/2016/04/160421133831.htm [↑](#footnote-ref-15)
16. http://keera.co.uk/blog/2013/03/19/creating-board-games-in-haskell/ [↑](#footnote-ref-16)
17. http://www.cs.uu.nl/research/techreps/repo/CS-2013/2013-003.pdf [↑](#footnote-ref-17)
18. https://en.wikipedia.org/wiki/Perceptrons\_(book) [↑](#footnote-ref-18)
19. http://karpathy.github.io/2015/05/21/rnn-effectiveness/ [↑](#footnote-ref-19)
20. https://gist.github.com/nattster/1518740 [↑](#footnote-ref-20)
21. http://www.lounge.se/wiki2/show/MissionariesCannibalsProblem [↑](#footnote-ref-21)
22. http://comments.gmane.org/gmane.comp.lang.haskell.cafe/64215 [↑](#footnote-ref-22)
23. http://enrico.kravina.net/code/en.index.php [↑](#footnote-ref-23)
24. http://illicittech.blogspot.bg/2007/06/travelling-salesman-problem.html [↑](#footnote-ref-24)
25. http://galois.com/wp-content/uploads/2014/08/pub\_JL\_StructuringDFSAlgorithms.pdf [↑](#footnote-ref-25)