Exploring Non-Determinism in Graph Algorithms

Nikita Danilenko

Institut für Informatik, Christian-Albrechts-Universität Kiel
Olshausenstraße 40, D-24098 Kiel
nda@informatik.uni-kiel.de

Abstract. Graph algorithms that are based on the computation of one or more paths are often written in an implicitly non-deterministic way, which suggests that the result of the algorithm does not depend on a particular path, but any path that satisfies a given property. Such algorithms provide an additional challenge in typical implementations, because one needs to replace the non-determinism with an actual implementation. In this paper we explore the effects of using non-determinism explicitly in the functional logic programming language Curry. To that end we consider three algorithms and implement them in a prototypically non-deterministic fashion.

1 Introduction

Consider a graph $G = (V, E)$, two vertices $s, t \in V$ and the question, whether there is a cycle in G that contains both vertices (i.e. both are contained in the same strong connected component). It is easy to come up with a solution for this task – simply check whether there are paths from s to t and from t to s in G, if so, the answer is "yes", otherwise it is "no". Now all one needs to do is to come up with how to specify the existence of paths, which is just as straight-forward. Next, one could ask for an actual cycle in case such a cycle exists. For every path p from s to t and every path q from t to s, the composition of p and q is such a cycle, i.e. the existence of a cycle does not depend on a particular choice of a path and there may be several different cycles.

When the above problems are considered in light of logic programming, one can use non-determinism to express the independence of choice and to compute one or more results from the specification alone. Aside from the usual benefit of the declarative approach of what to compute and not how, one can also observe that the different intermediate results (cycles or paths) yield the same overall answer (yes or no). While this is somewhat trivial in the above example, there are more sophisticated graph algorithms that rely on the computation of *some* value with a special property and it may not be obvious at all that different choices of such values yield the same results in the end. This invariance can be considered as a certain kind of confluence and the possibility to observe this invariance is a useful tool, particularly in teaching.

In this paper we consider graph algorithms that are based on the computation of paths. We begin with a simple observation relating the path search strategy to the solution search strategies and proceed to provide solutions to two non-trivial graph problems, namely the *maximum matching problem* and the *maximal flow problem* and provide examples of matchings and flows. Our main focus is on experimentation and applications in teaching, where theoretical results are likely to be followed by examples. Observing implicit non-determinism explicitly is of great assistance in this context since one can witness the invariance of the solution with respect to different choices that lead to it.

All code in this paper is written in Curry [12] and compiled with KiCS2 [5], which can be obtained at `http://www-ps.informatik.uni-kiel.de/kics2/`. Throughout the paper we refer to several functions and modules, all of which can be found using the search engine Curr(y)gle `https://www-ps.informatik.uni-kiel.de/kics2/currygle/`. A polished version of the presented code is available at GitHub under `https://github.com/nikitaDanilenko/ndga`.

2 Preliminaries

A graph is a pair $G = (V, E)$, where V is a non-empty finite set and $E \subseteq V \times V$. This is to say that we consider all graphs to be directed and realise graphs in which the direction does not matter as symmetric directed graphs. For any $v \in V$ we set $N_{\rightarrow}(v) := \{\, w \in V \mid (v, w) \in E \,\}$ and $N_{\leftarrow}(v) := \{\, w \in V \mid (w, v) \in E \,\}$; elements of $N_{\rightarrow}(v)$ are called *successors* and those of $N_{\leftarrow}(v)$ *predecessors* of v. A path in a graph is an injective sequence of vertices, i.e. a sequence that does not contain multiple occurrences of the same vertex. Since V is finite, every path traverses at most $|V|$ vertices. We represent vertices by integers, edges by pairs of vertices and paths by lists of distinct vertices, where the distinctness is required implicitly. We discuss this design decision in Section 3.

```
type Vertex = Int
type Edge   = (Vertex, Vertex)
type Path   = [Vertex]
```

Graphs are represented by an adjacency list model, where the adjacency lists are sorted in ascending natural order of the integers. These adjacency lists are stored in a finite map, one implementation of which is provided in the standard KiCS2 library *FiniteMap* as the data structure *FM key value*.

```
data Graph = Graph (FM Vertex [Vertex])
```

Additionally, we use sets of vertices explicitly for the maintenance of visited vertices. We use finite maps provided by KiCS2 for a representation of sets of vertices, where the functions *emptyFM*, *addToFM*, *elemFM*, *delFromFM* are provided, too. The function *emptyFM* is parametrised over an irreflexive order predicate which in our case is $(<) :: Int \rightarrow Int \rightarrow Bool$.

```
type VertexSet = FM Vertex ()
```

```
empty :: VertexSet
empty = emptyFM (<)

insert :: Vertex → VertexSet → VertexSet
insert i m = addToFM m i ()

inSet :: Vertex → VertexSet → Bool
inSet = elemFM

remove :: Vertex → VertexSet → VertexSet
remove = flip delFromFM

vertexListToSet :: [Vertex] → VertexSet
vertexListToSet = foldr insert empty
```

For the purpose of demonstration we use the two example graphs from the Figures 1, 2 in the following two sections and assume that they are implemented in the values *graph1* and *graph2* respectively.

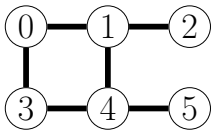

Fig. 1. Example graph G_1

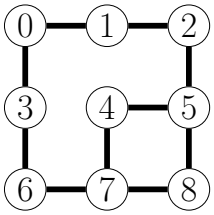

Fig. 2. Example graph G_2

3 Reachability and Paths

One standard example of logic programming in graph theory is the check whether a vertex set ts is reachable from a given vertex s. This is the case iff $s \in ts$ or if there is a successor $i \in V$ of s, such that ts is reachable from i. Since successors can lead back to already visited vertices, one additionally has to check, if the successor has been visited or not[1]. The implementation of the above may be as follows.

```
reachable :: Graph → Vertex → VertexSet → Success
reachable g from ts = find empty from where
  find vis s | s `inSet` ts                          = success
             | isEdge g s i ∧ ¬ (i `inSet` vis) = find (insert s vis) i where i free
```

This implementation contains a high level of abstraction and is very close to the original specification. Also, instead of defining how to access the successors of

[1] Avoiding this check can easily result in non-termination: consider the graph $(\{0,1,2\},\{(0,1),(1,0),(0,2)\})$ with DFS. Then checking whether 2 is reachable from 0 diverges, because the loop $(0,1,0)$ is entered before checking the other successors of 0. A similar example shows the non-termination in case of using BFS.

s, this implementation relies only on a test whether a certain edge is contained in the graph and the use of a free variable to find a fitting edge. The technique of translating existential quantifications into free variables is common in logic programming.

Instead of just checking for the existence of a path, one can ask for an actual path between two vertices. It is easy to implement such a search by modifying the above function.

```
path :: Graph → Vertex → VertexSet → Path
path g from ts = find empty from where
  find vis s | s `inSet` ts                    = [s]
             | isEdge g s i ∧ ¬ (i `inSet` vis) = s : find (insert s vis) i where i free
pathToSingle :: Graph → Vertex → Vertex → Path
pathToSingle g from t = path g from (vertexListToSet [t])
```

Again, the function is non-deterministic in its choice of the successor. There are two interesting consequences of this implementation. First of all, using the interactive mode of KiCS2 one can find all paths between two given vertices without any additional work. Alternatively, when all paths are required in a program, one can use set functions [2] to encapsulate the search and collect all results. Set functions are provided in the KiCS2 library *SetFunctions*, which provides functions *setK* (for $K \in \{0, \ldots, 7\}$)

```
setK :: (a0 → .. → aK−1 → b) → a0 → .. → aK−1 → Values b
```

which turn a function of a given arity into a set function, where *Values b* is the list of all results. Assuming that the graph from Figure 2 is implemented as *graph2* we can test the following results.

```
kics2> set3 path graph2 0 (vertexListToSet [5,8])
(Values [[0,1,2,5],[0,3,6,7,4,5],[0,3,6,7,8]])
kics2> set3 pathToSingle graph2 0 8
(Values [[0,1,2,5,4,7,8],[0,1,2,5,8],[0,3,6,7,4,5,8],[0,3,6,7,8]])
```

The second consequence is that the path search is based upon the order in which the free variable i is instantiated, which in turn depends on the search strategy. The search strategy can be chosen in KiCS2 either interactively (by using :*set STRATEGY*) or directly in the code. The latter can be accomplished by explicitly representing the search space as a *SearchTree* from the homonymous Curry module and using strategy dependent operations on the tree to obtain actual results. For example, we get:

```
kics2> someValueWith bfsStrategy (pathToSingle graph1 2 5)
[2,1,4,5]
kics2> someValueWith dfsStrategy (pathToSingle graph1 2 5)
[2,1,0,3,4,5]
```

Note that these two results correspond precisely to what the respective graph-theoretic strategies yield. In fact, in the above implementation one can decide

between choosing a successor and descending the recursive call (DFS) or checking other successors first and descending afterwards (BFS). However, both strategies will find all paths eventually, so that the above observation holds for the first result, but not necessarily for subsequent results (this can be observed using set functions, but not with *someValueWith*, because the latter is deterministic).

```
kics2> set3With bfsStrategy pathToSingle graph1 2 5
(Values [[2,1,4,5],[2,1,0,3,4,5]])
kics2> set3With dfsStrategy pathToSingle graph1 2 5
(Values [[2,1,0,3,4,5],[2,1,4,5]])
```

Since we check whether a vertex has been visited before by hand, the condition that the vertices of every resulting *Path* are distinct is maintained. Using an implementation based upon the constrained constructor pattern one can disallow the creation of invalid paths. However, the *search* itself requires access to previously visited vertices to avoid (infinite) repetition and thus its implementation would remain essentially the same as above. We omit the smart constructor approach for the sake of simplicity.

4 Maximum Matchings

A more sophisticated application of non-deterministic path search is the algorithm for finding maximum matchings. A matching in a symmetric graph is a set $M \subseteq E$ that is symmetric and functional[2]. In other words a matching consists of edges that do not share common vertices. A matching is called *maximum matching* iff there is no matching with a strictly greater cardinality. Clearly, maximum matchings are not necessarily unique, since they are maximal elements with respect to cardinality. Figure 3 shows some examples of matchings in the example graph from Figure 2, where the bold lines show matching edges and the dashed lines are non-matching edges.

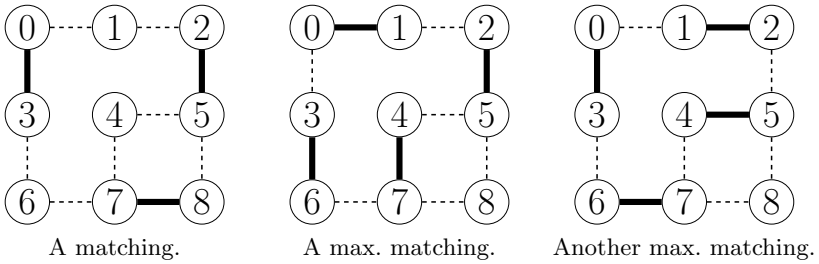

Fig. 3. Examples of matchings.

[2] I.e. $\forall\, x, y, z \in V : (x, y) \in M \wedge (x, z) \in M \Rightarrow y = z$.

There are several natural applications for maximum matchings, like assigning jobs to people, processes to machines or, more classically, spouses to one another. In all cases one may wish to make as many one-to-one assignments as possible, which translates directly into the search for a maximum matching. All of these examples can be modelled with so-called *bipartite* graphs, which are graphs that allow a partition of vertices into two sets, such that all edges of the graph connect only vertices from two different sets. For bipartite graphs there is a simple imperative algorithm that computes maximum matchings, while the algorithms for the non-bipartite case is significantly more sophisticated [9].

The Berge theorem [3] characterises maximum matchings and provides an algorithm for finding such matchings. We state it exactly as in [6].

Theorem 1 (Characterisation of maximum matchings, Berge).
Let $M \subseteq E$ be a matching. Let $\oplus$ denote the symmetric difference. For a path p we denote the set of the edges along p by $E(p)$. A path is called M-augmenting iff it starts and ends in a vertex that is not contained in some edge of M and alternates between edges of $E \setminus M$ and those of M. Then we have

1. *If there are no M-augmenting paths in G, then M is a maximum matching.*
2. *If there is an M-augmenting path p in G, then $M \oplus E(p)$ is a larger matching.*

This theorem can be used for an actual computation of a maximum matching. Starting with the empty matching one searches for an augmenting path and if such a path exists the current matching is expanded. If on the other hand no such path exists, the current matching is already a maximum matching.

For the remainder of this section we assume all graphs to be symmetric, in particular this concerns all graph arguments in functions. This condition can (and should!) be checked in a proper application, but to avoid additional code that is not necessary for the presentation, we assume this check to have been performed beforehand implicitly.

Let us first deal with the matching augmentation. Given a matching M and an augmenting path p Berge's theorem states that $M \oplus E(p)$ is a larger matching. Once we have computed this matching, we will check whether there is an augmenting path, using $M \oplus E(p)$ instead of M, which requires the computation of $E \setminus (M \oplus E(p))$. As we have stated in [6] it is simple to see that

$$E \setminus (M \oplus E(p)) = (E \setminus M) \oplus E(p) \ ,$$

so that both M and $E \setminus M$ are updated in the same fashion. Suppose that

$$\mathit{xorBiEdges} :: \mathit{Graph} \to [\mathit{Edge}] \to \mathit{Graph}$$

traverses a list of edges and for every edge it adds its undirected version (i.e. both directions) to the graph if the edge is not already present in the graph and removes both directions otherwise. Then the augmentation update can be implemented as follows, where we assume that m is the current matching and notM is its complement in E (i.e. $\mathit{notM} = E \setminus m$).

```
augmentBy :: Path → Graph → Graph → (Graph, Graph)
augmentBy path m notM = (m `xorBiEdges` es, notM `xorBiEdges` es) where
  es = zip path (tail path)
```

Without logic means checking the existence of an augmenting path and finding such a path in the positive case is quite technical[3]. Using logic means however, we can specify an augmenting path by first dealing with alternating paths and then adding the conditions for the first and last vertex. Suppose we have a target set of vertices $ts :: VertexSet$, a starting vertex s and a list of graphs $(g : gs) :: [Graph]$ that we want to traverse in sequence. Then an alternating path from s to ts through $g : gs$ exists iff $s \in ts$ holds or there is a successor i of s in g and there exists an alternating path from i to ts through $gs +\!\!+ [g]$. The intention is that we will use the list $[notM, m]$ in our application, where m is the current matching and $notM$ is its complement in E. Except for the traversal of multiple graphs in a cyclic order[4], the above specification is very similar to the one that specified the existence of a path. Similarly, the corresponding function is easy to modify to return an alternating path as well.

```
alternatingPath :: [Graph] → Vertex → VertexSet → Path
alternatingPath grs from ts = find empty from grs where
  find vis s (g : gs)
    | s `inSet` ts                    = [s]
    | isEdge g s i ∧ ¬ (i `inSet` vis) = s : find (insert s vis) i (gs +++ [g]) where i free
```

With this function we can find augmenting paths, too. Let us assume that we have a function at hand that computes those vertices of a symmetric graph which do not have any neighbours (i.e. successors, due to symmetry).

```
noSuccessors :: Graph → VertexSet
```

By definition an augmenting path starts and ends in a vertex that is not covered by the matching and since it is a path the start vertex should also be different from the end vertex, which is not guaranteed by the *alternatingPath* function from above. Fortunately, this is easily corrected by simply excluding the start vertex from the target vertex set. Again, we assume that m is a matching and $notM$ is its complement in E.

```
augmentingPath :: Graph → Graph → Path
augmentingPath m notM | s `inSet` unc = alternatingPath [notM, m] s (s `remove` unc)
  where unc = noSuccessors m
        s free
```

This function is prototypical by design – guessing a possible start vertex and trying to find a path is obviously not the most efficient way of finding an augmenting path. Instead, one could either modify the search function in a fashion

[3] The usual procedure is to implement a modified breadth-first search, which explicitly alternates between two graphs.

[4] We have implemented the cyclic list traversal inefficiently by adding the first element to the end of the list for demonstration purposes only. It is simple to replace this implementation by either functional lists or queues, both of which allow adding an element to the end in constant time.

that searches from a set of vertices instead of a single one or to use an explicitly parallel search strategy. These improvements lead to a less declarative look-and-feel, which is why we use the above version.

If there is no augmenting path in the graph then this function does not return any value. We use negation as failure with set functions to obtain a function that repeats the search for an augmenting path until there is no such path left.

```
maximumMatching :: Graph → Graph
maximumMatching g = go (emptyGraph (graphSize g), g) where
  go (m, notM) | isEmpty ps = m
               | otherwise  = go (augmentBy (chooseValue ps) m notM)
               where ps = set2 augmentingPath m notM
```

The function *graphSize* returns the number of vertices in the graph, *emptyGraph* creates an empty graph of a given size, *isEmpty* :: *Values a* → *Bool* checks whether the list of values is empty or not and *chooseValue* :: *Values a* → *a* nondeterministically chooses a value from the list of all values. Additionally, we can parametrise the *maximumMatching* function by a search strategy that is passed to *set2With* which is a version of *set2* that is parametrised by a search strategy.

We test the above function in the interactive mode of KiCS2 on the example graph G_2 from Figure 2. The function *graphEdges* :: *Graph* → [*Edge*] computes the edges of a graph and we use it for an uncluttered output.

```
kics2> graphEdges (maximumMatching graph2)
[(0,1),(1,0),(2,5),(3,6),(4,7),(5,2),(6,3),(7,4)]
More values? [Y(es)/n(o)/a(ll)] Y
<< four more times the same result >>
More values? [Y(es)/n(o)/a(ll)] Y
[(0,1),(1,0),(2,5),(3,6),(5,2),(6,3),(7,8),(8,7)]
More values? [Y(es)/n(o)/a(ll)] n
```

Observe that the first results are the first maximum matching from Figure 3.

As we have mentioned before, one typically distinguishes the cases of bipartite and non-bipartite graphs, because the search for an augmenting path is simpler in the former case. The above implementation does not rely on the graph being bipartite and will in fact work for non-bipartite graphs too, even those where usual algorithms will fail. The essence of this failure is known to be that every search is guided by some vertex ordering and in the non-bipartite case one can always create examples where a vertex is marked as visited prematurely, thus excluding this vertex from possible further searches. This problem cannot occur in the bipartite case – if there is an augmenting path, it can be found with every vertex ordering. The above implementation however, uses all possible vertex orderings and thus always finds a path if it exists. Clearly, this comes at the price of not being efficient (polynomial), but it is still interesting to observe that the function itself still yields the correct results, but only its complexity changes.

Another interesting observation is that such an implementation is very well suited for presentation, particularly in teaching. We have stated before that maximum matchings are not necessarily unique and the above function can be

used interactively in KiCS2 to find different maximum matchings. Similarly, for any given matching there might be several augmenting paths and one can again check that the choice of a particular path does not matter for the maximality of the result. In graphs with unique maximum matchings the confluence of the algorithm is observable by considering all solutions and removing duplicates. In the graph from Figure 1 there is exactly one maximum matching, but it can be found with the above function in 184 ways[5]. This demonstration of confluence in again interesting in teaching to indicate that even a possibly non-deterministic algorithm can yield a deterministic result.

Finally, we point out that there are more efficient algorithms that compute maximum matchings, namely the Hopcroft-Karp algorithm [13] and the Mucha-Sankowski [15] algorithm. The latter is based upon Gaussian elimination and not directly related to a search for paths. The former algorithm, however, computes in every augmentation step not just a single augmenting path, but a set of shortest, pairwise disjoint augmenting paths that is maximal with respect to inclusion. A function that computes such a set can be implemented as a combination of two searches (first a breadth-first search, followed by a modified depth-first search, cf. [7]). Interestingly, the algorithm still contains a very similar non-deterministic component as above, since there can be several sets that are all maximal and again, the correctness of the output (and sometimes even the output itself) does not depend on a particular choice of such a set. The actual implementation is not particularly difficult, but it *explicitly* relies on the fact that the BFS returns a graph that is the union of all shortest paths from the source set to the target set, rather than the property that some path has been found using BFS *implicitly*.

5 Maximal Flows in Networks

A problem that is conceptually related to maximum matchings is that of a maximal flow through a network. A network is a quadruple $N = ((V, E), s, t, c)$ that consists of an asymmetric graph[6] (V, E), two designated vertices $s, t \in V$ (where s is called *source* and t is called *sink*) such that $s \neq t$ and a capacity function[7] $c : E \to \mathbb{N}$. To avoid unnecessary brackets whenever X is a set and $h : E \to X$, we write $h(x, y)$ instead of $h((x, y))$. For any function $g : E \to \mathbb{N}$ let

$$\partial g : V \to \mathbb{Z}, \quad v \mapsto \left(\sum_{w \in N_{\rightarrow}(v)} g(v, w) \right) - \left(\sum_{w \in N_{\leftarrow}(v)} g(w, v) \right) .$$

The function ∂g measures for every vertex the difference between the amount of all outgoing values and the incoming values. A *flow* is a function $f : E \to \mathbb{N}$ that

[5] Using *liftIO length* $\circ$ *values2list* $\circ$ *set1 maximumMatching* instead of just the *maximumMatching* function yields the number of successfully found matchings.

[6] That is that for all $v, w \in V$ such that $(v, w) \in E$, we have $(w, v) \notin E$.

[7] Typically, one chooses $c : E \to \mathbb{Q}_{\geq 0}$, but since only finite graphs are considered, it is possible to multiply c by the least common multiple of its image values and, if necessary, to divide them later.

satisfies $f \leq c$ (pointwise) and for all $v \in V \setminus \{ s, t \}$ we have $\partial f(v) = 0$. This is known as the Kirchhoff's law "what goes is, must come out". The *value of a flow* f is defined as $|f| := \partial f(s)$ and a maximal flow is a flow that has a maximal flow value. In typical applications there are no edges leading into the source, which then allows the intuition that the flow value is the amount of goods that is sent through the network from the source.

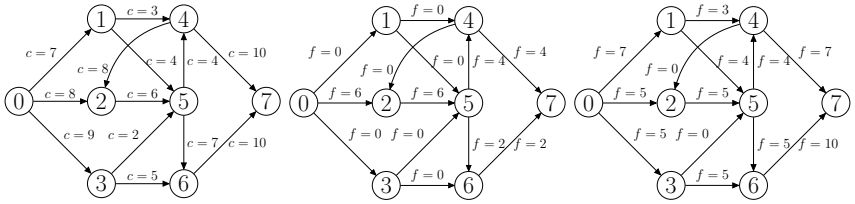

Fig. 4. Examples of a network and flows.

Flow problems are related (among many others) to routing problems, where one wishes to send a certain amount of goods through different distribution lines that have a limited capacity only (e.g. traffic or electrical current). The Kirchhoff law then simply states that there is no loss of goods along the way. There has been extensive research on finding maximal flows (cf. [11, 10, 7] for overviews and results) and efficient algorithms are known. We consider the original algorithm and variations thereof. The original algorithm for finding maximal flows is due to a theorem by Ford and Fulkerson, which is based upon [14].

Theorem 2 (Characterisation of maximal flows, Ford & Fulkerson). *Let $N = ((V, E), s, t, c)$ be a network and $f : E \to \mathbb{N}$ a flow. Let*

$$c_f : E \cup E^{-1} \to \mathbb{N}, \quad (v, w) \mapsto \begin{cases} c(v, w) - f(v, w) & : (v, w) \in E \\ f(w, v) & : \textit{otherwise} \end{cases}$$

and $E_f := \{ e \in E \cup E^{-1} \mid c_f(e) > 0 \}$. We call c_f the residual capacity w.r.t. *f. Then the following hold:*

(1) If there is no path from s to t in (V, E_f), then f is a maximal flow.
(2) If p is a path from s to t in (V, E_f), let $\varepsilon := \min \{ c_f(e) \mid e \in E(p) \}$ and

$$f_p : E \to \mathbb{N}, \quad (v, w) \mapsto \begin{cases} f(v, w) + \varepsilon & : (v, w) \in E(p) \cap E \\ f(v, w) - \varepsilon & : (w, v) \in E(p) \cap E \\ f(v, w) & : \textit{otherwise.} \end{cases}$$

Then f_p is a flow and $|f_p| = |f| + \varepsilon$ (p is called a flow-augmenting path*).*

This theorem is very similar to the Berge theorem from the previous section. In fact, it is known that the problem of finding maximum matchings in bipartite graphs can be solved using a particular instance of the flow problem. However, this technique works only for bipartite graphs in which an explicit bipartition is known, while the presented strategy does not require an explicit bipartition.

The theorem of Ford and Fulkerson provides an algorithm for finding maximum flows, which checks for the existence of a path and improves the flow in the positive case. When searching for any path, the algorithm is known as the Ford-Fulkerson algorithm, which is not necessarily polynomial in the graph size. When searching for shortest paths, this algorithm is known as the Edmonds-Karp algorithm [8] and has a complexity that is polynomial in graph size. Assuming a deterministic choice of the first element of a list of non-deterministically found augmenting paths, this fact can be reflected in Curry by specifying an explicit strategy for the path search[8]. An additional variation of the algorithm is finding a set of paths from s to t that are disjoint up to s and t, such that the set is maximal with respect to inclusion and use all paths from this set for an improvement. This is a version of the Dinitz [7] algorithm and is more efficient than the Edmonds-Karp algorithm, since it is possible to implement the search for such a set in a fashion that is just as complex as finding a single path.

For an implementation we consider capacities and flows to be functions from $V \times V$ to $\mathbb{N}$, which yield zeroes on $(V \times V) \setminus E$. For any $g, h : V \times V \to \mathbb{Z}$ set

$$\mathrm{swap}(g) : V \times V \to \mathbb{Z}, \quad (x, y) \mapsto g(y, x)$$

$$g \sqcap h : V \times V \to \mathbb{Z}, \quad e \mapsto \begin{cases} h(e) & : g(e) \neq 0 \\ 0 & : \text{otherwise.} \end{cases}$$

Then "swap" is an uncurried version of the function *flip* and $\sqcap$ is the left-forgetful intersection of functions. Let $\oplus$, $\ominus$ be the pointwise addition and subtraction of functions respectively and $\bullet$ the multiplication of a function with a constant.

From now on we assume that c is a capacity and f is a flow. The residual capacity c_f as in the Ford-Fulkerson theorem can be computed as

$$c_f := c \ominus f \oplus \mathrm{swap}(f) \ .$$

Indeed, for every $v, w \in V$ we find that due to the asymmetry at most one of the two values $f(v, w)$ and $f(w, v)$ can be non-zero, which yields

$$\begin{aligned} (c \ominus f \oplus \mathrm{swap}(f))\,(v, w) &= c(v, w) - f(v, w) + f(w, v) \\ &= \begin{cases} c(v, w) - f(v, w) & : (v, w) \in E \\ f(w, v) & : (v, w) \in E^{-1} \\ 0 & : \text{otherwise} \end{cases} = c_f(v, w) \ , \end{aligned}$$

[8] However, we use a non-deterministic choice of an augmenting path to be able to observe the different choices.

where the last step is only true up to the extension of c_f to $V \times V$. Now let p be a path from s to t in (V, E_f) and let $\varepsilon := \min\{\, c_f(e) \mid e \in E(p) \,\}$. Then set

$$\sigma_p : V \times V \to \mathbb{Z}, \quad e \mapsto \begin{cases} 1 & : e \in E(p) \\ 0 & : \text{otherwise,} \end{cases}$$

i.e. σ_p is the characteristic function of $E(p)$ in $V \times V$, and

$$u_p := \varepsilon \bullet (c \sqcap (\sigma_p \ominus \mathrm{swap}(\sigma_p)) \ .$$

The value u_p is a "point-free" version of the flow update indicated by the Ford-Fulkerson theorem: the term $\sigma_p \ominus \mathrm{swap}(\sigma_p)$ produces a function that yields 1 along the edges on p and -1 along all the reversed edges along p. The intersection produces a function that is 1 along the edges along p which are contained in E and -1 on those that are contained in E^{-1}. One easily verifies that $f_p = f \oplus u_p$. With this we can compute as follows, where all of the arithmetic rules below follow immediately from their pointwise counterparts.

$$\begin{aligned} c_{f_p} &= c \ominus f_p \oplus \mathrm{swap}(f_p) = c \ominus (f \oplus u_p) \oplus \mathrm{swap}(f \oplus u_p) \\ &= c \ominus f \ominus u_p \oplus \mathrm{swap}(f) \oplus \mathrm{swap}(u_p) = (c \ominus f \oplus \mathrm{swap}(f)) \ominus u_p \oplus \mathrm{swap}(u_p) \\ &= c_f \ominus u_p \oplus \mathrm{swap}(u_p) \ . \end{aligned}$$

Thus we can update the flow and the residual capacity using only the changes provided by the path, which reduces the number of necessary computations. To implement paths with values along traversed edges, we use the data type

```
data Path a = Final Vertex | From Vertex a (Path a)
```

and assume a function $\mathit{toEdges} :: \mathit{Path}\ a \to [(\mathit{Edge}, a)]$ to be at hand that collects all edges along the path with their corresponding values. We then model capacities and flows using finite maps FM with $(\mathit{Vertex}, \mathit{Vertex})$ keys and Int values[9].

```
type EdgeMap = FM (Vertex, Vertex) Int
```

The functions $(\oplus), (\ominus), (\sqcap) :: \mathit{EdgeMap} \to \mathit{EdgeMap} \to \mathit{EdgeMap}$, $(\bullet) :: \mathit{Int} \to \mathit{EdgeMap} \to \mathit{EdgeMap}$ and $\mathit{swap} :: \mathit{EdgeMap} \to \mathit{EdgeMap}$ are rather simple to define using standard operations on *FiniteMaps* (e.g. $(\oplus) = \mathit{plusFM\_C}\ (+)$). We can then implement the flow augmentation as follows, where the first argument is an augmenting path, the second one denotes the original capacity, the third one is the current residual capacity and the fourth one is the current flow.

```
augmentBy :: Path Int → EdgeMap → EdgeMap → EdgeMap → (EdgeMap, EdgeMap)
augmentBy p c c_f f = ((c_f ⊖ u_p) ⊕ swap u_p, f ⊕ u_p) where
```

[9] We could define a data type for natural numbers as well, but then manual conversion between naturals and integers requires some additional overhead; since this implementation is for demonstration, only, we assume the correct usage.

```
u_p   = eps • (c ⊓ (σ_p ⊖ swap σ_p))          -- the update
eps   = minlist (map snd edges)               -- the minimum along the path
σ_p   = fromList (map (λ(e, _) → (e, 1)) edges)  -- the characteristic function
edges = toEdges p
```

Note that the actual result is an exact copy of the computations from above. The maximisation function can then be realised in a fashion very similar to the one we used for matchings, but explicitly parametrised over a search strategy. This implementation adds an additional non-deterministic component, namely the choice of the actual augmenting path. The strategy for this choice is given by the top-level search strategy.

```
data Network = Network Graph Vertex Vertex EdgeMap

maximalFlowWith :: Strategy (Path Int) → Network → EdgeMap
maximalFlowWith str (Network _ s t c) = go (c, empty) where
  go (c_f, f) | isEmpty ps = f
              | otherwise = go (augmentBy (chooseValue ps) c c_f f)
              where ps = set1With str findAugmenting c_f
  findAugmenting = augmenting s t
```

All that remains is the *augmenting* function. In essence, it is another path search, but this time we use the capacity map to check for existing edges, because edges in the residual graph exist iff their capacity is positive.

```
augmenting :: Vertex → Vertex → EdgeMap → Path Int
augmenting s t capacity = go (emptyFM (<)) s where
  go vis from | from ≡ t                          = Final from
              | cfi > 0 ∧ ¬ (from `inSet` vis) = From from cfi (go (insert from vis) i)
              where i free
                    cfi = capacity ! (from, i)

(!) :: FiniteMap a Int → a → Int
m ! key = lookupFMWithDefault m 0 key
```

The non-determinism is again enclosed in the path search. Just as was the case with matchings, maximal flows are usually not unique and the presented implementation can be used to find all possibilities. Still, every maximal flow has the same flow value and this fact can be observed by defining the ∂ function.

Again, we test our implementation with the example network from Figure 4 and wrap the call in the function *showEdgeMap* :: *EdgeMap* → *String* that pretty-prints key-value pairs as *key* → *value*.

```
kics2> showEdgeMap (maximalFlowWith bfsStrategy) network1
(2,5) → 5, (0,2) → 5, (0,1) → 7, (1,4) → 3, (0,3) → 5, (1,5) → 4, (5,6) → 5,
(4,7) → 7, (3,6) → 5, (5,4) → 4, (6,7) → 10
More values? [Y(es)/n(o)/a(ll)] y
(2,5) → 3, (0,2) → 3, (0,1) → 7, (1,4) → 3, (0,3) → 7, (1,5) → 4, (5,6) → 5,
(4,7) → 7, (3,6) → 5, (3,5) → 2, (5,4) → 4, (6,7) → 10
More values? [Y(es)/n(o)/a(ll)] n
```

The first flow is exactly the maximal flow from Figure 4 and the second one demonstrates that maximal flows can variate in their edges as well as the flow values along the edges. Clearly, both flows have a flow value of 17.

6 Discussion

We have demonstrated how to apply non-deterministic path computations to compute the solution to some selected graph problems. The presented functions are not the most efficient ones by design, but intended as prototypes for demonstration. This prototypical approach has the additional advantage of being simple and declarative. Clearly, several parts of our implementations can be improved or described in a more declarative or more efficient fashion, but our focus is on the non-deterministic path computations, which are the essence of all the described algorithms.

The overall strategy of all computations in this paper can be considered as the computation of the preimages of a given function which needs to be maximised. For instance in case of flows this function is $|\cdot| : F \to \mathbb{N},\ f \mapsto |f|$, where F is the set of all flows in a given network. Similarly, every improvement step is a preimage computation for the function $\mathit{improve} : P \to F,\ p \mapsto f_p$ where P is the set of all flow-augmenting paths with respect to a given flow f. It is interesting to note that every preimage choice is a branching point in the overall computation and that every new choice can allow different branches that will still lead to the same final result. In the case of maximum matchings two different maximum matchings are distinct only in one or more edges, for flows we can observe significantly more variation, since not only the edges that have non-zero flow can be different, but even different non-zero flow values for the same edge are possible.

Being able to observe such differences is interesting in its own right, but can be of particular interest in teaching. In case of the maximum matching function there is no difference between the search strategies. The maximal flow function on the other hand behaves differently, as we have stated above, and the choice of a depth-first strategy combined with an "unlucky" vertex ordering yields a non-polynomial complexity, while the very same concrete program with a breadth-first search strategy is in a completely different complexity class.

In our applications, results may be computed repeatedly through different branches. Since set functions are implemented using lists, removing duplicates is possible but costly, since a straight-forward graph comparison takes $\mathcal{O}(|E|)$ operations. For matchings this is slightly better, since every matching has only $\mathcal{O}(|V|)$ edges, which makes the naïve duplicate removal less inefficient. Still, with focus on experimentation and teaching, an inefficient duplicate removal is still rather simple, because the Curry function $\mathit{nub} :: [\alpha] \to [\alpha]$ removes duplicates from a given list (of ground terms).

The presented implementation and the ideas behind it are not exclusive to Curry, but passing search strategies to a set function is already a built-in feature of Curry and particularly KiCS2. However, KiCS2 translates Curry programs to Haskell and using non-deteminism monads (see [4]) and replacing logic variables by overlapping rules (as in [1]) one can obtain a purely functional implementation. Such an implementation should be portable to every other language that supports higher-order functions. It should not be too difficult to translate the above functions into a relational setting, e.g. in Prolog, after removing the en-

capsulated non-determinism. Additionally, negations need to be handled with care in general, but in our case we used negations only to check for "being not contained in the visited vertices", which can be inlined and implemented by hand without explicit negation. However, Prolog uses a built-in DFS, which disallows the parametrisation over the search strategy. It is difficult to estimate how declarative and structurally complex the resulting program will be in another language. While we omitted some auxiliary functions, we still consider our code to be rather simple and straight-forward.

Acknowledgements: I thank Rudolf Berghammer for encouraging this work, Frank Huch for sparking its idea and the highly appreciated feedback of the reviewers.

References

1. S. Antoy and M. Hanus. Overlapping Rules and Logic Variables in Functional Logic Programs. In *ICLP*, pages 87–101, 2006.
2. S. Antoy and M. Hanus. Set Functions for Functional Logic Programming. In *Proc. of the 11th International ACM SIGPLAN Conference on Principle and Practice of Declarative Programming (PPDP'09)*, pages 73–82. ACM Press, 2009.
3. Claude Berge. Two Theorems in Graph Theory. In *PNAS*, volume 43 (9), pages 842–844. National Academy of Sciences, 1957.
4. B. Braßel, S. Fischer, M. Hanus, and F. Reck. Transforming Functional Logic Programs into Monadic Functional Programs. In Julio Mariño, editor, *WFLP*, volume 6559 of *LNCS*, pages 30–47. Springer, 2010.
5. B. Braßel, M. Hanus, B. Peemöller, and F. Reck. KiCS2: A New Compiler from Curry to Haskell. In *Proc. of the 20th Int. Workshop on Functional and (Constraint) Logic Prog. (WFLP 2011)*, pages 1–18. Springer LNCS 6816, 2011.
6. Nikita Danilenko. Using Relations to Develop a Haskell Program for Computing Maximum Bipartite Matchings. In Wolfram Kahl and Timothy G. Griffin, editors, *RAMICS*, volume 7560 of *LNCS*, pages 130–145. Springer, 2012.
7. Yefim Dinitz. Dinitz' Algorithm: The Original Version and Even's Version. In Oded Goldreich, Arnold L. Rosenberg, and Alan L. Selman, editors, *Essays in Memory of Shimon Even*, volume 3895 of *LNCS*, pages 218–240. Springer, 2006.
8. J. Edmonds and R. M. Karp. Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems. *J. ACM*, 19(2):248–264, 1972.
9. Jack Edmonds. Paths, trees and flowers. *Canadian J. Math.*, 17:449–467, 1965.
10. A. V. Goldberg and S. Rao. Beyond the Flow Decomposition Barrier. *J. ACM*, 45(5):783–797, 1998.
11. A. V. Goldberg and R. E. Tarjan. A New Approach to the Maximum-Flow Problem. *J. ACM*, 35(4):921–940, 1988.
12. Michael Hanus (ed.). Curry: An Integrated Functional Logic Language (Vers. 0.8.3). Available at `http://www.curry-language.org`, 2012.
13. J. E. Hopcroft and R. M. Karp. An $n^{5/2}$ Algorithm for Maximum Matchings in Bipartite Graphs. *SIAM J. Comput.*, 2(4):225–231, 1973.
14. L. R. Ford Jr. and D. R. Fulkerson. Maximal Flow through a Network. *Canadian J. Math.*, 8:399–404, 1956.
15. M. Mucha and P. Sankowski. Maximum Matchings via Gaussian Elimination. In *FOCS*, pages 248–255. IEEE Computer Society, 2004.