Flow Free Solver

A solver for Flow Free puzzles using *back tracking* search for CSPs.

The problem

Boards are typically a square grid with any number of colors to connect. A *well-designed* board (an assumption made by this solver) has a unique solution, and requires the entire board to be filled in without any "zig-zagging" paths. A consequence of being an NP-complete puzzle is that, although solutions can be verified quickly, there is no known efficient algorithm to find a solution, and the time required to solve the problem increases very quickly as the board size grows. How do we leverage a computer to quickly find the solution to a given board? We can devise a metric to score potential paths towards the solution, and we investigate the paths that maximize this function first.

Getting started

From terminal

```
cd ./src
python main_dumb.py # for the dumb heuristic
python main_smart.py # for the smart heuristic
```

Graphical

```
# It is recommended to create a separate environment before you install the requirements
pip install -r requirements.txt
cd ./src
python app.py
```

Approaches

These are approaches we took to solve these puzzles, few notes need to be taken before reading. We consider the map as matrix where each element in this matrix is a *variable* and these variables are coordinates in xy plan, where y grows downwards starting from the top left corner. Assignments are stored in a dictionary-styled data structure where keys are coordinates and values are colors for each coordinate, we use uppercase letters for terminals and lowercase for pipes.

Constraints

These are the procedures we took to check the consistency of any new assignments.

`Is_good_combination`

What we mean by good combination here the state of the selected assignment don't/won't cause any problems. we can wrap them up in the following points

- Number of free neighbors >= 2 => true
- Number of similar neighbors == 1 and Number of similar neighbors == 1 => true

- Number of similar neighbors == 2 and not(is surrounding square filled) => true
- Otherwise => false

`Is_neighbors_terminals_have_valid_path`

Checks weather or not any neighboring terminal in *locked out*, in other word if our newly assigned `var : value` causes any problem.

`Is_terminal_connected`

Use the cached on demand updated terminals to check if the same `value` terminals are already connected, because if so, it doesn't make sense to assign that value to a variable again

Dumb algorithm

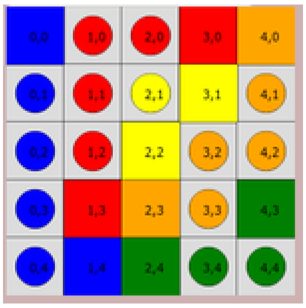
Picking a random value and random variable each time check whether or not this assignment is consultant. If it was consistent move to the next assignment in a *DFS-styled* backtracking.

Results for dump

map	time	Number of hits
5x5	7 ms	443
7x7	? ms	???
...	? ms	???

5x5:

For graphical results see figure 1.




```
map ../input/input55.txt solution time = 0.005998373031616211 sec
map ../input/input55.txt number of hits = [443]
BrrRO
bryYo
brYoo
bROoG
bBGgg
```

7x7 and higher:

TimeOut! more than (30s)

Smart Algorithm

Using a combination of helping heuristics and approaches that can be controlled via `config` dict in `src/algorithms/smart.py` including **MRV** to chose the next variable, **LCV** for choosing the value, **Degree Heuristics** as a tie breaker and **Weak locker** these heuristic are *"togglable"* due to optimization issues, check optimization labeled PRs for more information.

used combination:

- Forward checking
- MRV (minimum remaining value)
- Degree heuristic
- Least constraining value

forward checking

- find domain for variables
- if variable has zero domain
- return case failure

```
def forward_check(variables_domain):
    for coords in variables_domain:
        if len(variables_domain[coords]) == 0:
            return False
    return True
```

Results

\
5x5

results

- without forward_check

map	time	Number of hits
5x5	7 ms	443

- with forward_check

map	time	Number of hits
5x5	9 ms	28

MRV

- find domain for variables
- choose variables with smallest domain
- implementation pseudo code

```python
    smallest_domain = math.inf
    selected_coords = []
    for coord in variables_domain:
        domain_len = len(variables_domain[coord])
        if domain_len < smallest_domain:
            selected_coords = []
            smallest_domain = domain_len

        if smallest_domain == domain_len:
            selected_coords.append(coord)

    return selected_coords
```

initial results *

map	time (s)
7x7	1.87
8x8	1.73
9x9	6.87
10x10	?.??

optimization

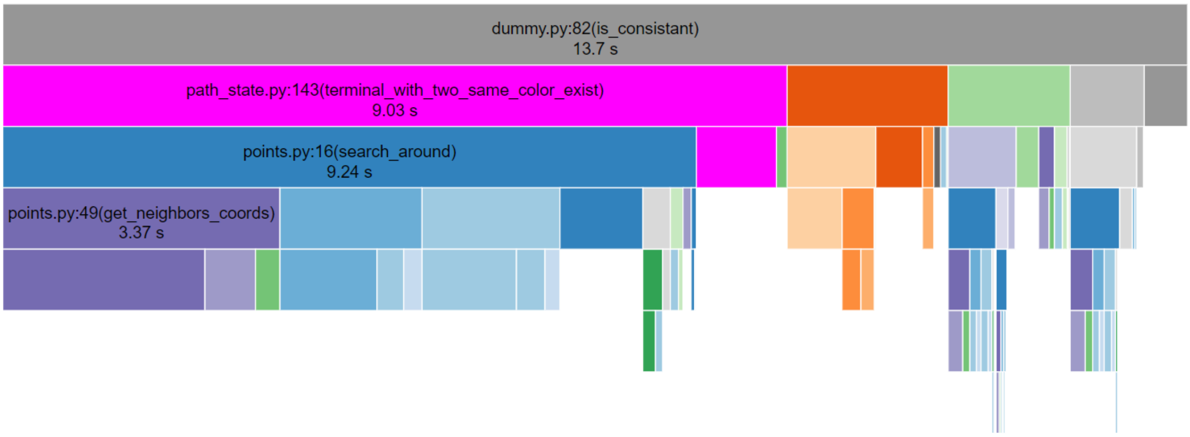
limitation

- variable domain calculation increase with map size
    - example
        - 14x14 every time calculate domain

          for (196 - terminals) variable
    - solution
        - update only constrained variables
- consistency check represent the bottleneck

improvement in constrains

profile for 991

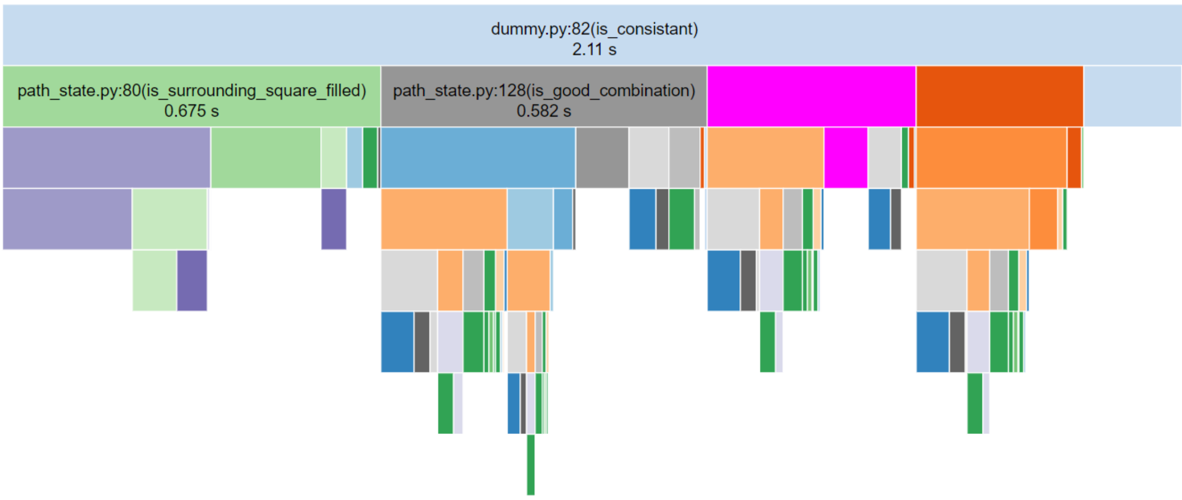


terminal constrain was checking every terminal has only on path

update

check only neighbor terminals
this improved performance significantly

profile for 991



results

map	time (s)
7x7	0.085
8x8	0.157
9x9	0.858
10x10(1)	3.300
10x10(2)	1.680
12x12	14.971
12x14	??.???

lazy surrounding squares

check surrounding squares "zigzag" only when have 2 same color neighbors. Because if we have a good combination we are in one of these two cases

- only one free neighbor and same color neighbor (no squares)
- only 2 or more free neighbors (no squares)

implementation

```
def check_for_good_combinations(coord, current_color, assignments, inp):
    empty_neighbors = search_around(coord, inp, assignments, is_empty)
    if len(empty_neighbors) >= 2:
        return True
    same_color_neighbors = get_same_color_neighbors(
        coord, current_color, assignments, inp)
    if len(same_color_neighbors) == 2:
        # we don't need is surrounding square anywhere but here
        ssf = is_surrounding_square_filled(assignments,inp,coord)
        return not ssf
    if len(empty_neighbors) == 1 and len(same_color_neighbors) == 1:
        return True
    return False
```

Results

map	time (s)
10x10(1)	2.26
10x10(2)	1.09
12x12	8.72
12x14	??.???

Cache connected terminals

We can cache connected terminals to quickly check whether or not the selected value is consistent\
Using a shared object between backtracks that gets updated only when a variable is consistent

Implementation

Inside backtrack

```
if is_consistant(initial_state, {var: value},  assignments, inp,
connected_terminals):
    before_assgen_connected_terminal = connected_terminals
    refreshed_connected_terminals = refresh_connected_terminals( {var: value},
assignments, connected_terminals, initial_state, inp)
```

Results

map	time (s)
10x10(1)	1.49
10x10(2)	0.728
12x12	5.38
12x14	??.???

---

dynamic domain-upgrade

- save variable domain
- only update constrained variables

the constrained variables

- are empty neighbor (point good combination)
- are empty neighbor for occupied neighbor
  (point neighbors/terminal combination)
- every point when terminal is connected
  (terminal connected)

implementation

```
variables_domain = {}
connection_changed = len(connected_terminals)>len(prev_connected_terminal)
first_run = prev_variable == None
if connection_changed or first_run:
    # update all variables
    for coord in variables:
        domain = get_available_domain(coord,
        assignments,connected_terminals)
        variables_domain[coord] = domain
else:
    variables_domain = pickle.loads(pickle.dumps(prev_domain))
    del variables_domain[prev_variable]
    big_neighbors = get_constrained_neighbors(prev_variable,inp,assignments )
    for coord in big_neighbors:
        domain = get_available_domain(coord, assignments,
inp,connected_terminals)
        variables_domain[coord] = domain
return variables_domain
```

results

map	time	Number of hits
5x5	6 ms	17
7x7	16 ms	41
8x8	30 ms	52
9x9 (1)	57 ms	67
10x10 (1)	189 ms	320
10x10(2)	93 ms	139
12x12	290 ms	331
12x14	193 ms	148
14x14	7875 ms	10309

degree heuristic

- use as tie breaker
- choose variable that constrain others
  implementation

```
most_constraining_count = -math.inf
for coord in variables:
    constrained_count = len(
        get_constrained_neighbors(coord,inp, assignments))
    if constrained_count > most_constraining_count:
        most_constraining_count = constrained_count
        most_constraining_var = coord
return most_constraining_var
```

results

- didn't improve
- made heuristic optional

least constraining value

- choose value that doesn't affect domains

implementation

```
count_value_ordered = []
for value in domain:
    updated_variable_domains = get_available_domain_multiple(
        {**{coord: value}, **assignments}, inp, coord,)
    count_constrained = 0

    for coord in updated_variable_domains:
        if len(updated_variable_domains[coord]) < len(variables_domain[coord]):
            count_constrained += 1
```

```
    count_value_ordered.append((count_constrained, value))
count_value_ordered.sort()
order_domain_values = []
for count, value in count_value_ordered:
    order_domain_values += value

return order_domain_values
```

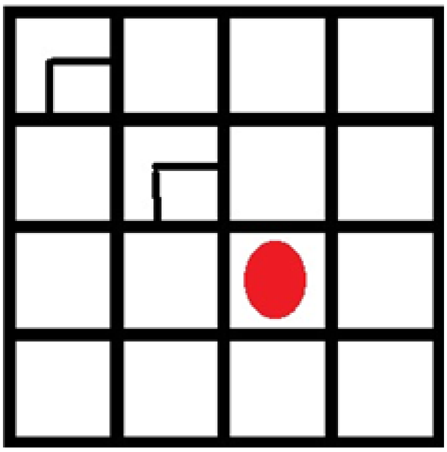
results

map	time	Number of hits
5x5	5 ms	17
7x7	16 ms	56
8x8	23 ms	52
9x9 (1)	65 ms	100
10x10 (1)	166 ms	330
10x10(2)	240 ms	482
12x12	838 ms	1178
12x14	163 ms	146
14x14	2230 ms	2374

Smarter solver

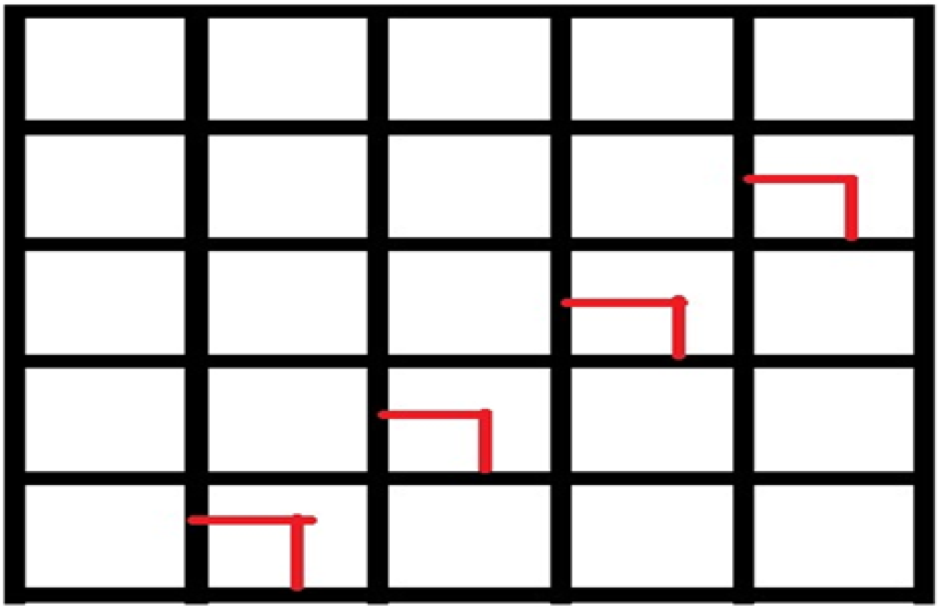
using directions

- We used direction values instead of color values.
- Directions offers better arc consistency
- Values domain is `{'└', '┌', '|', '┘', '—', '┐'}`
- We can use initially_forced constrains
- Four corners have initial single domain value propagating until hitting a number.
- Border variables has three values domain.



```
domain[(0,0)] = {'┌'}
domain[(h-1, 0)] = {'└'}
domain[(h-1, 0)] = {'└'}
domain[(h-1, w-1)] = {'┘'}
domain[(0, i)] = {'┌', '┐', '─'}
domain[(i, 0)] = {'└', '┌', '│'}
domain[(w-1, i)] = {'└', '┘', '─'}
domain[(i, h-1)] = {'┐', '┘', '│'}
```

- Directions have powerful arc consistency that can be used initially to eliminate domain values.



when we eliminate a domain value we can propagate this elimination

- After forced elimination of domain values we can start assignment.
- We should start by one-domain-value variables
- After every assignment, the neighbor variables domains is affected

- We should add variables for colors to check that the correct colors are connected together.
- The directions method solver have almost 3 values domain after initial eliminations
- This is very good branch factor compared to other methods.

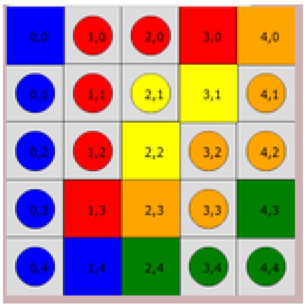
final results for smart

map	time	Number of hits
5x5	5 ms	17
7x7	16 ms	56
8x8	23 ms	52
9x9 (1)	65 ms	100
10x10 (1)	166 ms	330
10x10(2)	240 ms	482
12x12	838 ms	1178
12x14	163 ms	146
14x14	2230 ms	2374

5x5:

For graphical results see figure 1.

```
map ../input/input55.txt solution time = 0.005002737045288086 sec
map ../input/input55.txt number of hits = [17]
BrrRO
bryYo
brYoo
bROoG
bBGgg
```



7x7

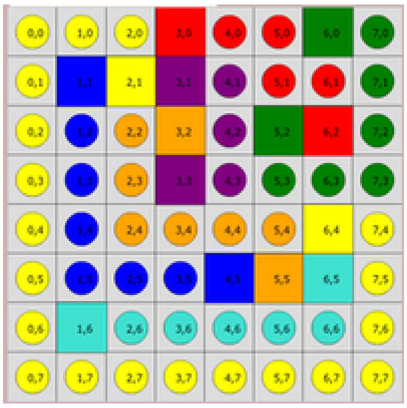
for graphical results see figure 2.

```
map ../input/input77.txt solution time = 0.01697063446044922 sec
map ../input/input77.txt number of hits = [56]
gggOooo
gBggGYo
gbbBRyo
gyyYryo
gyrrryo
gyRyyyo
GyyyOoo
```

8x8

for graphical results see figure 3.

```
map ../input/input88.txt solution time = 0.023000001907348633 sec
map ../input/input88.txt number of hits = [52]
yyyRrrGg
yBYPprrg
yboOpGRg
yboPpggg
ybooooYy
ybbbBOQy
yQqqqqqy
yyyyyyyy
```



9x9

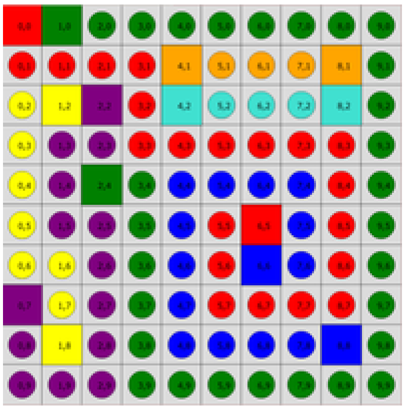
for graphical results see figure 4.

```
map ../input/input991.txt solution time = 0.06594991683959961 sec
map ../input/input991.txt number of hits = [100]
DbbBOKkkk
dbOooRrrk
dbRQqqQrk
DBrrrrrrk
gGkkkkkkk
gkkPppppG
gkYyyyYpg
gkkkkkKPg
ggggggggg
```

10x10 1

for graphical results see figure 5.

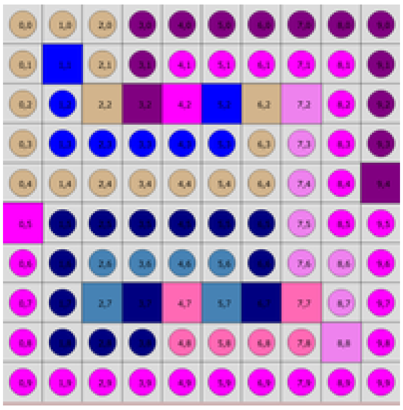
```
map ../input/input10101.txt solution time = 0.16699814796447754 sec
map ../input/input10101.txt number of hits = [330]
RGgggggggg
rrrrOooo0g
yYPrQqqqQg
ypprrrrrrg
ypGgbbbbrg
yppgbrRbrg
yypgbrBbrg
Pypgbrrrrg
pYpgbbbbBg
pppggggggg
```



10x10 2

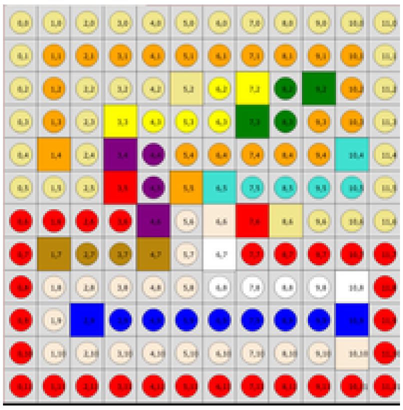
for graphical results see figure 6.

```
map ../input/input10102.txt solution time = 0.24605417251586914 sec
map ../input/input10102.txt number of hits = [482]
tttppppppp
tBtpfffffp
tbTPFBTVfp
tbbbbbtvfp
tttttttvfP
Fnnnnnnvff
fnssssnvvf
fnSNHSNHvf
fnnnhhhhvf
ffffffffff
```



12x12

for graphical results see figure 7.

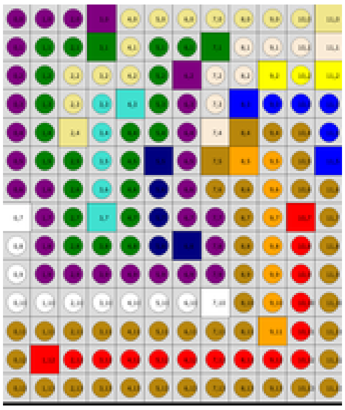


```
map ../input/input1212.txt solution time = 0.8389444351196289 sec
map ../input/input1212.txt number of hits = [1178]
kkkkkkkkkkkk
koooooooooок
kokkkKyYgGok
kokYyyyGgook
kOkPpoooooQk
kkkRpOQqqqqk
rrrrPaARKkkk
rDddDaWrrrrr
raaaaawwwwWr
raBbbbbbbbBr
raaaaaaaaaAr
rrrrrrrrrrrr
```

12x14

for graphical results see figure 8.

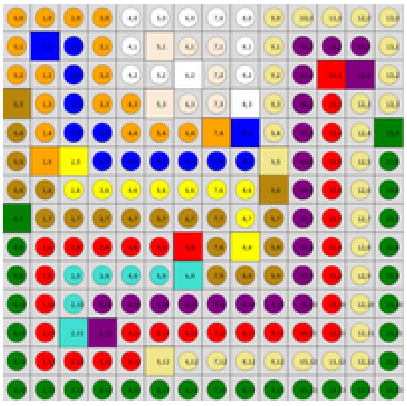
```
map ../input/input1214.txt solution time = 0.16399812698364258 sec
map ../input/input1214.txt number of hits = [146]
pppPkkkkkkkK
pggGkggGaaaA
pgkkkgPaaYyY
pgkqQgpaBbbb
pgKqggpADddb
pggqgNpDOodB
ppgqgnpddodd
WpgQgnppdoRd
wpgggnNpdord
wppppppdord
wwwwwwwWdord
ddddddddOrd
dRrrrrrrrrrd
dddddddddddd
```



14x14

for graphical results see figure 9.

```
map ../input/input1414.txt solution time = 2.230055093765259 sec
map ../input/input1414.txt number of hits = [2374]
ooowwwwwkkkkk
oBbowAaawkpppk
oobowwWawkpRPk
DobooAaaWkprkk
dobbooooBkprkG
dOYbbbbbbKprkg
ddyyyyyyyDprkg
GdddddddydprkG
grrrrrRdYdprkg
grqqqqQdddprkg
grqppppppppprkg
grQPrrrrrrrrkg
grrrrKkkkkkkkg
gggggggggggggg
```



References

Russell, S. J. (2016). Artificial intelligence: A modern approach. Harlow: Pearson.