Find Max Pairing In Bipartite Graph

Pairing Modules

`parovani :: [(String, [String])] -> [(String, String)]`

- Takes a bipartite graph and finds the maximal pairing in it
  - **INPUT**: `[(node from first partite v, [nodes from second partite, to which goes edge from v])]`
  - **RETURNS**: list of all pairs, first from the first partite, second from second

GraphDataStructures Modules

Module used for data-structures like Graphs, Nets, Queues and so on used by the PairingUtils Module

`GrafHrana a`

- **Type**: `GrafHrana a = (a, Int, Int)`
- **Description**: Represents an edge in the graph, where `a` is the vertex, and the two `Int` values represent flow and capacity.

`Graf a`

- **Type**: `Graf a = [(a, [GrafHrana a])]`
- **Description**: Represents a graph as a list of vertices, each with associated outgoing edges.

`RezervaHrana a`

- **Type**: `RezervaHrana a = (a, Int, Int, Int)`
- **Description**: Represents a residual edge in the flow network, including capacity and forward and backward flow.

`Rezervy a`

- **Type**: `Rezervy a = [(a, [RezervaHrana a])]`
- **Description**: Represents the residual graph, tracking vertices, edges, and reachability.

`Path a`

- **Type**: `Path a = [RezervaHrana a]`
- **Description**: A path within the residual graph, composed of a list of `RezervaHrana` edges.

Queue Interface and Implementation

The `Queue` class defines a generic interface for queue operations, which is implemented by the `SQueue` data type.

`Queue` Class

```
class Queue q where
  emptyQueue :: q a
  isEmpty :: q a -> Bool
  enqueue :: a -> q a -> q a
  dequeue :: q a -> (a, q a)
  enqueueLs :: [a] -> q a -> q a
```

- **Description**: Defines a generic queue interface with operations for checking if the queue is empty, adding single or multiple elements, and removing elements.

`SQueue` Data Type

```
data SQueue a = SQueue [a] [a]
```

- **Description**: A simple queue implementation using two lists for efficient enqueue and dequeue operations.

`SQueue` Instances

- `Eq` **Instance**: Compares two `SQueue` instances for equality by comparing their elements in order.
- `Queue` **Instance**: Provides the implementation of the `Queue` interface for `SQueue`.
- `Show` **Instance**: Displays the queue elements in a readable format.
- `Functor` **Instance**: Allows applying a function to all elements in the queue.

Example Usage

Creating and Manipulating a Queue

```
-- Create an empty queue and enqueue elements
let qs = enqueue 7 (enqueue 5 (enqueue 3 emptyQueue)) :: SQue
ue Int

-- Dequeue an element
```

```
let (element, qs2) = dequeue qs

-- Enqueue multiple elements
let qs3 = enqueueLs [1, 2, 3] qs2
```

Using `queue_of_nums`

```
-- Generate a queue of numbers from x to y
let numQueue = queue_of_nums 3 1000 :: SQueue Int
```

PairingUtil Module

This module implements algorithm used for the max pairing finding.

`bfsPath`

```
bfsPath :: (Eq a, Show a) => Rezervy a -> a -> a -> (Int, Pat
h a)
```

- **Description**: Implements BFS to find an unsaturated path from a source to a sink in the residual graph, returning the path and its minimum capacity.

`rozsirGraf`

```
rozsirGraf :: [(String, [String])] -> Graf String
```

- **Description**: Expands a bipartite graph by adding a source and sink vertex, preparing it for the Ford-Fulkerson algorithm.

`grafnapary`

```
grafnapary :: Graf String -> [(String, String)]
```

- **Description**: Converts a flow network into a list of vertex pairs, representing the maximal pairing in the bipartite graph.

`fordfalk`

```
haskellZkopírovat kód
fordfalk :: (Eq a, Show a) => Graf a -> a -> a -> Graf a
```

- **Description**: Executes the Ford-Fulkerson algorithm on the input graph to compute the maximum flow, returning the final flow network.

Utility

`trace'`

```
trace' :: (Show a) => a -> a
```

- **Description**: A debugging utility function that logs values when debugging is enabled. Controlled by the `is_debuging` flag.

`resetReached`

```
resetReached :: Rezervy a -> Rezervy a
```

- **Description**: Resets the "reached" status of vertices in the residual graph.

`getNextV`

```
getNextV :: (Eq a, Show a) => Rezervy a -> a -> Path a -> [Re
zervaHrana a]
```

- **Description**: Finds the next vertices in the residual network that are not on a saturated edge, filtering them based on the current path.
  - **INPUT**: Graph, starting vertex, current path
  - **RETURNS**: next edges that can be taken in traversal

`ffKrok`

```
ffKrok :: (Eq a, Show a) => Rezervy a -> a -> (Int, Path a) -
> Rezervy a
```

- **Description**: Performs one step of the Ford-Fulkerson method by augmenting the flow along an unsaturated path found by BFS.

`updateTok`

```
updateTok :: (Eq a, Show a) => Rezervy a -> a -> RezervaHrana
a -> Int -> Int -> Rezervy a
```

- **Description**: Updates the flow along a specific edge in the residual graph.

`bfsBody`

```
bfsBody :: (Eq a, Show a) => Rezervy a -> a -> SQueue (a, Pat
h a, Int) -> (Int, Path a)
```

- **Description**: The core loop of the BFS algorithm, processing each vertex and updating the queue with reachable vertices.

`fordfalk'`

```
fordfalk' :: (Eq a, Show a) => Rezervy a -> a -> a -> Rezervy
a
```

- **Description**: Recursively applies the Ford-Fulkerson method to find the maximum flow in the graph by repeatedly finding augmenting paths and updating the residual graph.

Graph Conversion

`rezervy`

```
rezervy :: Graf a -> Rezervy a
```

- **Description**: Converts a graph into its residual graph representation, where each edge tracks its capacity and flow.

`grafy`

```
grafy :: Rezervy a -> Graf a
```

- **Description**: Converts a residual graph back into the standard graph format.