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 (iDFS), 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 states in parallel share the same chance of being explored. Again, the answers by iDFS 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 (April 2019), 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

miniKanren programs, especially relational interpreters, have been proven to be useful in solving many problems [1]. A subtlety in writing relational programs having large `cond`^{e} expressions, such as interpreters, is that the order of `cond`^{e} clauses can affect the speed considerably. When a `cond`^{e} expression is large enough, the left clauses consume almost all the resource and the right ones are hardly explored. The unfair `disj` of the current search strategy, interleaving DFS, is the cause. Under the hood, `cond`^{e} uses `conj` to create a goal for each clause, and `disj` to combine these goals to one. The current `disj` allocates half resource to its first goal, then allocates the other half to the rest similarly, except for the last clause which takes all the resource.

Being aware of `disj` fairness, we also investigate `conj` fairness.

We propose three new search strategies, balanced interleaving DFS (biDFS), fair DFS (fDFS), and BFS. They have different characteristics of fairness (Table. 1). We prove that the pure subset of our BFS and the BFS proposed by Seres et al [4] produce the same result when queried. But our code is shorter and runs faster. Because the two BFSs are equivalent, we do not distinguish them in most places.

2 FAIRNESS

We demonstrate the aspects (`disj` or `conj`) and levels (unfair, almost-fair, or fair) of fairness by running queries about `repeato`, a relational definition that relates a term `x` with a non-empty list whose elements are `x` (Fig. 1).

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

XXXX-XXXX/2019/4-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

fairness	iDFS	biDFS	fDFS	BFS
disj	unfair	almost-fair	fair	fair
conj	unfair	unfair	unfair	fair

Table 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. 1. `repeato` and an example run

2.1 fair `disj`

In the following program, the three `cond`^{e} clauses differ in a trivial way. So we expect lists of each sort constitute 1/4 of the answer list. However, iDFS, the current search strategy, gives us much more lists of `a` than other sorts of lists. And some sorts (e.g. lists of `c`) are hardly found. The situation would be worse if we add more `cond`^{e} 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 [4]: search strategies with *fair* `disj` should allocate resource evenly among disjunctive goals. Running the same program with fDFS 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))
```

We conceive a middle place between fair and unfair – search strategies with *almost-fair* `disj` should allocate resource so evenly among disjunctive goals[4] that the the maximal ratio of resource is bounded by a constant. Our new search strategy, biDFS, has almost-fair `disj`. It is fair when the number of goals is a power of 2, otherwise, some goals are allocated twice more resource than the others. In the previous example, it 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 resource.


```
;; 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 sort constitute 1/4 of the answer list. However, search strategies with unfair `conj` (e.g. iDFS, fDFS) give us much more lists of `a` than other sorts of lists. And some sorts (e.g. lists of `c`) are hardly found. Although iDFS'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 biDFS, 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 sort 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 states. 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*.

```
;; 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 is requiring a search strategy with *fair* `conj` to package states in bags, where each bag contains finite states, and to allocate resource evenly among search spaces derived from states in the same bag. The way to package depends on search strategy. And how to allocate resource among search space related to different bags is intended left unspecified.

BFS packages states by their costs. The *cost* of a state is its depth in the search tree (i.e. the number of calls to relational definitions required to find them) [4]. In the following example, the cost of an answer is equal to the length of the inner lists plus the length of the outer list. Noted that BFS produces answers in increasing order of cost.

```
;; 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 (biDFS) has almost-fair `disj` and unfair `conj`. The implementation of biDFS differs from iDFS in the `disj` macro. We list the new `disj` with its helpers in Fig. 2. 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 `disj2`, whence the name of this search strategy. In contrast, the `disj` in iDFS essentially constructs the same sort of binary tree in one of the most unbalanced forms.

4 FAIR DEPTH-FIRST SEARCH

Fair DFS (fDFS) has fair `disj` and unfair `conj`. The implementation of fDFS differs from iDFS's in `disj2` (Fig. 3). `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 early, 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.

```
#| [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. 2. `balanced-disj`

5 BREADTH-FIRST SEARCH

BFS is fair in both `disj` and `conj`. Our implementation is based on fDFS (not iDFS). 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 aspects. First, as we mentioned in section 2.2, our BFS bag states by their cost. However, in this implementation, 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 [4] 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

As we mentioned in section 2.2, our BFS bag states by their cost. The bagging information is recorded subtly – the `car`s of a search space have cost 0 (i.e. they are in the same bag), and the costs of states 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 this facts obvious by changing the type of search space, modifying related function definitions, and introducing a few more functions.

```
#| 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. 3. How fDFS differs from iDFS

The new type is a pair whose `car` is a list of state (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. 4 (the others in Fig. 5). The first three functions in Fig. 4 are search space constructor. `none` makes an empty search space. `unit` makes an space from one state. `step` makes a space from a thunk. The remaining functions do the same thing as before. Now it should be not hard to see that `append-inf/fair` and `append-map-inf` do respect cost information.

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`.

Noted that some functions here constitute a *MonadPlus*: `none`, `unit`, `append-map-inf`, and `append-inf` correspond to `mzero`, `unit`, `bind`, and `mplus` respectively.

Functions implementing impure features are in Fig. 5. The first function, `elim`, is used to implement `ifte` and `once`. It takes a space `s-inf` and two continuations `ks` and `kf`. When `s-inf` contains some states, `ks` is called with the first state 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, whence the name. The remaining functions do the same thing as before.

5.2 comparison with Silvija's BFS

In this section, we compare the pure subset of our optimized BFS with the BFS found in [4]. We focus on the pure subset because Silvija's system is pure.

To compare efficiency, we translate her Haskell code into Racket (See supplements for the translated code). The translation is fairly straightforward due to the similarity in both logic programming system and

```
(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. 4. interface functions in optimized BFS (pure)

search space representation. The translated code is longer and slower. Details about efficiency difference are in section 6.

We prove the two search strategies are equivalent in Coq. Since search space can be infinite, we should use a co-inductive data type. However, Coq is too strict in the guardedness condition to accept a direct translation of the implementations. Therefore, we prove core theorems with finite search space instead. In order to generalize the conclusion to the cases with infinite search space, we prove a few more theorems saying that whenever we query answers lower than some finite cost, we can restrict goals to truncate

```
(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. 5. interface functions in optimized BFS (impure)

search spaces at some finite depth without changing the query result. (See supplements for the formal proof)

6 QUANTITATIVE EVALUATION

;;TODO check if reverso and appendo are absolutely the same as the ones in TRS2.

In this section, we compare the efficiency of search strategies. A concise description is in Table 2. A hyphen means running out of memory. The first three benchmarks are taken from [2]. Next two benchmarks about quine are modified from a similar test case in [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 relation interpreters. The last two benchmarks are about synthesizing expressions that evaluate to `'(I love you)`. This benchmark is also inspired by [1]. Again, the sibling benchmarks differ in the `cond`^{e} clause order of their relation 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 iDFS would perform badly in the second case because elimination rules complicate the problem when running backward. The evaluation supports our conjecture.

benchmark	size	iDFS	biDFS	fDFS	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
reverseo	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 2. The results of a quantitative evaluation: running times of benchmarks in milliseconds

In general, only iDFS and biDFS constantly perform well. fDFS 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`), fDFS is most resistant to clause permutation, followd by biDFS then iDFS. Among the two implementation of BFS, ours constantly performs as well or better. Interestingly, every strategies with fair `disj` suffers in `very-recursiveo` and fDFS 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 states in the memory.

7 RELATED WORKS

Edward points out a disjunct complex would be 'fair' if it is a full and balanced tree [5].

Silvija et al [4] also describe a breadth-first search strategy. We proof their BFS is equivalent to ours. However, ours looks simpler and performs better in comparison with a straightforward translation of their Haskell code.

8 CONCLUSION

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), 8.

[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] Silvija Seres, J Michael Spivey, and CAR Hoare. 1999. Algebra of Logic Programming.. In *ICLP*. 184–199.
[5] Edward Z Yang. 2010. Adventures in Three Monads. *The Monad. Reader Issue 15* (2010), 11.