List of Algorithm in Exhaustive search

Putranto

September 30, 2016

Algorithm 1: Hamming distance between two vector (`hamdist`)

Input: vector u and v (same length)
Output: hamming distance of u and v

Algorithm 2: Check for three consecutive 1's (`cekcons1`)

```
Input: a binary vector v
n ← length(v);
for i ← 0 to n − 2 do
    t ← [v_i, v_{i+1}, v_{i+2}]  if sum(t) = 3 then
        return true
    else
        return false
```

Algorithm 3: Check for three consecutive 0's (`cekcons0`)

```
Input: a binary vector v
n ← length(v);
for i ← 0 to n − 2 do
    t ← [v_i, v_{i+1}, v_{i+2}]  if sum(t) = 0 then
        return true
    else
        return false
```

Algorithm 4: Check for balancedness of a vector (`isbalance`)

Input: binary vector v

$n1 \leftarrow$ number of 1's in v;
$n0 \leftarrow$ number of 0's in v;
if $n1 = n0$ **then**
    return *true*;
else
    return *false*;

Algorithm 5: Check for similarity of two vector (`isidentic`)

Input: two vector with same length: u, v

if $u = v$ **then**
    return *true*
else
    return *false*

Algorithm 6: Check rows for 1^{st} and 3^{rd} constraint satisability (`checkforconsrow`)

Input: matrix A

for *a* ***in*** *rows of* A **do**
    if `cekcons0`$(a) =$ *true* ***or*** `cekcons1`$(a) =$ *true* ***or*** `isbalance`$(a) =$ *false* **then**
        return *false*
return *true*

Algorithm 7: Check a matrix for 1^{st} and 3^{rd} constraint satisability (`checkforcons`)

Input: a matrix A representing binary puzzle

if `checkforconsrow`$(A) =$ *false* ***or*** `checkforconsrow`$(A^{\intercal}) =$ *false* **then**
    return *false*
else
    return *true*

Algorithm 8: Forced move for columns w.r.t. 3^{rd} constraint (`partdistinctrow`)

Input: a matrix A representing binary puzzle, filled with 1, 0, and 9 (blank)

for a ***in*** *rows of* A **do**
  $n_1 \leftarrow$ number of 1's in a ;
  $n_0 \leftarrow$ number of 0's in a ;
  $n_{blank} \leftarrow$ number of blanks in a ;
  if $n_{blank} = 2$ ***and*** $n_1 = n_0$ **then**
    for b ***in*** *rows in* $A \setminus a$ **do**
      if `hamdist`$(a, b) = 2$ **then**
        $a_{i_{blank}} \leftarrow \overline{b_i}$; `/* replace blank cell` a_i `*/`

revise the algorithm. found a contradiction: x=[10100199], y=[19900199]

Algorithm 9: Iteratively forced move for matrix w.r.t. 3^{rd} constraint (`partdistinct`)

Input: a matrix A representing binary puzzle, filled with 1, 0, and 9 (blank)

$B \leftarrow$ `partdistinctrow`(A);
$B^\intercal \leftarrow$ `partdistinctrow`$(B^\intercal)$;
if $B = A$ **then**
  return B
else
  `partdistinct`(B)

Algorithm 10: Forced move a vector w.r.t. 2^{nd} constraint (`partbal`)

Input: a vector v with even length.

$n_0 \leftarrow$ number of 0's in v ;
$n_{blank} \leftarrow$ number of blanks in a ;
$m \leftarrow$ `length`$(v)/2$;
if $n_{blank} = 1$ **then**
  if $n_0 = m$ **then**
    $v_{i_{blank}} = 1$
  else
    $v_{i_{blank}} = 0$
return v

apply a new algorithm: The second, balancedness constraint can be used also to fill a blank under a certain condition. Let **x** be a column with two blanks. If the number of zeros is not equal to the number of ones, then we also can be sure what the blanks are.

Algorithm 11: Forced move a matrix w.r.t. 2^{nd} constraint (`fill2cons`)

Input: a matrix A representing binary puzzle, filled with 1, 0, and 9 (blank)

for *a* ***in*** *rows of* A **do**
    `partbal` (a)
for *a* ***in*** *columns of* A **do**
    `partbal` (a)

Algorithm 12: Forced move a matrix w.r.t. 2^{nd} constraint (`fill2consB`)

Input: a matrix A representing binary puzzle, filled with 1, 0, and 9 (blank)

$B \leftarrow$ `fill2cons`(A) ;
if $B = A$ **then**
    return B
else
    return `fill2consB`(B)

Algorithm 13: Forced move a vector w.r.t. 1^{st} constraint (`partnocons`)

Input: a vector v with length 3

$n_1 \leftarrow$ number of 1's in v ;
$n_0 \leftarrow$ number of 0's in v ;
$n_{blank} \leftarrow$ number of blanks in v ;
if $n_{blank} = 1$ ***and*** $n_1 = 2$ **then**
    $v_{i_{blank}} = 0$
else if $n_{blank} = 1$ ***and*** $n_0 = 2$ **then**
    $v_{i_{blank}} = 1$

Algorithm 14: Forced move a matrix w.r.t. 1^{st} constraint (`fill1cons`)

Input: a matrix A representing binary puzzle, filled with 1, 0, and 9 (blank)

```
nr ← number of rows in A ;
nc ← number of columns in A ;
for a in rows of A do
    for i in [0 ··· nc − 3] do
        partnocons([a[i], a[i + 1], a[i + 2]])
for a in columns of A do
    for i in [0 ··· nr − 3] do
        partnocons([a[i], a[i + 1], a[i + 2]])
```

Algorithm 15: Forced move w.r.t. 1^{st} constraint (`fill1consB`)

Input: a matrix A representing binary puzzle, filled with 1, 0, and 9 (blank)

```
B ← fill1cons(A) ;
if B = A then
    return B
else
    return fill1consB(B)
```

Algorithm 16: Forced move w.r.t. all constraint (`solvepart1`)

Input: a matrix A representing binary puzzle, filled with 1, 0, and 9 (blank)

```
B = fill1consB(A) ;
B = fill2consB(B) ;
B = partdistinct(B) ;
if B = A then
    return (B)
else
    return (solvepart1(A1))
```

Algorithm 17: Wrapper for puzzle guessing (`solvepart2`)

Input: a matrix A representing binary puzzle, *history_of_changed_cell*, *guess_counter*

Fill a blank cell in A with either 0 or 1 ;
$B \leftarrow$ `solvepart1`(A) ;
guess_counter$+ = 1$;
$history_of_changed_cell_{guess_counter} \leftarrow$list of changed and guessed cells;
return B, *history_of_changed_cell*, *guess_counter*

Algorithm 18: Wrapper for binario solver (`solvepuzzle`)

Input: a matrix A representing binary puzzle, filled with 1, 0, and 9 (blank)

```
B ← solvepart1(A) ;      /* Try to solve using forced move.  */
guess_list ← [ ] ;
n_blank ←number of blank in B;
guess_counter ← 0 ;
history_of_changed_cell ← {} ;
if B does not satisfy all the constraint then
    return A is invalid puzzle
while n_blank ≠ 0 do
    B, history_of_changed_cell, guess_counter ←
    solvepart2(B, history_of_changed_cell, guess_counter) ;
    if B does not satisfy all the constraint then
        guess_counter− = 1;
        while guess_counter in guess_list do
            remove guess_counter from guess_list ;
            guess_counter− = 1;
            if guess_counter = 0 then
                return A is invalid puzzle
        revert back to condition at guess_counter ;
        append(guess_counter) to guess_list
return B
```

Algorithm 19: Wrapper for binario solver (`solvepuzzle`)

Input: a matrix A representing binary puzzle, filled with 1, 0, and 9 (blank)

```
B ← forced(A) ;
guess_list ← [ ] ;
n_b ←number of blank in B;
guess_count ← 0 ;
cell_hist ← {} ;
if B does not satisfy all the constraint then
    return A is invalid puzzle
while n_b ≠ 0 do
    B, cell_hist, guess_count ← guess(B, cell_hist, guess_count) ;
    if B does not satisfy all the constraint then
        guess_count− = 1;
        while guess_count in guess_list do
            remove guess_count from guess_list ;
            guess_count− = 1;
            if guess_count = 0 then
                return A is invalid puzzle
        revert back to condition at guess_count ;
        append(guess_count) to guess_list
return B
```