

# miniKanren with fair search strategies

KUANG-CHEN LU, Indiana University

WEIXI MA, Indiana University

DANIEL P. FRIEDMAN, Indiana University

The syntax of a programming language should reflect its semantics. When writing a `conde` expression in miniKanren, a programmer would expect all clauses share the same chance of being explored, as these clauses are written in parallel. The existing search strategy, interleaving `DFSi`, however, prioritize its clauses by the order how they are written down. Similarly, when a `conde` is followed by another goal conjunctively, a programmer would expect answers in parallel share the same chance of being explored. Again, the answers by `DFSi` is different from the expectation. We have devised three new search strategies that have different level of fairness in `disj` and `conj`.

## ACM Reference Format:

Kuang-Chen Lu, Weixi Ma, and Daniel P. Friedman. 2019. miniKanren with fair search strategies. 1, 1 (May 2019), 12 pages. <https://doi.org/10.1145/nnnnnnn.nnnnnnn>

## 1 INTRODUCTION

miniKanren is a family of relational programming languages. Friedman et al. [2] introduce miniKanren and its implementation in the The Reasoned Schemer, 2nd Ed(TR2). miniKanren programs, especially relational interpreters, have been proven to be useful in solving many problems by Byrd et al. [1].

A subtlety arises when a `conde` contains many clauses: not every clause has the equal chance to contribute to the result. As an example, consider the following relation `repeato` and its invocation.

```
(defrel (repeato x out)
  (conde
    [(== '(,x) out)]
    [(fresh (res)
      (== '(,x . ,res) out)
      (repeato x res))]))
> (run 4 q
  (repeato '* q))
'((*) (* *) (* * *) (* * * *))
```

Next, consider the following disjunction of invoking `repeato` with four different letters.

---

Authors' addresses: Kuang-Chen LuIndiana University; Weixi MaIndiana University; Daniel P. FriedmanIndiana University.

---

Unpublished working draft. Not for distribution.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

© 2019 Association for Computing Machinery.  
XXXX-XXXX/2019/5-ART \$15.00  
<https://doi.org/10.1145/nnnnnnn.nnnnnnn>

```

48 > (run 12 q
49   (conde
50     [(repeato 'a q)]
51     [(repeato 'b q)]
52     [(repeato 'c q)]
53     [(repeato 'd q)]))
54

```

cond<sup>e</sup> intuitively relates its clauses with logical or. And thus an unsuspecting beginner would expect each letter to contribute equally to the result, as following.

```

57 '((a) (b) (c) (d)
58   (a a) (b b) (c c) (d d)
59   (a a a) (b b b) (c c c) (d d d))
60

```

The cond<sup>e</sup> in TR2, however, generates a less expected result.

```

61 '((a) (a a) (b) (a a a)
62   (a a a a) (b b)
63   (a a a a a) (c)
64   (a a a a a a) (b b b)
65   (a a a a a a a) (d))
66

```

The miniKanren in TR2 implements interleaving DFS(DFS<sub>i</sub>), the cause of this unexpected result. With this search strategy, each clause takes a half of its received computational resources and pass the other half to its following clauses, except for the last clause that takes all resources it receives. In the example above, the a clause takes half of all recourses. And the b clause takes a quarter. Thus c and d barely contribute to the result.

DFS<sub>i</sub> is sometimes powerful for an expert. By carefully organizing the order of cond<sup>e</sup> clauses, a miniKanren program can explore more “interesting” clauses than those uninteresting ones, and thus use computational resources efficiently. A little miniKanrenner, however, may beg to differ—understanding implementation details and fiddling with clauses order is not the first priority of a beginner.

There is another reason that miniKanren could use more search strategies than just DFS<sub>i</sub>. In many applications, there does not exist one order that serves for all purposes. For example, a relational dependent type checker contains clauses for constructors that build data and clauses for eliminators that use data. When this type checker is used to generate simple and shallow programs, the clauses of constructors should be put in the front of cond<sup>e</sup>. When performing proof search for complicated programs, the clauses of eliminator should take the focus. With DFS<sub>i</sub>, these two uses can not be efficient at the same time. In fact, to make one use efficient, the other one must be drastically slow.

The specification that every clause in the same cond<sup>e</sup> is given equal “search priority” is called fair disj. And search strategies with almost-fair disj give every clauses in the same cond<sup>e</sup> similar priority. Fair conj, a related concept, is more complicated. We defer it to the next section.

To summarize our contribution, we

- propose and implement balanced interleaving depth-first search (DFS<sub>bi</sub>), a new search strategy with almost-fair disj.
- propose and implement fair depth-first search (DFS<sub>f</sub>), a new search strategy with fair disj.
- implement in a new way breath-first search (BFS), a search strategy with fair disj and fair conj. (our code runs faster in all benchmarks and is simpler). And we prove formally that our BFS implementation is equivalent to the one by Seres et al. [5].

## 2 SEARCH STRATEGIES AND FAIRNESS

In this section, we define fair disj, almost-fair disj and fair conj. Before going further into fairness, we would like to give a short review about state, search space, and goal, because fairness is defined in terms of them. A *state* is a collection of constraints. Every answer given by miniKanren corresponds to a state. A *search space* is a collection of states. And a *goal* is a function from a state to a search space.

Now we elaborate fairness by running more queries about repeato.

### 2.1 fair disj

Given the following program, it is natural to expect lists of each letter to constitute 1/4 in the answer. DFS<sub>i</sub>, the current search strategy, however, results in many more lists of a than lists of other letters. And some letters (e.g. c and d) are rarely seen. The situation would be exacerbated if cond<sup>e</sup> contains more clauses.

```
;; iDFS
> (run 12 q
  (conde
    ((repeato 'a q))
    ((repeato 'b q))
    ((repeato 'c q))
    ((repeato 'd q))))
'((a) (a a) (b) (a a a)
  (a a a a) (b b)
  (a a a a a) (c)
  (a a a a a a) (b b b)
  (a a a a a a a) (d))
```

Under the hood, the cond<sup>e</sup> here is allocating computational effort to four trivially different search space. The unfair disj in DFS<sub>i</sub> allocates much more effort to the first search space. On the contrary, fair disj would allocate effort evenly to each search space.

```
;; DFSf (fair disj)
> (run 12 q
  (conde
    ((repeato 'a q))
    ((repeato 'b q))
    ((repeato 'c q))
    ((repeato 'd q))))
'((a) (b) (c) (d)
  (a a) (b b) (c c) (d d)
  (a a a) (b b b) (c c c) (d d d))
```

```

142 ;; BFS (fair disj)
143 > (run 12 q
144   (conde
145     ((repeato 'a q))
146     ((repeato 'b q))
147     ((repeato 'c q))
148     ((repeato 'd q))))
149
150 '((a) (b) (c) (d)
151   (a a) (b b) (c c) (d d)
152   (a a a) (b b b) (c c c) (d d d))

```

Running the same program again with almost-fair disj (e.g.  $\text{DFS}_{bi}$ ) gives the same result. Almost-fair, however, is not completely fair, as shown by the following example.

```

155 ;; biDFS (almost-fair disj)
156 > (run 16 q
157   (conde
158     [(repeato 'a q)]
159     [(repeato 'b q)]
160     [(repeato 'c q)]
161     [(repeato 'd q)]
162     [(repeato 'e q)]))
163
164 '((b) (c) (d) (a)
165   (b b) (c c) (d d) (e)
166   (b b b) (c c c) (d d d) (a a)
167   (b b b b) (c c c c) (d d d d) (e e))

```

$\text{DFS}_{bi}$  is fair only when the number of goals is a power of 2, otherwise, some goals would be allocated twice as many resources than the others. In the above example, where the  $\text{cond}^e$  has five clauses, the clauses of b, c, and d are allocated more resources.

We end this subsection with precise definitions of all levels of disj fairness. Our definition of *fair* disj is slightly generalize from the one by Seres et al. [5]. Their definition is for binary disjunct. We generalize it to multi-arity one.

**DEFINITION 2.1 (FAIR disj).** *A disj is fair iff it allocates computational resource evenly to search spaces produced by goals in the same disjunct (e.g. clauses in the same  $\text{cond}^e$ ).*

**DEFINITION 2.2 (ALMOST-FAIR disj).** *A disj is almost-fair iff it allocates computational resource so evenly to search spaces produced by goals in the same disjunct that the maximal ratio of resources is bounded by a constant.*

**DEFINITION 2.3 (UNFAIR disj).** *A disj is unfair iff it is not even almost-fair.*

## 2.2 fair conj

In the following program, the three  $\text{cond}^e$  clauses differ in a trivial way. So we expect lists of each letter constitute 1/4 of the answer list. Search strategies with unfair conj (e.g.  $\text{DFS}_i$ ,  $\text{DFS}_f$ ), however, give us many more lists of a than lists of other letters. And some letters (e.g. lists of c) are rarely found. Although  $\text{DFS}_i$ 's disj is unfair in general, it is fair when there is no call to relational definition in sub-goals, including this case. The situation

would be worse if we add more `conde` clauses. The result with `DFSbi`, whose `conj` is also unfair, is similar, but due to its different `disj`, the position of `b` and `c` are swapped.

```
;; iDFS
> (run 12 q
   (fresh (x)
    (conde
      ((== 'a x))
      ((== 'b x))
      ((== 'c x))
      ((== 'd x)))
    (repeato x q)))
'((a) (a a) (b) (a a a)
  (a a a a) (b b)
  (a a a a a) (c)
  (a a a a a a) (b b b)
  (a a a a a a a) (d))
```

Intuitively, search strategies with fair `conj` should produce each letter of lists equally frequently. Indeed, `BFS` does so.

```
;; BFS (fair conj)
> (run 12 q
   (fresh (x)
    (conde
      [(== 'a x)]
      [(== 'b x)]
      [(== 'c x)]
      [(== 'd x)])
    (repeato x q)))
'((a) (b) (c) (d)
  (a a) (b b) (c c) (d d)
  (a a a) (b b b) (c c c) (d d d))
```

A more interesting situation is when the first conjunctive goal produces infinite many answers. Consider the following example, a naive specification of fair `conj` might require search strategies to produce all sorts of singleton lists, but no longer ones, which makes the strategies *incomplete*.

<sup>c1</sup>

<sup>c1</sup> *MVC*: incomplete w.r.t. what? where's the definition of incomplete?

```

236 ;; naively fair conj
237 > (run 6 q
238   (fresh (xs)
239     (conde
240       [(repeato 'a xs)]
241       [(repeato 'b xs)])
242     (repeato xs q)))
243 '(((a)) ((b)))
244   ((a a)) ((b b)))
245   ((a a a)) ((b b b)))
246

```

Our solution requires a search strategy with *fair conj* to package answers in bags, where each bag contains finite answers, and to allocate resources evenly among search spaces derived from answers in the same bag. The way to package depends on search strategy. And how to allocate resources among search space related to different bags is unspecified. Our definition of *fair conj* is orthogonal with completeness. For example, a naively fair strategy is fair but not complete, while BFS is fair and complete.<sup>c1</sup>

BFS packages answers by their costs. The *cost* of a answer is its depth in the search tree (i.e. the number of calls to relational definitions required to find them) Seres et al. [5]. In the following example, every answer is a list of list of symbol. The cost of each of them is equal to the length of the inner lists plus the length of the outer list. In addition to being fair, BFS also produces answers in increasing order of cost.

```

257 ;; BFS (fair conj)
258 > (run 12 q
259   (fresh (xs)
260     (conde
261       [(repeato 'a xs)]
262       [(repeato 'b xs)])
263     (repeato xs q)))
264 '(((a)) ((b)))
265   ((a) (a)) ((b) (b))
266   ((a a)) ((b b))
267   ((a) (a) (a)) ((b) (b) (b))
268   ((a a) (a a)) ((b b) (b b))
269   ((a a a)) ((b b b)))
270

```

### 3 BALANCED INTERLEAVING DEPTH-FIRST SEARCH

Balanced interleaving DFS (DFS<sub>bi</sub>) has almost-fair *disj* and unfair *conj*. The implementation of DFS<sub>bi</sub> differs from DFS<sub>i</sub> in the *disj* macro. We list the new *disj* with its helpers in Fig. 1. The first helper function, *split*, takes a list of goals *ls* and a procedure *k*, partitions *ls* into two sub-lists of roughly equal length, and returns the application of *k* to the two sub-lists. *disj\** takes a non-empty list of goals *gs* and returns a goal. With the help of *split*, it essentially constructs a *balanced* binary tree where leaves are elements of *gs* and nodes are *disj2s*, hence the name of this search strategy. In contrast, the *disj* in DFS<sub>i</sub> constructs the binary tree with the same nodes but in the unbalanced form.

<sup>c1</sup>MVC: Also here, complete w.r.t what?

```

283 #| [Goal] x ([Goal] x [Goal] -> Goal) -> Goal |#
284 (define (split ls k)
285   (cond
286     [(null? ls) (k '() '())]
287     [else (split (cdr ls)
288                 (lambda (l1 l2)
289                   (k (cons (car ls) l2) l1))))])
290
291
292 #| [Goal] -> Goal |#
293 (define (disj* gs)
294   (cond
295     [(null? (cdr gs)) (car gs)]
296     [else
297      (split gs
298            (lambda (gs1 gs2)
299              (disj2 (disj* gs1)
300                    (disj* gs2))))])])
301
302
303 (define-syntax disj
304   (syntax-rules ()
305     [(disj) fail]
306     [(disj g ...) (disj* (list g ...))]))
307
308
309
310
311

```

Fig. 1. DFS<sub>bi</sub> implementation

#### 4 FAIR DEPTH-FIRST SEARCH

Fair DFS (DFS<sub>f</sub>) has fair disj and unfair conj. The implementation of DFS<sub>f</sub> differs from DFS<sub>i</sub>'s in disj2 (Fig. 2). disj2 is changed to call a new and fair version of append-inf. append-inf/fair immediately calls its helper, append-inf/fair<sup>^</sup>, with the first argument, s?, set to #t, which indicates that s-inf and t-inf haven't been swapped. The swapping happens at the third cond clause in the helper, where s? is updated accordingly. The first two cond clauses essentially copy the cars and stop recursion when one of the input spaces is obviously finite. The third clause, as we mentioned above, is just for swapping. When the fourth and last clause runs, we know that both s-inf and t-inf are ended with a thunk. In this case, a new thunk is constructed. The new thunk calls the driver recursively. Here changing the order of t-inf and s-inf won't hurt the fairness (though it will change the order of answers). We swapped them back so that answers are produced in a more natural order.

#### 5 BREADTH-FIRST SEARCH

BFS has both fair disj and fair conj. Our implementation is based on DFS<sub>f</sub> (not DFS<sub>i</sub>). To implement BFS based on DFS<sub>f</sub> all we have to do is renaming append-map-inf to append-map-inf/fair, then replacing its use of append-inf to append-inf/fair.

The implementation can be improved in two ways. First, as mentioned in section 2.2, BFS puts answers in bags and answers of the same cost are in the same bag. In this implementation, however, it is unclear where this information is recorded. Second, append-inf/fair is extravagant in memory usage. It makes  $O(n + m)$  new

```

330 #lang racket
331
332 #| Goal x Goal -> Goal |#
333 (define (disj2 g1 g2)
334   (lambda (s)
335     (append-inf/fair (g1 s) (g2 s))))
336
337 #| Space x Space -> Space |#
338 (define (append-inf/fair s-inf t-inf)
339   (let loop ([s? #t]
340              [s-inf s-inf]
341              [t-inf t-inf])
342     (cond
343       ((pair? s-inf)
344        (cons (car s-inf)
345              (loop s? (cdr s-inf) t-inf)))
346       ((null? s-inf) t-inf)
347       (s? (loop #f t-inf s-inf))
348       (else (lambda ()
349                (append-inf/fair (t-inf) (s-inf)))))))
350
351
352

```

Fig. 2. DFS<sub>f</sub> implementation

cons cells every time, where  $n$  and  $m$  are the “length”s of input search spaces. We address these issues in the first subsection.

Both our BFS and Seres’s BFS Seres et al. [5] produce answers in increasing order of cost. So it is interesting to see if they are equivalent. We prove so in Coq. The details are in the second subsection.

## 5.1 optimized BFS

<sup>c1</sup> <sup>c2</sup> <sup>c3</sup>

As mentioned in section 2.2, BFS puts answers in bags and answers of the same cost are in the same bag. The cost information is recorded subtly – the cars of a search space have cost 0 (i.e. they are in the same bag), and the costs of answers in thunk are computed recursively then increased by one. It is even more subtle that `append-inf/fair` and the `append-map-inf/fair` respects the cost information. We make these facts more obvious by changing the type of search space, modifying related function definitions, and introducing a few more functions.

The new type is a pair whose car is a list of answers (the bag), and whose cdr is either a `#f` or a thunk returning a search space. A falsy cdr means the search space is obviously finite.

Functions related to the pure subset are listed in Fig. 3 (the others in Fig. 4). They are compared with Seres et al.’s implementation later. The first three functions in Fig. 3 are search space constructors. `none` makes an

<sup>c1</sup>MVC: Though bag is well known, people rarely say “bagging”. How about putting information in a bag, or something better?

<sup>c2</sup>MVC: What is the bagging information?

<sup>c3</sup>LKC: It’s just cost... You’re right. I should have be more direct.



```

377 (define (none) '(() . #f))
378 (define (unit s) '((,s) . #f))
379 (define (step f) '(() . ,f))
380
381 (define (append-inf/fair s-inf t-inf)
382   (cons (append (car s-inf) (car t-inf))
383         (let ([t1 (cdr s-inf)]
384               [t2 (cdr t-inf)])
385           (cond
386             [(not t1) t2]
387             [(not t2) t1]
388             [else (lambda () (append-inf/fair (t1) (t2))))])))
389
390
391 (define (append-map-inf/fair g s-inf)
392   (foldr
393     (lambda (s t-inf)
394       (append-inf/fair (g s) t-inf))
395     (let ([f (cdr s-inf)])
396       (step (and f (lambda () (append-map-inf/fair g (f))))))
397     (car s-inf)))
398
399
400 (define (take-inf n s-inf)
401   (let loop ([n n]
402              [vs (car s-inf)])
403     (cond
404       ((and n (zero? n)) '())
405       ((pair? vs)
406        (cons (car vs)
407              (loop (and n (sub1 n)) (cdr vs))))
408       (else
409        (let ([f (cdr s-inf)])
410          (if f (take-inf n (f)) '()))))))
411
412

```

Fig. 3. new and changed functions in optimized BFS that implements pure features

empty search space; `unit` makes a space from one answer; and `step` makes a space from a thunk. The remaining functions do the same thing as before.

Luckily, the change in `append-inf/fair` also fixes the miserable space extravagance – the use of `append` helps us to reuse the first bag of `t-inf`.

Kiselyov et al. [3] has shown that a *MonadPlus* hides in implementations of logic programming system. Our BFS implementation is not an exception: `none`, `unit`, `append-map-inf`, and `append-inf` correspond to `mzero`, `unit`, `bind`, and `mplus` respectively.

```

424
425 (define (elim s-inf ks kf)
426   (let ([ss (car s-inf)]
427         [f (cdr s-inf)])
428     (cond
429       [(and (null? ss) f)
430        (step (lambda () (elim (f) ks kf)))]
431       [(null? ss) (kf)]
432       [else (ks (car ss) (cons (cdr ss) f))]))))
433
434
435 (define (ifte g1 g2 g3)
436   (lambda (s)
437     (elim (g1 s)
438           (lambda (s0 s-inf)
439             (append-map-inf/fair g2
440                                   (append-inf/fair (unit s0) s-inf)))
441           (lambda () (g3 s)))))
442
443
444 (define (once g)
445   (lambda (s)
446     (elim (g s)
447           (lambda (s0 s-inf) (unit s0))
448           (lambda () (none)))))
449

```

Fig. 4. new and changed functions in optimized BFS that implements impure features

Functions implementing impure features are in Fig. 4. The first function, `elim`, takes a space `s-inf` and two continuations `ks` and `kf`. When `s-inf` contains some answers, `ks` is called with the first answer and the rest space. Otherwise, `kf` is called with no argument. Here ‘s’ and ‘f’ means ‘succeed’ and ‘fail’ respectively. This function is an eliminator of search space, hence the name. The remaining functions do the same thing as before.

## 5.2 comparison with the BFS of Seres et al. [5]

In this section, we compare the pure subset of our optimized BFS with the BFS found in Seres et al. [5]. We focus on the pure subset because Silvija’s system is pure. Their system represents search spaces with streams of lists of answers, where each list is a bag.

To compare efficiency, we translate her Haskell code into Racket (See supplements for the translated code). The translation is direct due to the similarity in both logic programming systems and search space representations. The translated code is longer and slower. Details about difference in efficiency are in section 6.

We prove in Coq that the two BFSs are equivalent, i.e.  $(\text{run } n \text{ } g)$  produces the same result (See supplements for the formal proof).

benchmark	size	DFS <sub>i</sub>	DFS <sub>bi</sub>	DFS <sub>f</sub>	optimized BFS	Silvija's BFS
very-recursiveo	100000	579	793	2131	1438	3617
	200000	1283	1610	3602	2803	4212
	300000	2160	2836	-	6137	-
appendo	100	31	41	42	31	68
	200	224	222	221	226	218
	300	617	634	593	631	622
reverso	10	5	3	3	38	85
	20	107	98	51	4862	5844
	30	446	442	485	123288	132159
quine-1	1	71	44	69	-	-
	2	127	142	95	-	-
	3	114	114	93	-	-
quine-2	1	147	112	56	-	-
	2	161	123	101	-	-
	3	289	189	104	-	-
'(I love you)-1	99	56	15	22	74	165
	198	53	72	55	47	74
	297	72	90	44	181	365
'(I love you)-2	99	242	61	16	66	99
	198	445	110	60	42	64
	297	476	146	49	186	322

Table 1. The results of a quantitative evaluation: running times of benchmarks in milliseconds

## 6 QUANTITATIVE EVALUATION

In this section, we compare the efficiency of search strategies. A concise description is in Table 1. A hyphen means running out of memory. The first two benchmarks are taken from Friedman et al. [2]. *reverso* is from Rozplokhas and Boulytchev [4]. Next two benchmarks about quine are modified from a similar test case in Byrd et al. [1]. The modifications are made to circumvent the need for symbolic constraints (e.g.  $\neq$ , *absento*). Our version generates de Bruijnized expressions and prevent closures getting into list. The two benchmarks differ in the *cond<sup>e</sup>* clause order of their relational interpreters. The last two benchmarks are about synthesizing expressions that evaluate to '(I love you). This benchmark is also inspired by Byrd et al. [1]. Again, the sibling benchmarks differ in the *cond<sup>e</sup>* clause order of their relational interpreters. The first one has elimination rules (i.e. *application*, *car*, and *cdr*) at the end, while the other has them at the beginning. We conjecture that DFS<sub>i</sub> would perform badly in the second case because elimination rules complicate the problem when running backward. The evaluation supports our conjecture.

In general, only DFS<sub>i</sub> and DFS<sub>bi</sub> constantly perform well. DFS<sub>f</sub> is just as efficient in all benchmarks but *very-recursiveo*. Both BFS have obvious overhead in many cases. Among the three variants of DFS (they all have *unfair conj*), DFS<sub>f</sub> is most resistant to clause permutation, followed by DFS<sub>bi</sub> then DFS<sub>i</sub>. Among the two implementation of BFS, ours constantly performs as well or better. Interestingly, every strategies with *fair disj* suffers in *very-recursiveo* and DFS<sub>f</sub> performs well elsewhere. Therefore, this benchmark might be a special case. *Fair conj* imposes overhead constantly except in *appendo*. The reason might be that strategies with *fair conj* tend to keep more intermediate answers in the memory.

## 7 RELATED WORKS

Edward points out a disjunct complex would be ‘fair’ if it is a full and balanced tree Yang [6].

Silvija et al Seres et al. [5] also describe a breadth-first search strategy. We proof their BFS is equivalent to ours. But our code looks simpler and performs better in comparison with a straightforward translation of their Haskell code.

## 8 CONCLUSION

We analysis the definitions of fair disj and fair conj, then propose a new definition of fair conj. Our definition is orthogonal with completeness.

We devise three new search strategies: balanced interleaving DFS ( $\text{DFS}_{\text{bi}}$ ), fair DFS ( $\text{DFS}_{\text{f}}$ ), and BFS.  $\text{DFS}_{\text{bi}}$  has almost-fair disj and unfair conj.  $\text{DFS}_{\text{f}}$  has fair disj and unfair conj. BFS has both fair disj and fair conj.

Our quantitative evaluation shows that  $\text{DFS}_{\text{bi}}$  and  $\text{DFS}_{\text{f}}$  are competitive alternatives to  $\text{DFS}_{\text{i}}$ , the current search strategy, and that BFS is less practical.

We prove our BFS is equivalent to the BFS in Seres et al. [5]. Our code is shorter and runs faster than a direct translation of their Haskell code.

## ACKNOWLEDGMENTS

## REFERENCES

- [1] William E Byrd, Michael Ballantyne, Gregory Rosenblatt, and Matthew Might. 2017. A unified approach to solving seven programming problems (functional pearl). *Proceedings of the ACM on Programming Languages* 1, ICFP (2017).
- [2] Daniel P. Friedman, William E. Byrd, Oleg Kiselyov, and Jason Hemann. 2018. *The Reasoned Schemer, Second Edition*.
- [3] Oleg Kiselyov, Chung-chieh Shan, Daniel P Friedman, and Amr Sabry. 2005. Backtracking, interleaving, and terminating monad transformers:(functional pearl). *ACM SIGPLAN Notices* 40, 9 (2005), 192–203.
- [4] Dmitri Rozplokhas and Dmitri Boulytchev. 2018. Improving Refutational Completeness of Relational Search via Divergence Test. In *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming*. ACM, 18.
- [5] Silvija Seres, J Michael Spivey, and C. A. R. Hoare. 1999. Algebra of Logic Programming.. In *ICLP*. 184–199.
- [6] Edward Z. Yang. 2010. Adventures in Three Monads. *The Monad. Reader Issue* 15 (2010), 11.