[c1]

miniKanren with fair search strategies

KUANG-CHEN LU, Indiana University
WEIXI MA, Indiana University
DANIEL P. FRIEDMAN, Indiana University

We describe fairness levels in disjunction implementation and conjunction implementation. Specifically, a disjunction implementation can be fair, almost-fair, or unfair. And a conjunction implementation can be fair or unfair. We compare the fairness level of four search strategies: interleaving depth-first search (DFSi), balanced interleaving depth-first search (DFSbi), fair depth-first search (DFSf), and breadth-first search (BFS). DFSi is one of the best-understood search strategies in miniKanren community. DFSbi and DFSf are new. And we present a new implementation of BFS. We demonstrate by a quantitative evaluation that DFSbi and DFSf are competitive alternatives to DFSi, and that BFS is less practical.

ACM Reference Format:
Kuang-Chen Lu, Weixi Ma, and Daniel P. Friedman. 2019. miniKanren with fair search strategies. 1, 1 (May 2019), 14 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 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].

[c2]

A subtlety arises when a cond^e contains many clauses: not every clause has an 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))
'((*) (* *) (* * *) (* * * *))
```

[c1] *LKC*: submission deadline: Mon 27 May 2019
[c2] *LKC*: Shall we delete the sentence about relational interpreters? It is misleading: it is emphasizing relational interpreters, but we say nothing further about them later.

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

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

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

cond^e intuitively relates its clauses with logical or. And thus an unsuspicious beginner would expect each letter to contribute equally to the result, as follows.

```
'((a) (b) (c) (d)
  (a a) (b b) (c c) (d d)
  (a a a) (b b b) (c c c) (d d d))
```

The cond^e in TR2, however, generates a less expected result.

```
'((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))
```

The miniKanren in TR2 implements interleaving DFS (DFS$_i$), the cause of this unexpected result. With this search strategy, each clause takes 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$_i$ is sometimes powerful for an expert. By carefully organizing the order of cond^e clauses, a miniKanren program can explore more "interesting" clauses than those uninteresting ones, and thus use computational resources efficiently. A little miniKanrener, 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$_i$. 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 the type checker is used to generate simple and shallow programs, the clauses of constructors should be put in the front of cond^e. When performing proof searches for complicated programs, the clauses of eliminator should take the focus. With DFS$_i$, these two uses cannot 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^e is given equal "search priority" is called fair disj. And search strategies with almost-fair disj give every clause 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$_{bi}$), 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. And we prove formally that our BFS implementation is equivalent to the one by Seres et al. [5]. Our code runs faster in all benchmarks and is simpler.

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 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 query result. $\mathrm{DFS_i}$, the current search strategy, however, results in many more lists of as 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.

```
;; DFSi (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))
```

Under the hood, the cond^e here is allocating computational resource to four trivially different search space. The unfair `disj` in $\mathrm{DFS_i}$ allocates many more resources to the first search space. On the contrary, fair `disj` would allocate resources 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))
```

```
;; 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))
```

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

```
;; 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))
```

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 `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 disjunction. We generalize it to multi-arity one.

Definition 2.1 (fair `disj`). *A* `disj` *is fair if and only if it allocates computational resource evenly to search spaces produced by goals in the same disjunction (i.e. clauses in the same* `cond`^{e}*).*

Definition 2.2 (almost-fair `disj`). *A* `disj` *is almost-fair if and only if it allocates computational resource so evenly to search spaces produced by goals in the same disjunction that the maximal ratio of resources is bounded by a constant.*

Definition 2.3 (unfair `disj`). *A* `disj` *is unfair if and only if it is not even almost-fair.*

2.2 fair conj

[c1]

Given the following program, it is natural to expect lists of each letter to constitute 1/4 in the answer. Search strategies with unfair `conj` (e.g. DFS_{i}, DFS_{bi}, DFS_{f}), however, results in many more lists of `a`s than lists of other letters. And some letters are rarely seen. The situation would be exacerbated if `cond`^{e} contains more clauses. [c2]

[c1] *LKC*: Checked grammer of text before this line

[c2] *LKC*: Should I add a footnote?
" Although DFS_{i}'s `disj` is unfair in general, it is fair when there is no call to relational definition in `cond`^{e} clauses (including this case). "

```
;; DFSi (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))
```

```
;; DFSf (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))
```

```
;; DFSbi (unfair conj)
> (run 12 q
    (fresh (x)
      (conde
        ((≡ 'a x))
        ((≡ 'b x))
        ((≡ 'c x))
        ((≡ 'd x)))
      (repeato x q)))
'((a) (a a) (c) (a a a)
  (a a a a) (c c)
  (a a a a a) (b)
  (a a a a a a) (c c c)
  (a a a a a a a) (d))
```

Under the hood, the cond^e and the call to `repeato` are connected by `conj`. The cond^e goal outputs a search space including four trivially different states. Then the next conjunctive goal, `(repeato x q)`, is applied to each of these states, producing four trivially different search spaces. In the examples above, the `conj`s are allocating more computational resources to the search space of `a`. On the contrary, fair `conj` would allocate resources evenly to each search space. For example,

```
;; 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 conjunct 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. A search strategy is *complete* if and only if it can find out all the answers within finite time, otherwise, it is *incomplete*.

```
;; naively fair conj
> (run 6 q
    (fresh (xs)
      (cond^e
        ((repeat^o 'a xs))
        ((repeat^o 'b xs)))
      (repeat^o xs q)))
'(((a)) ((b))
  ((a a)) ((b b))
  ((a a a)) ((b b b)))
```

Our solution requires a search strategy with *fair* `conj` to organize states in bags in search spaces, where each bag contains finite states, and to allocate resources evenly among search spaces derived from states in the same bag. It is up to a search strategy designer to decide by what criteria to put states in the same bag, and how to allocate resources among search spaces related to different bags.

BFS puts states of the same cost in the same bag, and allocate resources carefully among search spaces related to different bags such that answers are produced in increasing order of cost. 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) [5]. In the following example, every answer is a list of list of symbols, where inner lists are the same. Here the cost of each answer is equal to the length of its inner lists plus the length of its outer list.

```
;; BFS (fair conj)
> (run 12 q
    (fresh (xs)
      (cond^e
        ((repeat^o 'a xs))
        ((repeat^o 'b xs)))
      (repeat^o 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)))
```

We end this subsection with precise definitions of all levels of `conj` fairness.

Definition 2.4 (fair `conj`). *A `conj` is fair if and only if it allocates computational resource evenly to search spaces produced from states in the same bag. A bag is a finite collection of state. And search strategies with fair `conj` should represent search spaces with possibly infinite collection of bags.*

Definition 2.5 (unfair `conj`). *A `conj` is unfair if and only if it is not fair.*

3 INTERLEAVING DEPTH-FIRST SEARCH

In this section, we review the implementation of interleaving depth-first search (DFS$_\text{i}$). We focus on parts that are relevant to this paper. TRS2 [2] provides a comprehensive description of the whole miniKanren implementation.

```
(define-syntax cond^e
  (syntax-rules ()
    ((cond^e (g ...) ...)
     (disj (conj g ...) ...))))
```

Fig. 1. implementation of cond^e

```
#| Goal × Goal → Goal |#
(define (disj2 g1 g2)
  (lambda (s)
    (append^∞ (g1 s) (g2 s))))

#| Space × Space → Space |#
(define (append^∞ s^∞ t^∞)
  (cond
    ((null? s^∞) t^∞)
    ((pair? s^∞)
     (cons (car s^∞)
       (append^∞ (cdr s^∞) t^∞)))
    (else (lambda ()
            (append^∞ t^∞ (s^∞))))))

(define-syntax disj
  (syntax-rules ()
    ((disj) (fail))
    ((disj g0 g ...) (disj+ g0 g ...))))

(define-syntax disj+
  (syntax-rules ()
    ((disj+ g) g)
    ((disj+ g0 g1 g ...) (disj2 g0 (disj+ g1 g ...)))))
```

Fig. 2. implementation of DFS$_i$ (Part I)

The definition of cond^e (Fig. 1) is shared by all search strategies. It relates clauses disjunctively, and goals in the same clause conjunctively.

Fig. 2 and Fig. 3 shows parts that are later compared with other search strategies. We follow some conventions to name variables: variables bound to states are named s; variables bound to goals are named g (possibly with subscript); variables bound to search spaces have names ending with $^\infty$. Fig. 2 shows the implementation of disjunction. The first function, disj2, implements binary disjunction. It applies the two disjunctive goals to the input state s and composes the two resulting search spaces with append$^\infty$. The following syntax definitions say disjunctive operator is right-associative. Fig. 3 shows the implementation of conjunction. The first function,

```
#| Goal × Goal → Goal |#
(define (conj2 g1 g2)
  (lambda (s)
    (append-map∞ g2 (g1 s))))

#| Goal × Space → Space |#
(define (append-map∞ g s∞)
  (cond
    ((null? s∞) '())
    ((pair? s∞)
     (append∞ (g (car s∞))
       (append-map∞ g (cdr s∞))))
    (else (lambda ()
            (append-map∞ g (s∞))))))

(define-syntax conj
  (syntax-rules ()
    ((conj) (fail))
    ((conj g0 g ...) (conj+ g0 g ...))))

(define-syntax conj+
  (syntax-rules ()
    ((conj+ g) g)
    ((conj+ g0 g1 g ...) (conj2 g0 (conj+ g1 g ...)))))
```

Fig. 3. implementation of DFS$_{\mathrm{i}}$ (Part II)

`conj2`, implements binary conjunction. It applies the *first* goal to the input state, then applies the second goal to states in the resulting search space. The latter process is done with a helper function. append-map$^{\infty}$ applies its input goal to states in its input search spaces and compose the resulting search spaces. It reuses append$^{\infty}$ for search space composition. The following syntax definitions say conjunctive operator is also right-associative.

4 BALANCED INTERLEAVING DEPTH-FIRST SEARCH

Balanced interleaving DFS (DFS$_{\mathrm{bi}}$) has almost-fair `disj` and unfair `conj`. The implementation of DFS$_{\mathrm{bi}}$ differs from DFS$_{\mathrm{i}}$'s in the `disj` macro. We list the new `disj` with its helper in Fig. 4. When there are one or more disjunctive goals, `disj` builds a balanced binary tree whose leaves are the goals and whose nodes are `disj2`s, hence the name of this search strategy. The new helper, `disj+`, takes two additional ‘arguments’. They accumulate goals to be put in the left and right subtrees. The first clause handles the case where there is only one goal. In this case, the tree is the goal itself. When there are more goals, we partition the list of goals into two sublists of roughly equal lengths and recur on the two sublists. Goals are moved to the accumulators in the fourth and last clause. As we are moving two goals each time, there are two base cases: (1) no goal remains; (2) one goal remains. These two base cases are handled in the second clause and the third clause respectively. In contrast, the `disj` in DFS$_{\mathrm{i}}$ constructs the binary tree in a particularly unbalanced form.

```
(define-syntax disj
  (syntax-rules ()
    ((disj) fail)
    ((disj g ...) (disj+ (g ...) () ()))))

(define-syntax disj+
  (syntax-rules ()
    [(disj+ () () g) g]
    [(disj+ (gl ...) (gr ...))
     (disj2 (disj+ () () gl ...)
            (disj+ () () gr ...))]
    [(disj+ (gl ...) (gr ...) g0)
     (disj2 (disj+ () () gl ... g0)
            (disj+ () () gr ...))]
    [(disj+ (gl ...) (gr ...) g0 g1 g ...)
     (disj+ (gl ... g0) (gr ... g1) g ...)]))
```

Fig. 4. implementation of $\mathrm{DFS_{bi}}$

```
#| Goal × Goal → Goal |#
(define (disj2 g1 g2)
  (lambda (s)
    (append∞_fair (g1 s) (g2 s))))

#| Space × Space → Space |#
(define (append∞_fair s∞ t∞)
  (let loop ((s? #t)
             (s∞ s∞)
             (t∞ t∞))
    (cond
      ((pair? s∞)
       (cons (car s∞)
         (loop s? (cdr s∞) t∞)))
      ((null? s∞) t∞)
      (s? (loop #f t∞ s∞))
      (else (lambda ()
              (loop #t (t∞) (s∞)))))))
```

Fig. 5. implementation of $\mathrm{DFS_f}$

5 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. 5). `disj2` is changed to call a new and fair version of `append`$^\infty$. `append`$^\infty_{fair}$ immediately calls its helper, `loop`, with the first argument, `s?`, set to `#t`, which indicates that `s`$^\infty$ and `t`$^\infty$ 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`$^\infty$ and `t`$^\infty$ are ended with a thunk and they have been swapped. In this case, a new thunk is constructed. Here changing the order of `t`$^\infty$ and `s`$^\infty$ won't hurt the fairness (though it will change the order of answers). We swap them back so that answers are produced in a more natural order.

6 BREADTH-FIRST SEARCH

BFS has both fair `disj` and fair `conj`. Our implementation is based on DFS_f (not DFS_i). To implement BFS based on DFS_f, we need `append-map`$^\infty_{fair}$ in addition to `append`$^\infty_{fair}$. The only difference between `append-map`$^\infty$ and `append-map`$^\infty_{fair}$ is that the latter calls `append`$^\infty_{fair}$ instead of `append`$^\infty$.

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`$^\infty_{fair}$ 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.

6.1 optimized BFS

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`$^\infty_{fair}$ and the `append-map`$^\infty_{fair}$ 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. 6 (the others in Fig. 7). They are compared with Seres et al.'s implementation in our proof. The first three functions in Fig. 6 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`$^\infty_{fair}$ also fixes the miserable space extravagance – the use of `append` helps us to reuse the first bag of `t`$^\infty$.

Kiselyov et al. [3] has shown that a *MonadPlus* hides in implementations of logic programming system. Our BFS implementation is not an exception: `append-map`$^\infty_{fair}$ is like `bind`, but takes arguments in reversed order; `none`, `unit`, and `append`$^\infty_{fair}$ correspond to `mzero`, `unit`, and `mplus` respectively.

Functions implementing impure features are in Fig. 7. The first function, `elim`, takes a space `s`$^\infty$ and two continuations `ks` and `kf`. When `s`$^\infty$ 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 like an eliminator of search space, hence the name. The remaining functions do the same thing as before.

```
#| → Space |#
(define (none)   `(()   . #f))

#| State → Space |#
(define (unit s) `((,s) . #f))

#| (→ Space) → Space |#
(define (step f) `(()   . ,f))

#| Space × Space → Space |#
(define (append∞fair s∞ t∞)
  (cons (append (car s∞) (car t∞))
    (let ((t1 (cdr s∞))
          (t2 (cdr t∞)))
      (cond
        ((not t1) t2)
        ((not t2) t1)
        (else (lambda () (append∞fair (t1) (t2))))))))

#| Goal × Space → Space |#
(define (append-map∞fair g s∞)
  (foldr
    (lambda (s t∞)
      (append∞fair (g s) t∞))
    (let ((f (cdr s∞)))
      (step (and f (lambda () (append-map∞fair g (f))))))
    (car s∞)))

#| Maybe Nat × Space → [State] |#
(define (take∞ n s∞)
  (let loop ((n n)
             (vs (car s∞)))
    (cond
      ((and n (zero? n)) '())
      ((pair? vs)
       (cons (car vs)
         (loop (and n (sub1 n)) (cdr vs))))
      (else
       (let ((f (cdr s∞)))
         (if f (take∞ n (f)) '())))))))
```

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

```
#| Space × (State × Space → Space) × (→ Space) → Space |#
(define (elim s∞ ks kf)
  (let ((ss (car s∞))
        (f (cdr s∞)))
    (cond
      ((and (null? ss) f)
       (step (lambda () (elim (f) ks kf))))
      ((null? ss) (kf))
      (else (ks (car ss) (cons (cdr ss) f))))))

#| Goal × Goal × Goal → Goal |#
(define (ifte g1 g2 g3)
  (lambda (s)
    (elim (g1 s)
      (lambda (s0 s∞)
        (append-map∞_fair g2
          (append∞_fair (unit s0) s∞)))
      (lambda () (g3 s)))))

#| Goal → Goal |#
(define (once g)
  (lambda (s)
    (elim (g s)
      (lambda (s0 s∞) (unit s0))
      (lambda () (none)))))
```

Fig. 7. new and changed functions in optimized BFS that implements impure features

6.2 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 their 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 their 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 than our BFS implementation. 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).

7 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

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

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. ≠, `absent`^{o}). 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, 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 considerable overhead constantly except in `appendo`. The reason might be that strategies with fair `conj` tend to keep more intermediate answers in the memory.

8 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 is shorter and performs better in comparison with a straightforward translation of their Haskell code.

9 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 two new search strategies (i.e. balanced interleaving DFS (DFS_{bi}) and fair DFS (DFS_f)) and devise a new implementation of BFS. These strategies have different features in fairness: $_{bi}$ 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.