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 `cond`^{e} 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 DFS (DFS_i), however, prioritize its clauses by the order how they are written down. Similarly, when a `cond`^{e} is followed by another goal conjunctively, a programmer would expect answers in parallel share the same chance of being explored. Again, the answers by DFS_i 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. miniKanren programs, especially relational interpreters, have been proven to be useful in solving many problems by Byrd et al. [1].

A subtlety in writing miniKanren programs with large `cond`^{e} expressions, such as relational interpreters, is that the order of `cond`^{e} clauses sometimes affect the speed considerably. This phenomenon appears when running with the miniKanren in Friedman et al. [2], one of the most well-understood implementation. This is because for all `cond`^{e} expressions with more than two clauses, the clauses are given different "search priority". Left clauses are given higher priorities. This biased treatment causes 2 problems when a `cond`^{e} expression has many clauses: the right-most clause can hardly contribute to query result; and the efficiency of the whole program might depend largely on the order of these clauses.

[c1]

Under the hood, `cond`^{e} clauses are combined with `disj`. The `disj` implementation in [2] is not *fair*.

Definition 1.1 (fair `disj`). *A `disj` is fair iff it allocates computational resource evenly among search spaces derived from disjunctive goals. Seres et al. [5]*

A closely related concept is *almost-fair* `disj`.

Definition 1.2 (almost-fair `disj`). *A `disj` is almost-fair iff it allocates computational resource so evenly among search spaces derived from disjunctive goals that the maximal ratio of resources is bounded by a constant.*

[c1] *LKC*: Should I explain "state" and "search space" here?

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

XXXX-XXXX/2019/5-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

fairness	DFS_i	DFS_{bi}	DFS_f	BFS
disj	unfair	almost-fair	fair	fair
conj	unfair	unfair	unfair	fair

Fig. 1. fairness of search strategies

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

Fig. 2. `repeato` and an example run

The `disj` in Friedman et al. [2], or more precisely, in its search strategy, interleaving DFS(DFS_i), is neither fair nor almost-fair. Fair depth-first search (DFS_f), a new search strategy in this paper, has fair `disj`. And balanced interleaving depth-first search (DFS_{bi}), another new strategy, has almost-fair `disj`. Breath-first search (BFS), a known strategy Seres et al. [5], has fair `disj` and *fair* `conj`.

Definition 1.3 (fair `conj`). *A `conj` is fair iff it allocates computational resource evenly among search spaces derived from states in the same bag, where bags are finite list of states.*

A comparison of fairness of search strategies is in Fig. 1.

To summarize our contribution, we

- propose a new concept, almost-fair `disj`.
- propose a new definition of fair `conj`.
- propose and implement balanced interleaving depth-first search (DFS_{bi}), a new search strategy with almost-fair `disj`.
- propose and implement fair depth-first search (DFS_f), 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)
- prove our BFS implementation is equivalent with the one by Seres et al. [5].

2 FAIRNESS

In this section, we elaborate fairness by running queries about `repeato`, a relational definition that relates a term `x` with a non-empty list whose elements are `x` (Fig. 2).

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_i, the current search strategy, however, results in many more lists of `a`s than lists of other letters.

And some letters, e.g. `c` and `d`, are rarely seen. The situation would be exacerbated if `cond`^{e} contains more clauses.

```
;; iDFS (unfair disj)
> (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))
```

We borrow the definition of *fair* `disj` from Seres et al. [5]: search strategies with *fair* `disj` should allocate resources evenly among disjunctive goals. Running the same program with $\mathrm{DFS_f}$ and BFS give the following result.


```
;; fDFS and BFS (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))
```

Now we are in a middle place between fair and unfair – search strategies with *almost-fair* `disj` should allocate resources so evenly among disjunctive goals that the maximal ratio of resources is bounded by a constant. Our new search strategy, $\mathrm{DFS_{bi}}$, has almost-fair `disj`. It is fair when the number of goals is a power of 2, otherwise, some goals are allocated twice as many resources than the others. In the previous example, $\mathrm{DFS_{bi}}$ gives the same result. And in the following example, where the `cond`^{e} has 5 clause, the clauses of `b`, `c`, and `d` are allocated more resources.

```
;; biDFS (almost-fair disj)
> (run 16 q
    (conde
      [(repeato 'a q)]
      [(repeato 'b q)]
      [(repeato 'c q)]
      [(repeato 'd q)]
      [(repeato 'e q)]))
'((b) (c) (d) (a)
  (b b) (c c) (d d) (e)
  (b b b) (c c c) (d d d) (a a)
  (b b b b) (c c c c) (d d d d) (e e))
```

2.2 fair conj

In the following program, the three `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. $\mathrm{DFS_i}$, $\mathrm{DFS_f}$), however, give us many more lists of `a`s than lists of other letters. And some letters (e.g. lists of `c`) are rarely found. Although $\mathrm{DFS_i}$'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 `cond`^{e} clauses. The result with $\mathrm{DFS_{bi}}$, whose `conj` is also unfair, is similar, but due to its different `disj`, the position of `b` and `c` are swapped.

```
;; iDFS and fDFS (unfair conj)
> (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*. [c1] [c2]

```
;; naively fair conj
> (run 6 q
    (fresh (xs)
      (conde
        [(repeato 'a xs)]
        [(repeato 'b xs)])
      (repeato xs q)))
'(((a)) ((b))
  ((a a)) ((b b))
  ((a a a)) ((b b b)))
```


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. [c3]

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. [c4] [c5]

[c1] *MVC*: incomplete w.r.t. what? where's the definition of incomplete?

[c2] *LKC*: I am a bit confused. I assume completeness is a well-known concept in the context of logic programming. For example, this paper doesn't cite any source when it talks about completeness.

Hemann, Jason, et al. "A small embedding of logic programming with a simple complete search." ACM SIGPLAN Notices. Vol. 52. No. 2. ACM, 2016.

[c3] *MVC*: Also here, complete w.r.t what?

[c4] *MVC*: Here inner and outer are very confusing. Can you be more specified?

[c5] *LKC*: updated

```
#| [Goal] x ([Goal] x [Goal] -> Goal) -> Goal |#
(define (split ls k)
  (cond
    [(null? ls) (k '() '())]
    [else (split (cdr ls)
            (lambda (l1 l2)
              (k (cons (car ls) l2) l1)))]))

#| [Goal] -> Goal |#
(define (disj* gs)
  (cond
    [(null? (cdr gs)) (car gs)]
    [else
     (split gs
       (lambda (gs1 gs2)
         (disj2 (disj* gs1)
                (disj* gs2))))]))

(define-syntax disj
  (syntax-rules ()
    [(disj) fail]
    [(disj g ...) (disj* (list g ...))]))
```

Fig. 3. DFS_{bi} implementation

```
;; BFS (fair conj)
> (run 12 q
    (fresh (xs)
      (conde
        [(repeato 'a xs)]
        [(repeato 'b xs)])
      (repeato xs q)))
'(((a)) ((b))
  ((a) (a)) ((b) (b))
  ((a a)) ((b b))
  ((a) (a) (a)) ((b) (b) (b))
  ((a a) (a a)) ((b b) (b b))
  ((a a a)) ((b b b)))
```

3 BALANCED INTERLEAVING DEPTH-FIRST SEARCH

Balanced interleaving DFS (DFS_{bi}) has almost-fair `disj` and unfair `conj`. The implementation of DFS_{bi} differs from DFS_{i} in the `disj` macro. We list the new `disj` with its helpers in Fig. 3. The first helper

```
#| Goal x Goal -> Goal |#
(define (disj2 g1 g2)
  (lambda (s)
    (append-inf/fair (g1 s) (g2 s))))

#| Space x Space -> Space |#
(define (append-inf/fair s-inf t-inf)
  (append-inf/fair^ #t s-inf t-inf))

#| Bool x Space x Space -> Space |#
(define (append-inf/fair^ s? s-inf t-inf)
  (cond
    ((pair? s-inf)
     (cons (car s-inf)
       (append-inf/fair^ s? (cdr s-inf) t-inf)))
    ((null? s-inf) t-inf)
    (s? (append-inf/fair^ #f t-inf s-inf))
    (else (lambda ()
            (append-inf/fair (t-inf) (s-inf))))))
```

Fig. 4. DFS_f implementation

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 `disj2`s, hence the name of this search strategy. In contrast, the `disj` in DFS_i constructs the binary tree with the same nodes but in the unbalanced form.

4 FAIR DEPTH-FIRST SEARCH

Fair DFS (DFS_f) has fair `disj` and unfair `conj`. The implementation of DFS_f differs from DFS_i's in `disj2` (Fig. 4). `disj2` is changed to call a new and fair version of `append-inf`. `append-inf/fair` immediately calls its helper, `append-inf/fair^`, 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 `car`s 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 is fair in both `disj` and `conj`. Our implementation is based on $\mathrm{DFS_f}$ (not $\mathrm{DFS_i}$). All we have to do is apply two trivial changes to `append-map-inf`. First, rename it to `append-map-inf/fair`. Second, replace 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 `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

[c1] [c2] [c3]

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. 5 (the others in Fig. 6). They are compared with Seres et al.'s implementation later. The first three functions in Fig. 5 are search space constructors. `none` makes an 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.

Functions implementing impure features are in Fig. 6. 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.

[c1] *MVC*: Though bag is well known, people rarely say "bagging". How about putting information in a bag, or something better?

[c2] *MVC*: What is the bagging information?

[c3] *LKC*: It's just cost... You're right. I should have be more direct.

```
(define (none)   `(()    . #f))
(define (unit s) `((,s) . #f))
(define (step f) `(()    . ,f))

(define (append-inf/fair s-inf t-inf)
  (cons (append (car s-inf) (car t-inf))
    (let ([t1 (cdr s-inf)]
          [t2 (cdr t-inf)])
      (cond
        [(not t1) t2]
        [(not t2) t1]
        [else (lambda () (append-inf/fair (t1) (t2)))]))))

(define (append-map-inf/fair g s-inf)
  (foldr
    (lambda (s t-inf)
      (append-inf/fair (g s) t-inf))
    (let ([f (cdr s-inf)])
      (step (and f (lambda () (append-map-inf/fair g (f))))))
    (car s-inf)))

(define (take-inf n s-inf)
  (let loop ([n n]
             [vs (car s-inf)])
    (cond
      ((and n (zero? n)) '())
      ((pair? vs)
       (cons (car vs)
         (loop (and n (sub1 n)) (cdr vs))))
      (else
       (let ([f (cdr s-inf)])
         (if f (take-inf n (f)) '()))))))
```

Fig. 5. new and changed functions in optimized BFS that implements pure features

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. `(run n g)` produces the same result (See supplements for the formal proof).

```
(define (elim s-inf ks kf)
  (let ([ss (car s-inf)]
        [f (cdr s-inf)])
    (cond
      [(and (null? ss) f)
       (step (lambda () (elim (f) ks kf)))]
      [(null? ss) (kf)]
      [else (ks (car ss) (cons (cdr ss) f))])))

(define (ifte g1 g2 g3)
  (lambda (s)
    (elim (g1 s)
      (lambda (s0 s-inf)
        (append-map-inf/fair g2
          (append-inf/fair (unit s0) s-inf)))
      (lambda () (g3 s)))))

(define (once g)
  (lambda (s)
    (elim (g s)
      (lambda (s0 s-inf) (unit s0))
      (lambda () (none)))))
```

Fig. 6. new and changed functions in optimized BFS that implements impure features

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`^{e} 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`^{e} 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_i 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_i and DFS_bi constantly perform well. DFS_f 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_f is most resistant to clause permutation, followd by DFS_bi then DFS_i. Among the two implementation of BFS, ours constantly performs as well or better. Interestingly,

benchmark	size	DFS_i	DFS_{bi}	DFS_f	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

every strategies with fair `disj` suffers in `very-recursiveo` and DFS_f 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 (DFS_{bi}), fair DFS (DFS_f), and BFS. DFS_{bi} has almost-fair `disj` and unfair `conj`. DFS_f has fair `disj` and unfair `conj`. BFS has both fair `disj` and fair `conj`.

Our quantitative evaluation shows that DFS_{bi} and DFS_f are competitive alternatives to DFS_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.