agis-0.2.0.0: Heuristic Search Library & Framework for Haskell

Heuristic Search Library & Framework for Haskell

Modules

Search.Monadic.Base

Copyright	(c) Diego Vicente Martín 2017
License	GPL-3
Maintainer	diegovicente@protonmail.com
Stability	experimental
Safe Haskell	Safe
Language	Haskell2010

This module contains all the type and data declarations that are used accross the monadic part of library. It is a good idea to read and understand all these types to gain a better intuition about how the library works.

The `Monadic` part of the library uses a monad to keep track of different statistics of the search during the execution, as well as different logs. This causes overhead in the computations, so it should be used for the study of different algorithms instead of heavy computations. To solve problems, the `Search.Pure` modules are equivalent without using this monad, and will perform better.

Contents

Algorithms

Algorithm Components

`data Node a` #

A `Node` is the minimal unit of a search: it holds a state and all its contextual infromation.

Constructors

`Node`

`getState :: a`	Current state in the node
`getCost :: Double`	Current cost of the node
`getHeuristic :: Double`	Heuristic value of the node
`getPath :: [a]`	List of ordered previous states
`getVisited :: HashSet a`	Log(n) access structure of visited states

Instances

- `(Hashable a, Eq a) => Eq (Node a)` #
- `Show a => Show (Node a)` #
- `Hashable a => Hashable (Node a)` #

`newNode :: (Eq a, Hashable a) => a -> Node a` #

`newNode` receives a state and wraps it in a new `Node`.

`data ProblemSpace a` #

A `ProblemSpace` is a data record that contain all the needed information of a problem.

Constructors

`ProblemSpace`

`getActions :: [Operator a]`	List of actions of the problem
`getInitial :: a`	Initial state of the problem
`getGoalF :: a -> Bool`	Function to check if a state is final

`type Algorithm a = ProblemSpace a -> SearchM (Maybe (Node a))` #

The `Algorithm` receives a problem as input and returns the infinite list of solution nodes. An algorithm has to be fed (at least) with a problem, but some algorithms can have more complex signatures like `Cost a -> Algorithm a`. The `Monadic` modules return a tuple of a `Node` and search `Statistics` per solution

The Search Monad

`data SearchM a` #

The `SearchM` represents a Monad that is in charge of tracking all the `Statistics` of a search. Bear in mind that only the methods form the `Monadic` modules are able to cope with this monad. This monad's purpose is to be used during the search, but every `Algorithm` should drop the monad and return a `[(Node a, Statistics)]` list.

Constructors

`SearchM`

`getNode :: a`	Node with solution
`getStats :: Statistics`	Complete search statistics

Instances

```
Monad SearchM
Functor SearchM
Applicative SearchM
```

```
data Statistics
```

The record `Statistics` keep track of different search measures. These values are kept in a different object instead of `SearchM` itself to be able to keep track of the different statistics of the solutions returned by the list: if they were in the monad we could not be able to get expanded nodes but the ones of the complete search.

Constructors

`Statistics`

`nodesExpanded :: Integer`	Number of nodes expanded through the whole search
`nodesEnqueued :: Integer`	Number of nodes that have been enqueued through the whole search
`maxQueueLength :: Integer`	Maximum number of nodes that have been enqueued at the same time

Instances

```
Eq Statistics
Show Statistics
```

Functions

```
type NodeEvaluation a = Node a -> Double
```

To evaluate nodes in different situations, we want a `NodeEvaluation` function that can receive a `Node` and return a numeric value.

```
type Operator a = a -> Maybe a
```

An `Operator` is a function that is able to expand a state a if valid, or return `Nothing` if not.

```
type Cost a = a -> Operator a -> Double
```

A `Cost` function receives a state, an action and returns its cost

```
type Heuristic a = a -> Double
```

A `Heuristic` function receives a state and returns its heuristic value

Data Structures

```
class DataStructure ds where
```

When programming new functional search algorithms, one of the most important pieces of the algorithm is the `DataStructure`: It will be the component holding the nodes of the search, and defining its behavior (when to expand a given node) will shape the algorithm. For a type to be defined as a `DataStructure`, some functions have to be defined for it.

Minimal complete definition

`add, addList, next, isEmpty, sizeDS`

Methods

```
add :: (Eq a, Hashable a) => Node a -> ds a -> ds a
```

`add` defines how a new node is added to the structure.

```
addList :: (Eq a, Hashable a) => [Node a] -> ds a -> ds a
```

`addList` defines how a list of nodes have to be added to the structure

```
next :: (Eq a, Hashable a) => ds a -> (ds a, Node a)
```

`next` defines which node of the structure should be selected next

```
isEmpty :: (Eq a, Hashable a) => ds a -> Bool
```

`isEmpty` checks if the structure has no `Node`s left

```
sizeDS :: (Eq a, Hashable a) => ds a -> Int
```

`sizeDS` returns the number of nodes that are enqueued at the moment

Instances

```
DataStructure BeamStack
```

DataStructure BoundedStack #

DataStructure PriorityQueue #

DataStructure Queue #

DataStructure Stack #

Search.Monadic.Benchmark

Copyright	(c) Diego Vicente Martín 2017
License	GPL-3
Maintainer	diegovicente@protonmail.com
Stability	experimental
Safe Haskell	None
Language	Haskell2010

This modules provides an easy interface to the `criterion` library for performing benchmarks and obtain comprehensive data, as well as several pre-configured benchmarks of small size using different toy problems.

Contents

Benchmark functions

`benchmark :: [Benchmark] -> IO ()` #

`benchmark` receives a list of `Benchmark` to perform using the default configuration

`benchmarkBy :: Config -> [Benchmark] -> IO ()` #

`benchmarkBy` receives a `criterion` `Config` and benchmarks according to it. Read the `criterion` configuration for more details about it.

Criterion types

`data Benchmark :: *` #

Specification of a collection of benchmarks and environments. A benchmark may consist of:

- An environment that creates input data for benchmarks, created with `env`.
- A single `Benchmarkable` item with a name, created with `bench`.
- A (possibly nested) group of `Benchmark`s, created with `bgroup`.

Instances

- `Show Benchmark`

`data Config :: *` #

Top-level benchmarking configuration.

Instances

- `Eq Config`
- `Data Config`
- `Read Config`
- `Show Config`
- `Generic Config`
- `type Rep Config`

Benchmark suites

`withMaze :: [(String, Algorithm Coord)] -> [Benchmark]` #

`withMaze` returns a list of `Benchmark` of the `Algorithm`s passed (identiefied with a corresponding `String`) in a maze.

`withEightPuzzle :: [(String, Algorithm Puzzle)] -> [Benchmark]` #

`withMaze` returns a list of `Benchmark` of the `Algorithm`s passed (identiefied with a corresponding `String`) in 8-Puzzle.

`withNQueens :: [(String, Algorithm Queens)] -> [Benchmark]` #

`withMaze` returns a list of `Benchmark` of the `Algorithm`s passed (identiefied with a corresponding `String`) in N-Queens.

Search.Monadic.DataStructure.BeamStack

Copyright	(c) Diego Vicente Martín 2017
License	GPL-3
Maintainer	diegovicente@protonmail.com
Stability	experimental
Safe Haskell	Safe
Language	Haskell2010

This module contains a `DataStructure` instance, `BeamStack` that behaves like a LIFO list of preference: only a given number of nodes are pushed to the beginning of the list, that is, the most promising nodes (by a provided function) are added to the front. This allows us to perform a kind of DFS, but enqueueing less nodes than in the regular structure. `BeamStack` is called that because it is the cornerstone of the `beamSearch` algorithm. It is based in a regular list, that offers *O(1)* complexity adding elements to the front.

Contents

Instance of `DataStructure`

`class DataStructure ds where` #

When programming new functional search algorithms, one of the most important pieces of the algorithm is the `DataStructure`: It will be the component holding the nodes of the search, and defining its behavior (when to expand a given node) will shape the algorithm. For a type to be defined as a `DataStructure`, some functions have to be defined for it.

Minimal complete definition

`add, addList, next, isEmpty, sizeDS`

Methods

`add :: (Eq a, Hashable a) => Node a -> ds a -> ds a` #

`add` defines how a new node is added to the structure.

`addList :: (Eq a, Hashable a) => [Node a] -> ds a -> ds a` #

`addList` defines how a list of nodes have to be added to the structure

`next :: (Eq a, Hashable a) => ds a -> (ds a, Node a)` #

`next` defines which node of the structure should be selected next

`isEmpty :: (Eq a, Hashable a) => ds a -> Bool` #

`isEmpty` checks if the structure has no `Node`s left

`sizeDS :: (Eq a, Hashable a) => ds a -> Int` #

`sizeDS` returns the number of nodes that are enqueued at the moment

Instances

- `DataStructure BeamStack` #
- `DataStructure BoundedStack` #
- `DataStructure PriorityQueue` #
- `DataStructure Queue` #
- `DataStructure Stack` #

`data BeamStack a` #

A `BeamStack` represents a Last-In First-Out structure, where only the `limit` most promising nodes defined by a `sortingFunction` are actually added to the structure. That allows to expand the nodes in a DFS fashion, but consuming less memory.

Constructors

`BeamStack`

- `beamSToList :: [Node a]`
- `beamWidth :: Int`
- `evalFunction :: NodeEvaluation a`

Instances

- `DataStructure BeamStack` #
- `(Hashable a, Eq a) => Eq (BeamStack a)` #
- `Show a => Show (BeamStack a)` #

Constructor shortcuts

`newBeamStack :: (Hashable a, Eq a) => [Node a] -> Int -> NodeEvaluation a -> BeamStack a` #

Search.Monadic.DataStructure.BoundedStack

Copyright	(c) Diego Vicente Martín 2017
License	GPL-3
Maintainer	diegovicente@protonmail.com
Stability	experimental
Safe Haskell	Safe
Language	Haskell2010

This module contains a `DataStructure` instance, `BoundedStack`, that behaves as a Last-In First-Out structure. `BoundedStack` is used in the default library as the `DataStructure` for `idfs` and `idAStar`. `BoundedStack` is based on a regular list since it is *O(1)* new nodes to the beginning of a list.

Contents

Instance of `DataStructure`

```
class DataStructure ds where
```

When programming new functional search algorithms, one of the most important pieces of the algorithm is the `DataStructure`: It will be the component holding the nodes of the search, and defining its behavior (when to expand a given node) will shape the algorithm. For a type to be defined as a `DataStructure`, some functions have to be defined for it.

Minimal complete definition

`add, addList, next, isEmpty, sizeDS`

Methods

```
add :: (Eq a, Hashable a) => Node a -> ds a -> ds a
```

`add` defines how a new node is added to the structure.

```
addList :: (Eq a, Hashable a) => [Node a] -> ds a -> ds a
```

`addList` defines how a list of nodes have to be added to the structure

```
next :: (Eq a, Hashable a) => ds a -> (ds a, Node a)
```

`next` defines which node of the structure should be selected next

```
isEmpty :: (Eq a, Hashable a) => ds a -> Bool
```

`isEmpty` checks if the structure has no `Node`s left

```
sizeDS :: (Eq a, Hashable a) => ds a -> Int
```

`sizeDS` returns the number of nodes that are enqueued at the moment

Instances

- `DataStructure BeamStack`
- `DataStructure BoundedStack`
- `DataStructure PriorityQueue`
- `DataStructure Queue`
- `DataStructure Stack`

```
data BoundedStack a
```

A `BoundedStack` represents a stack whose nodes are depth-bounded: following the same LIFO behavior as a `Stack`, but only the nodes that fulfill the condition *f(n) <= l* are added to the `BoundedStack`, where *f(n)* is the `NodeEvaluation` function used to restrict the structure and *l* is the limit.

Constructors

```
BoundedStack
    bStackToList :: [Node a]
    getEval :: NodeEvaluation a
    getLimit :: Double
```

Instances

- `DataStructure BoundedStack`
- `(Hashable a, Eq a) => Eq (BoundedStack a)`
- `Show a => Show (BoundedStack a)`

Constructor shortcuts

```
newBoundedStack :: Eq a => [Node a] -> NodeEvaluation a -> Double -> BoundedStack a
```

Create a new `BoundedStack` from a list of `Node`s and a limit

Search.Monadic.DataStructure.PriorityQueue

Copyright	(c) Diego Vicente Martín 2017
License	GPL-3
Maintainer	diegovicente@protonmail.com
Stability	experimental
Safe Haskell	Safe
Language	Haskell2010

This module contains a `DataStructure` instance, `PriorityQueue`, that behaves as a sorted structure. `PriorityQueue` is used in the default library as the `DataStructure` for `ucs`, `greedy`, or `aStar`. `PriorityQueue` is based on `IntPSQ`, which let us create a sorted Priority Queue on top of it.

Contents

Instance of `DataStructure`

`class DataStructure ds where` #

When programming new functional search algorithms, one of the most important pieces of the algorithm is the `DataStructure`: It will be the component holding the nodes of the search, and defining its behavior (when to expand a given node) will shape the algorithm. For a type to be defined as a `DataStructure`, some functions have to be defined for it.

Minimal complete definition

`add, addList, next, isEmpty, sizeDS`

Methods

`add :: (Eq a, Hashable a) => Node a -> ds a -> ds a` #

`add` defines how a new node is added to the structure.

`addList :: (Eq a, Hashable a) => [Node a] -> ds a -> ds a` #

`addList` defines how a list of nodes have to be added to the structure

`next :: (Eq a, Hashable a) => ds a -> (ds a, Node a)` #

`next` defines which node of the structure should be selected next

`isEmpty :: (Eq a, Hashable a) => ds a -> Bool` #

`isEmpty` checks if the structure has no `Node`s left

`sizeDS :: (Eq a, Hashable a) => ds a -> Int` #

`sizeDS` returns the number of nodes that are enqueued at the moment

Instances

- `DataStructure BeamStack` #
- `DataStructure BoundedStack` #
- `DataStructure PriorityQueue` #
- `DataStructure Queue` #
- `DataStructure Stack` #

`data PriorityQueue a` #

A `PriorityQueue` represents a sorted queue of nodes, that allows to expand the nodes in a given order nevermind the order in which they were expanded. All new `Node`s are added to its corresponding position (according to the `NodeComparison` function) and the next node returned is the one with the **minimum** value according to the sorting function. It is built on top of a Data.Heap structure, to allow ordered insertion.

Constructors

`PriorityQueue`

- **`priorityQueueToHeap :: IntPSQ Double (Node a)`**
- **`valueFunction :: NodeEvaluation a`**

Instances

- `DataStructure PriorityQueue` #
- `(Hashable a, Eq a) => Eq (PriorityQueue a)` #
- `Show a => Show (PriorityQueue a)` #

Constructor shortcuts

`newPriorityQueue :: (Hashable a, Eq a) => [Node a] -> NodeEvaluation a -> PriorityQueue a` #

Create a new `PriorityQueue` from a list of `Node`s and sorted by a `NodeComparison` function

Search.Monadic.DataStructure.Queue

Copyright	(c) Diego Vicente Martín 2017
License	GPL-3
Maintainer	diegovicente@protonmail.com
Stability	experimental
Safe Haskell	Safe
Language	Haskell2010

This module contains a `DataStructure` instance, `Queue`, that behaves as a First-In First-Out structure. `Queue` is used in the default library as the `DataStructure` for `bfs`. `Queue` is built upon the `Data.Sequence` package to allow *O(1)* complexity wen adding elements at the end of the Sequence.

Contents

Instance of `DataStructure`

```
class DataStructure ds where
```

When programming new functional search algorithms, one of the most important pieces of the algorithm is the `DataStructure`: It will be the component holding the nodes of the search, and defining its behavior (when to expand a given node) will shape the algorithm. For a type to be defined as a `DataStructure`, some functions have to be defined for it.

Minimal complete definition

`add, addList, next, isEmpty, sizeDS`

Methods

```
add :: (Eq a, Hashable a) => Node a -> ds a -> ds a
```

`add` defines how a new node is added to the structure.

```
addList :: (Eq a, Hashable a) => [Node a] -> ds a -> ds a
```

`addList` defines how a list of nodes have to be added to the structure

```
next :: (Eq a, Hashable a) => ds a -> (ds a, Node a)
```

`next` defines which node of the structure should be selected next

```
isEmpty :: (Eq a, Hashable a) => ds a -> Bool
```

`isEmpty` checks if the structure has no `Node`s left

```
sizeDS :: (Eq a, Hashable a) => ds a -> Int
```

`sizeDS` returns the number of nodes that are enqueued at the moment

Instances

```
DataStructure BeamStack
DataStructure BoundedStack
DataStructure PriorityQueue
DataStructure Queue
DataStructure Stack
```

```
newtype Queue a
```

A `Queue` represents a First-In First-Out structure: Each new `Node` is added to the back of the `Queue`, and when extracting the next `Node` the first one that was added is returned

Constructors

```
Queue
    queueToSeq :: Seq (Node a)
```

Instances

```
DataStructure Queue
(Eq a, Hashable a) => Eq (Queue a)
Show a => Show (Queue a)
```

Constructor shortcuts

```
newQueue :: (Hashable a, Eq a) => [Node a] -> Queue a
```

Create a new `Queue` from a list of `Node`s

```
startQueue :: (Hashable a, Eq a) => a -> Queue a
```

Create a new `Queue` from a state (to be treated as initial node)

Search.Monadic.DataStructure.Stack

Copyright	(c) Diego Vicente Martín 2017
License	GPL-3
Maintainer	diegovicente@protonmail.com
Stability	experimental
Safe Haskell	Safe
Language	Haskell2010

This module contains a `DataStructure` instance, `Stack`, that behaves as a Last-In First-Out structure. `Stack` is used in the default library as the `DataStructure` for `dfs`. `Stack` is based on a regular list since it is *O(1)* new nodes to the beginning of a list.

Contents

Instance of `DataStructure`

`class DataStructure ds where` #

When programming new functional search algorithms, one of the most important pieces of the algorithm is the `DataStructure`: It will be the component holding the nodes of the search, and defining its behavior (when to expand a given node) will shape the algorithm. For a type to be defined as a `DataStructure`, some functions have to be defined for it.

Minimal complete definition

`add, addList, next, isEmpty, sizeDS`

Methods

`add :: (Eq a, Hashable a) => Node a -> ds a -> ds a` #

`add` defines how a new node is added to the structure.

`addList :: (Eq a, Hashable a) => [Node a] -> ds a -> ds a` #

`addList` defines how a list of nodes have to be added to the structure

`next :: (Eq a, Hashable a) => ds a -> (ds a, Node a)` #

`next` defines which node of the structure should be selected next

`isEmpty :: (Eq a, Hashable a) => ds a -> Bool` #

`isEmpty` checks if the structure has no `Node`s left

`sizeDS :: (Eq a, Hashable a) => ds a -> Int` #

`sizeDS` returns the number of nodes that are enqueued at the moment

Instances

- `DataStructure BeamStack` #
- `DataStructure BoundedStack` #
- `DataStructure PriorityQueue` #
- `DataStructure Queue` #
- `DataStructure Stack` #

`newtype Stack a` #

A `Stack` represents a Last-In First-Out structure: Each new `Node` is added to the beginning of the `Stack`, and when popping the next `Node` to be expanded the last one in is returned

Constructors

`Stack`

`stackToList :: [Node a]`

Instances

- `DataStructure Stack` #
- `(Eq a, Hashable a) => Eq (Stack a)` #
- `Show a => Show (Stack a)` #

Constructor shortcuts

`newStack :: (Hashable a, Eq a) => [Node a] -> Stack a` #

Create a new `Stack` from a list of `Node`s

`startStack :: (Hashable a, Eq a) => a -> Stack a` #

Create a new `Stack` from a state (to be treated as initial node)

Search.Monadic.General

Copyright	(c) Diego Vicente Martín 2017
License	GPL-3
Maintainer	diegovicente@protonmail.com
Stability	experimental
Safe Haskell	Safe
Language	Haskell2010

Contents

This module contains the general method to perform searches: `generalSearch`. This method can be considered the generalizations of search algorithms, and different search algorithms can be implemented by passing different `DataStructures` to it. Also, due to the high memory consumption it generates by actually storing the nodes in a structure, the methods `depthSearch` and `limitedDepthSearch` are also provided, which perform the search by ordering recursive stack calls (and thus using linear memory). These methods offer a more limited control over the execution. Furthermore, an implementation of the Depth-First Branch & Bound is provided using folds in `depthBNB`.

`generalSearch` is inspired by the `GENERAL-SEARCH` algorithm proposed in Russel & Norvig's *Artificial Intelligence: A Modern Approach*. Contrary to its pure counterpart, the lazy evaluation cannot be taken advantage of due to the the `SearchM` collection of statistics, so the first solution found is returned.

General Search

generalSearch #

`:: (DataStructure ds, Eq a, Hashable a)`	
`=> ProblemSpace a`	`ProblemSpace` to be solved
`-> Cost a`	`Cost` function to use
`-> Heuristic a`	`Heuristic` function to use
`-> ds a`	`DataStructure` that manages the node expansion
`-> SearchM (Maybe (Node a))`	Returns the solution obtained

`generalSearch` performs a brute force search traversing a tree. Receives a problem space used to model the problem, a list of nodes to be expanded and a `DataStructure` function that dictates how the expanded nodes are inserted in the list of nodes.

depthSearch #

`:: (Eq a, Hashable a)`	
`=> ProblemSpace a`	`ProblemSpace` to be solved
`-> Cost a`	`Cost` function to use
`-> Heuristic a`	`Heuristic` function to use
`-> NodeEvaluation a`	`NodeEvaluation` to sort the expanded nodes
`-> Node a`	Current `Node` to be expanded
`-> SearchM (Maybe (Node a))`	Solution wrapped in the `SearchM` monad

`depthSearch` performs a search without an explicit `DataStructure`, instead it performs a recursion tree that enables backtracking to the nodes expanded. Nodes expanded in each call can be sorted with a given `NodeEvaluation` function, or this feature can be ignored by passing `noSorting` dummy function to the function.

limitedDepthSearch #

`:: (Eq a, Hashable a)`	
`=> ProblemSpace a`	`ProblemSpace` to be solved
`-> Cost a`	`Cost` function to use
`-> Heuristic a`	`Heuristic` function to use
`-> NodeEvaluation a`	`NodeEvaluation` to sort the expanded nodes
`-> Double`	Limit to be imposed to the `NodeEvaluation`
`-> Node a`	Current `Node` to be expanded
`-> SearchM (Maybe (Node a))`	Solution wrapped in the `SearchM` monad

`limitedDepthSearch` performs a search in similar way of `depthSearch`, but imposing a given limit on the value returned by the `NodeEvaluation` function. If the value of `f node` is larger than the provided limit, the node expanded is not recursively called upon.

iterateSearch #

`:: (Eq a, Hashable a)`	
`=> ProblemSpace a`	`ProblemSpace` to be solved
`-> Cost a`	`Cost` function to use
`-> Heuristic a`	`Heuristic` function to use
`-> NodeEvaluation a`	`NodeEvaluation` to sort the expanded nodes
`-> [Double]`	List of limits to be imposed to `limitedDepthSearch`
`-> SearchM (Maybe (Node a))`	Solution wrapped in the `SearchM` monad

`iterateSearch` is a wrapper function that runs `limitedDepthSearch` with a list of limits: if a limit returns no solution, the search using the next limit is executed. The result provided includes the sum of all the different searches performed (with different limits).

depthBNB #

`:: (Eq a, Hashable a)`	
`=> ProblemSpace a`	`ProblemSpace` to be solved
`-> Cost a`	`Cost` function to use
`-> Heuristic a`	`Heuristic` function to use
`-> NodeEvaluation a`	`NodeEvaluation` to sort and bound the expanded nodes
`-> Node a`	Current `Node` to be expanded
`-> (Double, Maybe (Node a))`	The current bound and intermediate solutions found (in ascending cost order)
`-> SearchM (Double, Maybe (Node a))`	The final bound and all solutions found (in ascending cost order)

`depthBNB` is a recursive function to perform Branch & Bound using folds. The accumulator of the fold stores both the lowest bound currently found and the current best solution of the search.

expand #

`:: (Eq a, Hashable a)`	
`=> Node a`	`Node` to be expanded
`-> Cost a`	`Cost` function to update the new nodes
`-> Heuristic a`	`Heuristic` function to update the new nodes
`-> [Operator a]`	List of actions to apply on the node
`-> [Node a]`	List of all the valid `Node`s expanded

`expand` function applies all the actions of the problem and checks if the node has been previously visited in the `Node`.

Dummy functions

Dummy functions are used when no `Heuristic` and/or `Cost` function is required (i.e uninformed algorithms like `bfs`). In that case, the function passed to `generalSearch` just updates the values in an uniform way.

noCost `:: Cost a` #

Dummy `Cost` function.

noHeuristic `:: Heuristic a` #

Dummy `Heuristic` function.

noSorting `:: NodeEvaluation a` #

Dummy sorting for `depthSearch`.

Logging functions

Logging functions are used to provide a convenient way to update the statistics of the search. Each function is just a shortcut to creating a new `SearchM` to be bind together to the already collected statistics. These functions try to help in case the user wants to build their own monadic algorithm, to collect all the necessary statistics.

logExpanded `:: SearchM ()` #

Increase the count of expanded nodes by one.

logEnqueued `:: Int -> SearchM ()` #

Increase the count of enqueued nodes by a given `Int`.

logLength `:: (Eq a, Hashable a, DataStructure ds) => ds a -> SearchM ()` #

Log a new `DataStructure` length to be compared to current recorded maximum.

Search.Monadic.Informed

Copyright	(c) Diego Vicente Martín 2017
License	GPL-3
Maintainer	diegovicente@protonmail.com
Stability	experimental
Safe Haskell	Safe
Language	Haskell2010

This module contains different informed search algorithms ready to be used.

Documentation

```
hillClimbing :: (Eq a, Hashable a) => Cost a -> Heuristic a -> Algorithm a
```

`hillClimbing` runs a Greedy Heuristic Search

```
aStar :: (Eq a, Hashable a) => Cost a -> Heuristic a -> Algorithm a
```

`aStar` runs an A* Search

```
idAStar :: (Eq a, Hashable a) => (Double, Double) -> Cost a -> Heuristic a -> Algorithm a
```

`idAStar` runs an Iterative Deepening A* Search

```
beam :: (Eq a, Hashable a) => Int -> Cost a -> Heuristic a -> Algorithm a
```

`beam` runs a Beam Search of a given beam width

```
dfBNB :: (Eq a, Hashable a) => Cost a -> Heuristic a -> Algorithm a
```

`dfBNB` performs a Depth-First Branch & Bound Search.

Search.Monadic.ToyProblem.EightPuzzle

Copyright	(c) Diego Vicente Martín 2017
License	GPL-3
Maintainer	diegovicente@protonmail.com
Stability	experimental
Safe Haskell	None
Language	Haskell2010

This module contains a way to approach the 8-Puzzle problem using the library: by understanding the blank tile as an agent that is able to move accross the board, we can try to search for actions that we have to perform (move the blank tile to a certain direction) to solve the puzzle. A way to find the shortest solution is to run a Breadth-First Search on it.

Contents

The 8-Puzzle problem

Types

```
type Coord = (Int, Int)
```

`Coord` helps us express a coordinate in 2D

```
type Puzzle = Vector Int
```

`Puzzle` is represented as a vector of Ints, which allows us to update it easily

```
type Index = Int
```

`Index` helps us express an index in 1D

```
data Direction
```

`Direction` is used to define the direction of the movements

Problem parts

```
easy :: Puzzle
```

`initial` is the initial board for tests

```
hard :: Puzzle
```

`hard` is a puzzle can be solved in 17 movements

```
hamming :: Puzzle -> Puzzle -> Int
```

`hamming` returns the Hamming value of a puzzle: the number of tiles that are not correctly positioned.

```
swapTiles :: Puzzle -> Coord -> Coord -> Puzzle
```

`swapTiles` swaps two tiles (identified by its coordinates)

```
moveBlank :: Direction -> Puzzle -> Maybe Puzzle
```

`moveBlank` moves the blank tile if the direction is possible, or returns `Nothing` if not

```
isFinal :: Puzzle -> Bool
```

`isFinal` checks if the puzzle is in a goal state. An 8-Puzzle is solved if all its tiles are in order (despite the blank)

```
puzzleHeuristic :: Heuristic Puzzle
```

`puzzleHeuristic` generates a `Heuristic` function using `hamming`

Interfaces to the Library

```
buildProblemSpace :: Puzzle -> ProblemSpace Puzzle
```

`buildProblemSpace` receives a initial state of the `Puzzle` and returns a complete `ProblemSpace` to be solved.

```
nStepsUninformed :: Puzzle -> Algorithm Puzzle -> (Maybe Int, Statistics)
```

`nSteps` solves the `puzzle` with a given algorithm and returns the number of steps that it took.

```
nStepsInformed :: Puzzle -> (Cost Puzzle -> Heuristic Puzzle -> Algorithm Puzzle) -> (Maybe Int, Statistics)
```

Search.Monadic.ToyProblem.MapParser

Copyright	(c) Diego Vicente Martín 2017
License	GPL-3
Maintainer	diegovicente@protonmail.com
Stability	experimental
Safe Haskell	None
Language	Haskell2010

Contents

This module contains an implementation of Search that is able to be parse a Moving AI map (as provided in http://www.movingai.com/benchmarks/wc3maps512/) and perform the shortest route available between two points given.

Moving AI Routing

Types

`type Coord = (Int, Int)` #

`Coord` type represents a pair of coordinates in the maze

`data Direction` #

`Direction` represents the available directions

Instances

- `Eq Direction` #
- `Show Direction` #

`type Map = Vector (Vector Char)` #

A `Map` is a multi-dimensional `Vector` of characters.

Problem Parts

`parse :: [String] -> Map` #

`parse` is a function dedicated to parse Moving AI map strings into a `Map`.

`getCell :: Map -> Coord -> Char` #

`getCell` returns the value of a `Coord` in a given `Map`

`cellCost :: Char -> Double` #

`cellCost` returns the cost of a cell's value.

`move :: Map -> Direction -> Coord -> Maybe Coord` #

`move` returns a new `Coord` if it was possible to move from an initial position to a destination using a `Direction` in a given `Map`

`manhattan :: Coord -> Coord -> Double` #

`manhattan` computes the Manhattan distance between two points

Default maps

`small :: [String]` #

An easy maze, used for several testing purposes, defined in a 5x5 grid.

`medium :: [String]` #

A medium sized maze, defined in a 10x10 grid.

`big :: [String]` #

A medium sized maze, defined in a 15x15 grid.

Interfaces to the Library

`costFunction :: Map -> Cost Coord` #

`costFunction` is able to return the cost of a movement in a `Map`.

```
buildProblemSpace :: Map -> Coord -> Coord -> ProblemSpace Coord
```

`buildProblemspace` generates a new `ProblemSpace` by receiving a `Map` and an initial and final `Coord`.

```
solveUninformed :: Map -> Coord -> Coord -> Algorithm Coord -> (Maybe (Node Coord), Statistics)
```

`solveUninformed` finds a route between two given `Coord` in a `Map` using a given uninformed `Algorithm`.

```
solveInformed :: Map -> Coord -> Coord -> (Cost Coord -> Heuristic Coord -> Algorithm Coord) -> (Maybe (Node Coord), Statistics)
```

`solveUninformed` finds a route between two given `Coord` in a `Map` using a given uninformed `Algorithm`.

Input/Output

```
readMap :: FilePath -> IO Map
```

`readWMap` receives a `FileName` and reads it using `parse` to return a `Map`.

```
route :: FilePath -> Coord -> Coord -> IO ()
```

`route` finds a the shortest path between two points with a A* Search in the map stored in `FilePath`. It prints in the screen the cost of the path found.

Search.Monadic.ToyProblem.NQueens

Copyright	(c) Diego Vicente Martín 2017
License	GPL-3
Maintainer	diegovicente@protonmail.com
Stability	experimental
Safe Haskell	None
Language	Haskell2010

Contents

This module contains a way to approach the N-Queens problem using the library: to do so we understand the actions of the problem as adding a new queen to next column possible and a given row. That way the problem is modelled as a tree, that can be search using for example a Depth-First Search.

An example of how the solution is displayed is:

```
>>> solveQueens dfs
♕ . . . . . . .
. . . . ♕ . . .
. . . . . . . ♕
. . . . . ♕ . .
. . ♕ . . . . .
. . . . . . ♕ .
. ♕ . . . . . .
. . . ♕ . . . .
 - Number of expanded nodes: 114
 - Number of enqueued nodes: 0
 - Maximum length of the queue: 0
```

The N-Queens problem

Types

`type Coord = (Int, Int)` #

`Coord` type represents a position in the board

`type Queens = [Coord]` #

`Queens` is a list of `Coord` that represent the position of each queen

Problem parts

`isValid :: Queens -> Coord -> Bool` #

`isValid` checks if the new queen proposed is valid or not by checking conflicts with the existing queens in the board

`isFinal :: Queens -> Bool` #

`isFinal` checks if a given board is a goal state. The problem is over if all queens have been placed with no conflicts

`conflicts :: Queens -> Int` #

`conflicts` return the number of queens that can attack other queens

`newQueen :: Int -> Queens -> Maybe Queens` #

`newQueen` adds a queen in the first column available in the right, or returns `Nothing` if there is a conflict with an existing queen

`showQueens :: Queens -> IO ()` #

`showQueens` prints the board in a readable way: where `.` represents an empty cell and ♕ represents a queen (there's an example in the module description)

Interfaces to the Library

`queens :: ProblemSpace Queens` #

The N-Queens problem modelled as a `ProblemSpace`

`solveQueens :: Algorithm Queens -> IO ()` #

`solveQueens` solves `queens` and displays its result as found (using `showQueens`)

`getCoords :: Algorithm Queens -> Maybe [Coord]` #

Extract the coordinates for testing purposes

Search.Monadic.Uninformed

Copyright	(c) Diego Vicente Martín 2017
License	GPL-3
Maintainer	diegovicente@protonmail.com
Stability	experimental
Safe Haskell	Safe
Language	Haskell2010

This module contains different uninformed search algorithms ready to be used.

Contents

Uninformed Search Algorithms

`bfs :: (Eq a, Hashable a) => Algorithm a` #

`bfs` runs a Breadth-First Search

`dfs :: (Eq a, Hashable a) => Algorithm a` #

`dfs` runs a Depth-First Search

`idfs :: (Eq a, Hashable a) => (Int, Int) -> Algorithm a` #

`idfs` runs an Iterative-Deepening Depth-First Search. The first argument, a pair of `Int`s (`step`, `inf`), represent the main parameters of the search: each new iteration the depth test is incremented by adding `step` as long as the new depth is lower than `inf`.

`ucs :: (Eq a, Hashable a) => Cost a -> Algorithm a` #

`ucs` runs an Uniform-Cost Search with a given cost function

Dummy functions

`noCost :: Cost a` #

Dummy `Cost` function.

`noHeuristic :: Heuristic a` #

Dummy `Heuristic` function.

`noSorting :: NodeEvaluation a` #

Dummy sorting for `depthSearch`.

Search.Pure.Base

Copyright	(c) Diego Vicente Martín 2017
License	GPL-3
Maintainer	diegovicente@protonmail.com
Stability	experimental
Safe Haskell	Safe
Language	Haskell2010

This module contains all the type and data declarations that are used accross the pure part of the library. It is a good idea to read and understand all these types to gain a better intuition about how the library works. The `Pure` methods and data types are recommended to be used when trying to solve a problem, disregarding the execution details and statistics of the algorithms.

To keep track of execution logs and obtain stastics, use the `Monadic` version, defined as `Search.Monadic` modules.

Contents

Algorithms

Algorithm Components

`data Node a` #

A `Node` is the minimal unit of a search: it holds a state and all its contextual infromation.

Constructors

`Node`

`getState :: a`	Current state in the node
`getCost :: Double`	Current cost of the node
`getHeuristic :: Double`	Heuristic value of the node
`getPath :: [a]`	List of ordered previous states
`getVisited :: HashSet a`	Log(n) access structure of visited states

Instances

- `(Hashable a, Eq a) => Eq (Node a)` #
- `Show a => Show (Node a)` #
- `Hashable a => Hashable (Node a)` #

`newNode :: (Eq a, Hashable a) => a -> Node a` #

`newNode` receives a state and wraps it in a new `Node`.

`data ProblemSpace a` #

A `ProblemSpace` is a data record that contain all the needed information of a problem.

Constructors

`ProblemSpace`

`getActions :: [Operator a]`	List of actions of the problem
`getInitial :: a`	Initial state of the problem
`getGoalF :: a -> Bool`	Function to check if a state is final

`type Algorithm a = ProblemSpace a -> [Node a]` #

The `Algorithm` receives a problem as input and returns the infinite list of solution nodes. An algorithm has to be fed (at least) with a problem, but some algorithms can have more complex signatures like `Cost a -> Algorithm a`.

Functions

`type NodeEvaluation a = Node a -> Double` #

To evaluate nodes in different situations, we want a `NodeEvaluation` function that can receive a `Node` and return a numeric value.

`type Operator a = a -> Maybe a` #

An `Operator` is a function that is able to expand a state a if valid, or return `Nothing` if not.

`type Cost a = a -> Operator a -> Double` #

A `Cost` function receives a state, an action and returns its cost.

`type Heuristic a = a -> Double` #

A `Heuristic` function receives a state and returns its heuristic value.

Data Structures

```
class DataStructure ds where
```

When programming new functional search algorithms, one of the most important pieces of the algorithm is the `DataStructure`: It will be the component holding the nodes of the search, and defining its behavior (when to expand a given node) will shape the algorithm. For a type to be defined as a `DataStructure`, some functions have to be defined for it.

Minimal complete definition

`add`, `addList`, `next`, `isEmpty`

Methods

```
add :: (Eq a, Hashable a) => Node a -> ds a -> ds a
```

`add` defines how a new node is added to the structure.

```
addList :: (Eq a, Hashable a) => [Node a] -> ds a -> ds a
```

`addList` defines how a list of nodes have to be added to the structure.

```
next :: (Eq a, Hashable a) => ds a -> (ds a, Node a)
```

`next` defines which node of the structure should be selected next.

```
isEmpty :: (Eq a, Hashable a) => ds a -> Bool
```

`isEmpty` checks if the structure has no `Node`s left.

Instances

- `DataStructure BeamStack`
- `DataStructure BoundedStack`
- `DataStructure PriorityQueue`
- `DataStructure Queue`
- `DataStructure Stack`

Search.Pure.Benchmark

Copyright	(c) Diego Vicente Martín 2017
License	GPL-3
Maintainer	diegovicente@protonmail.com
Stability	experimental
Safe Haskell	None
Language	Haskell2010

This modules provides an easy interface to the `criterion` library for performing benchmarks and obtain comprehensive data, as well as several pre-configured benchmarks of small size using different toy problems.

Contents

Benchmark functions

```
benchmark :: [Benchmark] -> IO ()
```
#

`benchmark` receives a list of `Benchmark` to perform using the default configuration

```
benchmarkBy :: Config -> [Benchmark] -> IO ()
```
#

`benchmarkBy` receives a `criterion` `Config` and benchmarks according to it. Read the `criterion` configuration for more details about it.

Criterion types

```
data Benchmark :: *
```
#

Specification of a collection of benchmarks and environments. A benchmark may consist of:

- An environment that creates input data for benchmarks, created with `env`.
- A single `Benchmarkable` item with a name, created with `bench`.
- A (possibly nested) group of `Benchmark`s, created with `bgroup`.

Instances

- `Show Benchmark`

```
data Config :: *
```
#

Top-level benchmarking configuration.

Instances

- `Eq Config`
- `Data Config`
- `Read Config`
- `Show Config`
- `Generic Config`
- `type Rep Config`

Benchmark suites

```
withMaze :: [(String, Algorithm Coord)] -> [Benchmark]
```
#

`withMaze` returns a list of `Benchmark` of the `Algorithm`s passed (identiefied with a corresponding `String`) in a maze.

```
withEightPuzzle :: [(String, Algorithm Puzzle)] -> [Benchmark]
```
#

`withEightPuzzle` returns a list of `Benchmark` of the `Algorithm`s passed (identiefied with a corresponding `String`) in 8-Puzzle.

```
withNQueens :: [(String, Algorithm Queens)] -> [Benchmark]
```
#

`withNQueens` returns a list of `Benchmark` of the `Algorithm`s passed (identiefied with a corresponding `String`) in N-Queens to find the first solution.

Search.Pure.DataStructure.BeamStack

Copyright	(c) Diego Vicente Martín 2017
License	GPL-3
Maintainer	diegovicente@protonmail.com
Stability	experimental
Safe Haskell	Safe
Language	Haskell2010

This module contains a `DataStructure` instance, `BeamStack` that behaves like a LIFO list of preference: only a given number of nodes are pushed to the beginning of the list, that is, the most promising nodes (by a provided function) are added to the front. This allows us to perform a kind of DFS, but enqueueing less nodes than in the regular structure. `BeamStack` is called that because it is the cornerstone of the `beamSearch` algorithm. It is based in a regular list, that offers *O(1)* complexity adding elements to the front.

Contents

Instance of DataStructure

```
class DataStructure ds where
```

When programming new functional search algorithms, one of the most important pieces of the algorithm is the `DataStructure`: It will be the component holding the nodes of the search, and defining its behavior (when to expand a given node) will shape the algorithm. For a type to be defined as a `DataStructure`, some functions have to be defined for it.

Minimal complete definition

```
add, addList, next, isEmpty
```

Methods

```
add :: (Eq a, Hashable a) => Node a -> ds a -> ds a
```

`add` defines how a new node is added to the structure.

```
addList :: (Eq a, Hashable a) => [Node a] -> ds a -> ds a
```

`addList` defines how a list of nodes have to be added to the structure.

```
next :: (Eq a, Hashable a) => ds a -> (ds a, Node a)
```

`next` defines which node of the structure should be selected next.

```
isEmpty :: (Eq a, Hashable a) => ds a -> Bool
```

`isEmpty` checks if the structure has no `Node`s left.

Instances

```
DataStructure BeamStack
DataStructure BoundedStack
DataStructure PriorityQueue
DataStructure Queue
DataStructure Stack
```

```
data BeamStack a
```

A `BeamStack` represents a Last-In First-Out structure, where only the `limit` most promising nodes defined by a `sortingFunction` are actually added to the structure. That allows to expand the nodes in a DFS fashion, but consuming less memory.

Constructors

```
BeamStack
    beamSToList :: [Node a]
    beamWidth :: Int
    evalFunction :: NodeEvaluation a
```

Instances

```
DataStructure BeamStack
(Hashable a, Eq a) => Eq (BeamStack a)
Show a => Show (BeamStack a)
```

Constructor shortcuts

```
newBeamStack :: (Hashable a, Eq a) => [Node a] -> Int -> NodeEvaluation a -> BeamStack a
```

Create a new `BeamStack` from a list of `Node`s and sorted by a `NodeComparison` function, with a defined limit

```
startBeamStack :: (Hashable a, Eq a) => a -> Int -> NodeEvaluation a -> BeamStack a
```

Search.Pure.DataStructure.BoundedStack

Copyright	(c) Diego Vicente Martín 2017
License	GPL-3
Maintainer	diegovicente@protonmail.com
Stability	experimental
Safe Haskell	Safe
Language	Haskell2010

This module contains a `DataStructure` instance, `BoundedStack`, that behaves as a Last-In First-Out structure. `BoundedStack` is used in the default library as the `DataStructure` for `idfs` and `idAStar`. `BoundedStack` is based on a regular list since it is *O(1)* new nodes to the beginning of a list.

Contents

Instance of `DataStructure`

`class DataStructure ds where` #

When programming new functional search algorithms, one of the most important pieces of the algorithm is the `DataStructure`: It will be the component holding the nodes of the search, and defining its behavior (when to expand a given node) will shape the algorithm. For a type to be defined as a `DataStructure`, some functions have to be defined for it.

Minimal complete definition

`add, addList, next, isEmpty`

Methods

`add :: (Eq a, Hashable a) => Node a -> ds a -> ds a` #

`add` defines how a new node is added to the structure.

`addList :: (Eq a, Hashable a) => [Node a] -> ds a -> ds a` #

`addList` defines how a list of nodes have to be added to the structure.

`next :: (Eq a, Hashable a) => ds a -> (ds a, Node a)` #

`next` defines which node of the structure should be selected next.

`isEmpty :: (Eq a, Hashable a) => ds a -> Bool` #

`isEmpty` checks if the structure has no `Node`s left.

Instances

- `DataStructure BeamStack` #
- `DataStructure BoundedStack` #
- `DataStructure PriorityQueue` #
- `DataStructure Queue` #
- `DataStructure Stack` #

`data BoundedStack a` #

A `BoundedStack` represents a stack whose nodes are depth-bounded: following the same LIFO behavior as a `Stack`, but only the nodes that fulfill the condition *f(n) <= l* are added to the `BoundedStack`, where *f(n)* is the `NodeEvaluation` function used to restrict the structure and *l* is the limit.

Constructors

`BoundedStack`

- `bStackToList :: [Node a]`
- `getEval :: NodeEvaluation a`
- `getLimit :: Double`

Instances

- `DataStructure BoundedStack` #
- `(Hashable a, Eq a) => Eq (BoundedStack a)` #
- `Show a => Show (BoundedStack a)` #

Constructor shortcuts

`newBoundedStack :: Eq a => [Node a] -> NodeEvaluation a -> Double -> BoundedStack a` #

Create a new `BoundedStack` from a list of `Node`s and a limit

`startBoundedStack :: Eq a => a -> NodeEvaluation a -> Double -> BoundedStack a` #

Create a new `BoundedStack` from a state (to be treated as initial node) and a limit

Search.Pure.DataStructure.PriorityQueue

Copyright	(c) Diego Vicente Martín 2017
License	GPL-3
Maintainer	diegovicente@protonmail.com
Stability	experimental
Safe Haskell	Safe
Language	Haskell2010

This module contains a `DataStructure` instance, `PriorityQueue`, that behaves as a sorted structure. `PriorityQueue` is used in the default library as the `DataStructure` for `ucs`, `greedy`, or `aStar`. `PriorityQueue` is based on `IntPSQ`, which let us create a sorted Priority Queue on top of it.

Contents

Instance of `DataStructure`

`class DataStructure ds where` #

When programming new functional search algorithms, one of the most important pieces of the algorithm is the `DataStructure`: It will be the component holding the nodes of the search, and defining its behavior (when to expand a given node) will shape the algorithm. For a type to be defined as a `DataStructure`, some functions have to be defined for it.

Minimal complete definition

`add, addList, next, isEmpty`

Methods

`add :: (Eq a, Hashable a) => Node a -> ds a -> ds a` #

`add` defines how a new node is added to the structure.

`addList :: (Eq a, Hashable a) => [Node a] -> ds a -> ds a` #

`addList` defines how a list of nodes have to be added to the structure.

`next :: (Eq a, Hashable a) => ds a -> (ds a, Node a)` #

`next` defines which node of the structure should be selected next.

`isEmpty :: (Eq a, Hashable a) => ds a -> Bool` #

`isEmpty` checks if the structure has no `Node`s left.

Instances

- `DataStructure BeamStack` #
- `DataStructure BoundedStack` #
- `DataStructure PriorityQueue` #
- `DataStructure Queue` #
- `DataStructure Stack` #

`data PriorityQueue a` #

A `PriorityQueue` represents a sorted queue of nodes, that allows to expand the nodes in a given order nevermind the order in which they were expanded. All new `Node`s are added to its corresponding position (according to the `NodeComparison` function) and the next node returned is the one with the **minimum** value according to the sorting function. It is built on top of a Data.Heap structure, to allow ordered insertion.

Constructors

`PriorityQueue`

- `priorityQueueToHeap :: IntPSQ Double (Node a)`
- `valueFunction :: NodeEvaluation a`

Instances

- `DataStructure PriorityQueue` #
- `(Hashable a, Eq a) => Eq (PriorityQueue a)` #
- `Show a => Show (PriorityQueue a)` #

Constructor shortcuts

`newPriorityQueue :: (Hashable a, Eq a) => [Node a] -> NodeEvaluation a -> PriorityQueue a` #

Create a new `PriorityQueue` from a list of `Node`s and sorted by a `NodeComparison` function

`startPriorityQueue :: (Hashable a, Eq a) => a -> NodeEvaluation a -> PriorityQueue a` #

Create a new `PriorityQueue` from a state (to be treated as initial node)

Search.Pure.DataStructure.Queue

Copyright	(c) Diego Vicente Martín 2017
License	GPL-3
Maintainer	diegovicente@protonmail.com
Stability	experimental
Safe Haskell	Safe
Language	Haskell2010

This module contains a `DataStructure` instance, `Queue`, that behaves as a First-In First-Out structure. `Queue` is used in the default library as the `DataStructure` for `bfs`. `Queue` is built upon the `Data.Sequence` package to allow *O(1)* complexity wen adding elements at the end of the Sequence.

Contents

Instance of `DataStructure`

```
class DataStructure ds where
```

When programming new functional search algorithms, one of the most important pieces of the algorithm is the `DataStructure`: It will be the component holding the nodes of the search, and defining its behavior (when to expand a given node) will shape the algorithm. For a type to be defined as a `DataStructure`, some functions have to be defined for it.

Minimal complete definition

`add, addList, next, isEmpty`

Methods

```
add :: (Eq a, Hashable a) => Node a -> ds a -> ds a
```

`add` defines how a new node is added to the structure.

```
addList :: (Eq a, Hashable a) => [Node a] -> ds a -> ds a
```

`addList` defines how a list of nodes have to be added to the structure.

```
next :: (Eq a, Hashable a) => ds a -> (ds a, Node a)
```

`next` defines which node of the structure should be selected next.

```
isEmpty :: (Eq a, Hashable a) => ds a -> Bool
```

`isEmpty` checks if the structure has no `Node`s left.

Instances

- `DataStructure BeamStack`
- `DataStructure BoundedStack`
- `DataStructure PriorityQueue`
- `DataStructure Queue`
- `DataStructure Stack`

```
newtype Queue a
```

A `Queue` represents a First-In First-Out structure: Each new `Node` is added to the back of the `Queue`, and when extracting the next `Node` the first one that was added is returned

Constructors

```
Queue
    queueToSeq :: Seq (Node a)
```

Instances

- `DataStructure Queue`
- `(Eq a, Hashable a) => Eq (Queue a)`
- `Show a => Show (Queue a)`

Constructor shortcuts

```
newQueue :: (Hashable a, Eq a) => [Node a] -> Queue a
```

Create a new `Queue` from a list of `Node`s

```
startQueue :: (Hashable a, Eq a) => a -> Queue a
```

Create a new `Queue` from a state (to be treated as initial node)

Search.Pure.DataStructure.Stack

Copyright	(c) Diego Vicente Martín 2017
License	GPL-3
Maintainer	diegovicente@protonmail.com
Stability	experimental
Safe Haskell	Safe
Language	Haskell2010

This module contains a `DataStructure` instance, `Stack`, that behaves as a Last-In First-Out structure. `Stack` is used in the default library as the `DataStructure` for `dfs`. `Stack` is based on a regular list since it is *O(1)* new nodes to the beginning of a list.

Contents

Instance of `DataStructure`

`class DataStructure ds where` #

When programming new functional search algorithms, one of the most important pieces of the algorithm is the `DataStructure`: It will be the component holding the nodes of the search, and defining its behavior (when to expand a given node) will shape the algorithm. For a type to be defined as a `DataStructure`, some functions have to be defined for it.

Minimal complete definition

`add, addList, next, isEmpty`

Methods

`add :: (Eq a, Hashable a) => Node a -> ds a -> ds a` #

`add` defines how a new node is added to the structure.

`addList :: (Eq a, Hashable a) => [Node a] -> ds a -> ds a` #

`addList` defines how a list of nodes have to be added to the structure.

`next :: (Eq a, Hashable a) => ds a -> (ds a, Node a)` #

`next` defines which node of the structure should be selected next.

`isEmpty :: (Eq a, Hashable a) => ds a -> Bool` #

`isEmpty` checks if the structure has no `Node`s left.

Instances

- `DataStructure BeamStack` #
- `DataStructure BoundedStack` #
- `DataStructure PriorityQueue` #
- `DataStructure Queue` #
- `DataStructure Stack` #

`newtype Stack a` #

A `Stack` represents a Last-In First-Out structure: Each new `Node` is added to the beginning of the `Stack`, and when popping the next `Node` to be expanded the last one in is returned

Constructors

`Stack`

`stackToList :: [Node a]`

Instances

- `DataStructure Stack` #
- `(Eq a, Hashable a) => Eq (Stack a)` #
- `Show a => Show (Stack a)` #

Constructor shortcuts

`newStack :: (Hashable a, Eq a) => [Node a] -> Stack a` #

Create a new `Stack` from a list of `Node`s

`startStack :: (Hashable a, Eq a) => a -> Stack a` #

Create a new `Stack` from a state (to be treated as initial node)

Search.Pure.General

Copyright	(c) Diego Vicente Martín 2017
License	GPL-3
Maintainer	diegovicente@protonmail.com
Stability	experimental
Safe Haskell	Safe
Language	Haskell2010

Contents

This module contains the general method to perform searches: `generalSearch`. This method can be considered the generalizations of search algorithms, and different search behaviors can be implemented by passing different `DataStructures` to it. Also, due to the high memory consumption it generates by actually storing the nodes in a structure, the methods `depthSearch` and `limitedDepthSearch` are also provided, which perform the search by ordering recursive stack calls (and thus using linear memory). These methods offer a more limited control over the execution. Furthermore, an implementation of the Depth-First Branch & Bound is provided using folds in `depthBNB`.

`generalSearch` is inspired by the `GENERAL-SEARCH` algorithm proposed in Russel & Norvig's *Artificial Intelligence: A Modern Approach*. However, taking advantage of Haskell's lazy evaluation, the algorithm keeps on evaluating nodes even after a first solution is found, returning the list of all solutions existing in the search space. This feature is also present in `depthSearch` and `limitedDepthSearch`. Please notice that, depending on the nature of the problem, this list may be infinite. On the other hand, the list returned by `depthBNB` returns only the solutions in the search space, that (due to the nature of the algorithm) are not all the solutions to be found in the problem space.

General Search

generalSearch

`:: (DataStructure ds, Eq a, Hashable a)`	
`=> ProblemSpace a`	`ProblemSpace` to be solved
`-> Cost a`	`Cost` function to use
`-> Heuristic a`	`Heuristic` function to use
`-> ds a`	`DataStructure` that manages the node expansion
`-> [Node a]`	Returns the list of all final nodes (solutions)

`generalSearch` performs a brute force search traversing a tree. Receives a problem space used to model the problem, several evaluation functions and a `DataStructure` that holds the nodes. The behavior of the algorithm is dictated by the order in which the `DataStructure` makes the nodes be expanded.

depthSearch

`:: (Eq a, Hashable a)`	
`=> ProblemSpace a`	`ProblemSpace` to be solved
`-> Cost a`	`Cost` function to use
`-> Heuristic a`	`Heuristic` function to use
`-> NodeEvaluation a`	`NodeEvaluation` to sort the expanded nodes
`-> Node a`	Current `Node` to be expanded
`-> [Node a]`	Returns the list of all final nodes (solutions)

`depthSearch` performs a search without an explicit `DataStructure`, instead it performs a recursion tree that enables backtracking to the nodes expanded. Nodes expanded in each call can be sorted with a given `NodeEvaluation` function, or this feature can be ignored by passing `noSorting` dummy function to the function.

limitedDepthSearch

`:: (Eq a, Hashable a)`	
`=> ProblemSpace a`	`ProblemSpace` to be solved
`-> Cost a`	`Cost` function to use
`-> Heuristic a`	`Heuristic` function to use
`-> NodeEvaluation a`	`NodeEvaluation` to sort the expanded nodes
`-> Double`	Limit to be imposed to the `NodeEvaluation`
`-> Node a`	Current `Node` to be expanded
`-> [Node a]`	Returns the list of all final nodes (solutions)

`limitedDepthSearch` performs a search in similar way of `depthSearch`, but imposing a given limit on the value returned by the `NodeEvaluation` function. If the value of `f node` is larger than the provided limit, the node expanded is not recursively called upon but ignored.

depthBNB

`:: (Eq a, Hashable a)`	
`=> ProblemSpace a`	`ProblemSpace` to be solved
`-> Cost a`	`Cost` function to use
`-> Heuristic a`	`Heuristic` function to use
`-> NodeEvaluation a`	`NodeEvaluation` to sort and bound the expanded nodes
`-> Node a`	Current `Node` to be expanded

`-> (Double, [Node a])`	The current bound and intermediate solutions found (in ascending cost order)
`-> (Double, [Node a])`	The final bound and all solutions found (in ascending cost order)

`depthBNB` is a recursive function to perform Branch & Bound using folds. The accumulator of the fold stores both the lowest bound currently found and the list of solutions found (in a First In, Last Out fashion).

`expand` #

`:: (Eq a, Hashable a)`	
`=> Node a`	`Node` to be expanded
`-> Cost a`	`Cost` function to update the new nodes
`-> Heuristic a`	`Heuristic` function to update the new nodes
`-> [Operator a]`	List of actions to apply on the node
`-> [Node a]`	List of all the valid `Node`s expanded

`expand` function applies all the actions of the problem and checks if the node has been previously visited in the `Node` current path. This function is used in all the search methods implmented above.

Dummy functions

Dummy functions are used when no `Heuristic`, `Cost` and/or `NodeEvaluation` function is required (i.e uninformed algorithms like `bfs`). In that case, the function passed to the appropriate search method. Dummy functions are only shortcuts to returning a constant.

`noCost :: Cost a` #

Dummy `Cost` function.

`noHeuristic :: Heuristic a` #

Dummy `Heuristic` function.

`noSorting :: NodeEvaluation a` #

Dummy sorting for `depthSearch`.

Search.Pure.Informed

Copyright	(c) Diego Vicente Martín 2017
License	GPL-3
Maintainer	diegovicente@protonmail.com
Stability	experimental
Safe Haskell	Safe
Language	Haskell2010

This module contains different informed search algorithms ready to be used.

Documentation

```
hillClimbing :: (Eq a, Hashable a) => Cost a -> Heuristic a -> Algorithm a
```
#

`hillClimbing` runs a HillClimbing Heuristic Search.

```
aStar :: (Eq a, Hashable a) => Cost a -> Heuristic a -> Algorithm a
```
#

`aStar` runs an A* Search.

```
idAStar :: (Eq a, Hashable a) => (Double, Double) -> Cost a -> Heuristic a -> Algorithm a
```
#

`idAStar` runs an Iterative-Deepening A* Search.

```
beam :: (Eq a, Hashable a) => Int -> Cost a -> Heuristic a -> Algorithm a
```
#

`beam` runs a Beam Search of a given beam width.

```
dfBNB :: (Eq a, Hashable a) => Cost a -> Heuristic a -> Algorithm a
```
#

`dfBNB` performs a Depth-First Branch & Bound Search. Due to the nature of this algorithm, it does not return the list of all solutions in the problem space: Instead, it returns all the solutions that it has found in ascending cost order.

Search.Pure.ToyProblem.EightPuzzle

Copyright	(c) Diego Vicente Martín 2017
License	GPL-3
Maintainer	diegovicente@protonmail.com
Stability	experimental
Safe Haskell	None
Language	Haskell2010

This module contains a way to approach the 8-Puzzle problem using the library: by understanding the blank tile as an agent that is able to move accross the board, we can try to search for actions that we have to perform (move the blank tile to a certain direction) to solve the puzzle. A way to find the shortest solution is to run a Breadth-First Search on it.

Contents

The 8-Puzzle problem

Types

`type Coord = (Int, Int)` #

`Coord` helps us express a coordinate in 2D

`type Puzzle = Vector Int` #

`Puzzle` is represented as a vector of Ints, which allows us to update it easily

`type Index = Int` #

`Index` helps us express an index in 1D

`data Direction` #

`Direction` is used to define the direction of the movements

Problem parts

`easy :: Puzzle` #

`initial` is the initial board for tests

`hard :: Puzzle` #

`hard` is a puzzle can be solved in 17 movements

`hamming :: Puzzle -> Puzzle -> Int` #

`hamming` returns the Hamming value of a puzzle: the number of tiles that are not correctly positioned.

`swapTiles :: Puzzle -> Coord -> Coord -> Puzzle` #

`swapTiles` swaps two tiles (identified by its coordinates)

`moveBlank :: Direction -> Puzzle -> Maybe Puzzle` #

`moveBlank` moves the blank tile if the direction is possible, or returns `Nothing` if not

`isFinal :: Puzzle -> Bool` #

`isFinal` checks if the puzzle is in a goal state. An 8-Puzzle is solved if all its tiles are in order (despite the blank)

`puzzleHeuristic :: Heuristic Puzzle` #

`puzzleHeuristic` generates a `Heuristic` function using `hamming`

Interfaces to the Library

`buildProblemSpace :: Puzzle -> ProblemSpace Puzzle` #

`buildProblemSpace` receives a initial state of the `Puzzle` and returns a complete `ProblemSpace` to be solved.

`nStepsUninformed :: Puzzle -> Algorithm Puzzle -> Maybe Int` #

`nStepsUninformed` solves a given `Puzzle` using an uninformed `Algorithm` and returns the number of steps that it took to solve.

`nStepsInformed :: Puzzle -> (Cost Puzzle -> Heuristic Puzzle -> Algorithm Puzzle) -> Maybe Int` #

Search.Pure.ToyProblem.MapParser

Copyright	(c) Diego Vicente Martín 2017
License	GPL-3
Maintainer	diegovicente@protonmail.com
Stability	experimental
Safe Haskell	None
Language	Haskell2010

This module contains an implementation of Search that is able to be parse a Moving AI map (as provided in http://www.movingai.com/benchmarks/wc3maps512/) and perform the shortest route available between two points given.

Contents

Moving AI Routing

Types

`type Coord = (Int, Int)` #

`Coord` type represents a pair of coordinates in the maze

`data Direction` #

`Direction` represents the available directions

Instances

- `Eq Direction` #
- `Show Direction` #

`type Map = Vector (Vector Char)` #

A `Map` is a multi-dimensional `Vector` of characters.

Problem Parts

`parse :: [String] -> Map` #

`parse` is a function dedicated to parse Moving AI map strings into a `Map`.

`getCell :: Map -> Coord -> Char` #

`getCell` returns the value of a `Coord` in a given `Map`

`cellCost :: Char -> Double` #

`cellCost` returns the cost of a cell's value.

`move :: Map -> Direction -> Coord -> Maybe Coord` #

`move` returns a new `Coord` if it was possible to move from an initial position to a destination using a `Direction` in a given `Map`

`manhattan :: Coord -> Coord -> Double` #

`manhattan` computes the Manhattan distance between two points

Default maps

`small :: [String]` #

An easy maze, used for several testing purposes, defined in a 5x5 grid.

`medium :: [String]` #

A medium sized maze, defined in a 10x10 grid.

`big :: [String]` #

A medium sized maze, defined in a 15x15 grid.

Interfaces to the Library

`costFunction :: Map -> Cost Coord` #

`costFunction` is able to return the cost of a movement in a `Map`.

`buildProblemSpace :: Map -> Coord -> Coord -> ProblemSpace Coord` #

`buildProblemspace` generates a new `ProblemSpace` by receiving a `Map` and an initial and final `Coord`.

`solveUninformed :: Map -> Coord -> Coord -> Algorithm Coord -> Maybe (Node Coord)` #

`solveUninformed` finds a route between two given `Coord` in a `Map` using a given uninformed `Algorithm`.

`solveInformed :: Map -> Coord -> Coord -> (Cost Coord -> Heuristic Coord -> Algorithm Coord) -> Maybe (Node Coord)` #

`solveInformed` finds a route between two given `Coord` in a `Map` using a given informed `Algorithm`.

Input/Output

`readMap :: FilePath -> IO Map` #

`readMap` receives a `FileName` and reads it using `parse` to return a `Map`.

`route :: FilePath -> Coord -> Coord -> IO ()` #

`route` finds a the shortest path between two points with a A* Search in the map stored in `FilePath`. It prints in the screen the cost of the path found.

`timing :: IO ()` #

Search.Pure.ToyProblem.NQueens

Copyright	(c) Diego Vicente Martín 2017
License	GPL-3
Maintainer	diegovicente@protonmail.com
Stability	experimental
Safe Haskell	None
Language	Haskell2010

Contents

This module contains a way to approach the N-Queens problem using the library: to do so we understand the actions of the problem as adding a new queen to next column possible and a given row. That way the problem is modelled as a tree, that can be search using for example a Depth-First Search.

An example of how the solution is displayed is:

```
>>> solveQueens dfs
♕ . . . . . . .
. . . . ♕ . . .
. . . . . . . ♕
. . . . . ♕ . .
. . ♕ . . . . .
. . . . . . ♕ .
. ♕ . . . . . .
. . . ♕ . . . .
```

The N-Queens problem

Types

`type Coord = (Int, Int)` #

`Coord` type represents a position in the board

`type Queens = [Coord]` #

`Queens` is a list of `Coord` that represent the position of each queen

Problem parts

`isValid :: Queens -> Coord -> Bool` #

`isValid` checks if the new queen proposed is valid or not by checking conflicts with the existing queens in the board

`isFinal :: Queens -> Bool` #

`isFinal` checks if a given board is a goal state. The problem is over if all queens have been placed with no conflicts

`conflicts :: Queens -> Int` #

`conflicts` return the number of queens that can attack other queens

`newQueen :: Int -> Queens -> Maybe Queens` #

`newQueen` adds a queen in the first column available in the right, or returns `Nothing` if there is a conflict with an existing queen

`showQueens :: Queens -> IO ()` #

`showQueens` prints the board in a readable way: where `.` represents an empty cell and ♕ represents a queen (there's an example in the module description)

Interfaces to the Library

`queens :: ProblemSpace Queens` #

The N-Queens problem modelled as a `ProblemSpace`

`solveQueens :: Algorithm Queens -> IO ()` #

`solveQueens` solves `queens` and displays its result as found (using `showQueens`)

`getCoords :: Algorithm Queens -> [Coord]` #

Extract the coordinates for testing purposes

`countSolutions :: Algorithm Queens -> Int` #

`countSolutions` counts how many different solutions the given `Algorithm` is able to find in the problem

Search.Pure.Uninformed

Copyright	(c) Diego Vicente Martín 2017
License	GPL-3
Maintainer	diegovicente@protonmail.com
Stability	experimental
Safe Haskell	Safe
Language	Haskell2010

This module contains different uninformed search algorithms ready to be used.

Contents

Uninformed Search Algorithms

`bfs :: (Eq a, Hashable a) => Algorithm a` #

`bfs` runs a Breadth-First Search

`dfs :: (Eq a, Hashable a) => Algorithm a` #

`dfs` runs a Depth-First Search

`idfs :: (Eq a, Hashable a) => (Int, Int) -> Algorithm a` #

`idfs` runs an Iterative-Deepening Depth-First Search. The first argument, a pair of `Int`s (`step`, `inf`), represent the main parameters of the search: each new iteration the depth test is incremented by adding `step` as long as the new depth is lower than `inf`.

`ucs :: (Eq a, Hashable a) => Cost a -> Algorithm a` #

`ucs` runs an Uniform-Cost Search with a given cost function

Dummy functions

`noCost :: Cost a` #

Dummy `Cost` function.

`noHeuristic :: Heuristic a` #

Dummy `Heuristic` function.

`noSorting :: NodeEvaluation a` #

Dummy sorting for `depthSearch`.