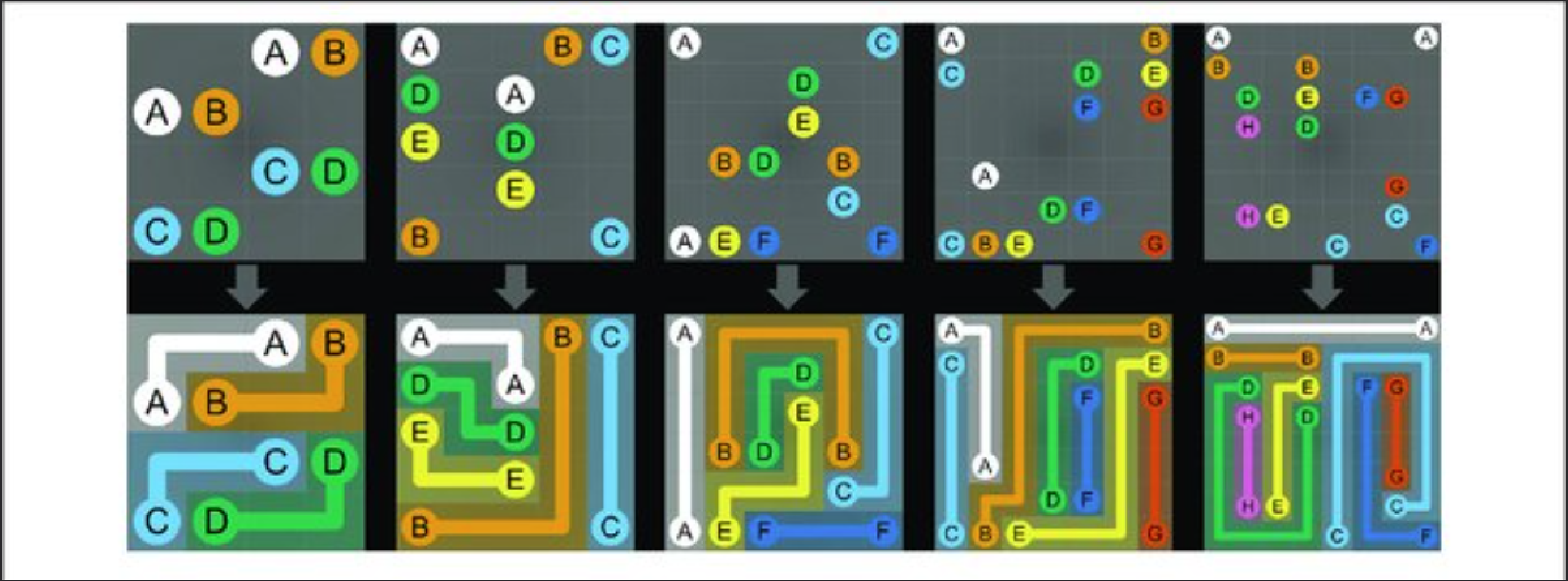
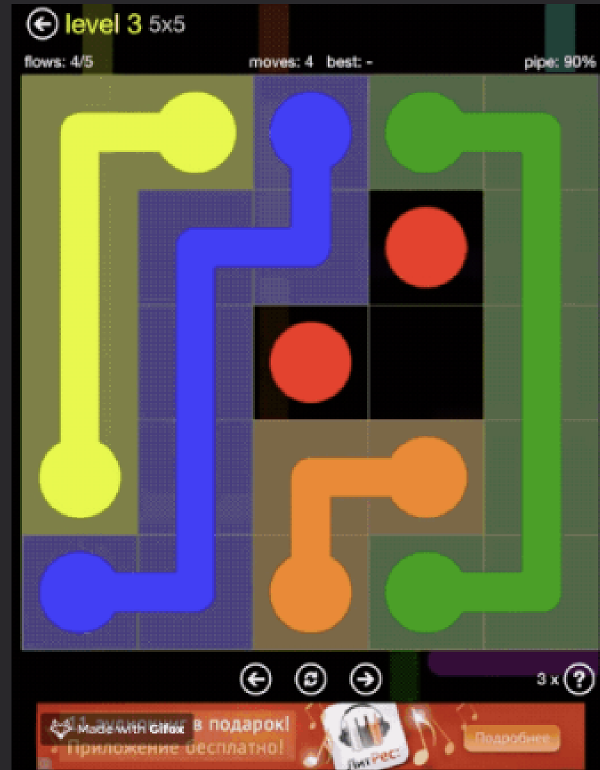
Flow Free - a circuit design problem



In this programming assignment you'll be expected to build a solver for the Flow Free game. Its origins can be traced back to 1897, a puzzle published by the professional puzzler Sam Loyd, in a column he wrote for the *Brooklyn Daily Eagle* (*more details in the Matematica Journal*). This puzzle was popularised in Japan under the name of Numberlink.

Game Rules



The game presents a **grid** with colored dots occupying some of the squares. The objective is to connect dots of the same color by drawing *pipes* between them such that the entire grid is occupied by pipes. However, pipes may not intersect.

The difficulty is determined by the size of the grid, ranging from 5x5 to 15x15 squares.

The Algorithm

Each possible configuration of the Flow Free grid is called a *state*. Each position in the grid can be free or contain a color. The Flow Free Graph $G = \langle V, E \rangle$ is implicitly defined, i.e. the graph is not completely loaded in memory at once, instead, it starts only with the initial state, and the remainder of the graph is discovered while searching for a solution. The vertex set V is defined as all the possible configurations (states), and the edges E connecting two vertexes are defined by the legal movements **(right, left, up, down) for each color.** The code provided makes sure that only legal moves are generated:

> A move is legal only if it extends a color pipe. A color cannot be placed if it's not adjacent to the pipe of the same color. Furthermore, to decrease the number of edges in the Graph, the pipe can only be started from one of the two initial colors, making one of them the start and the other the goal of the pipe.

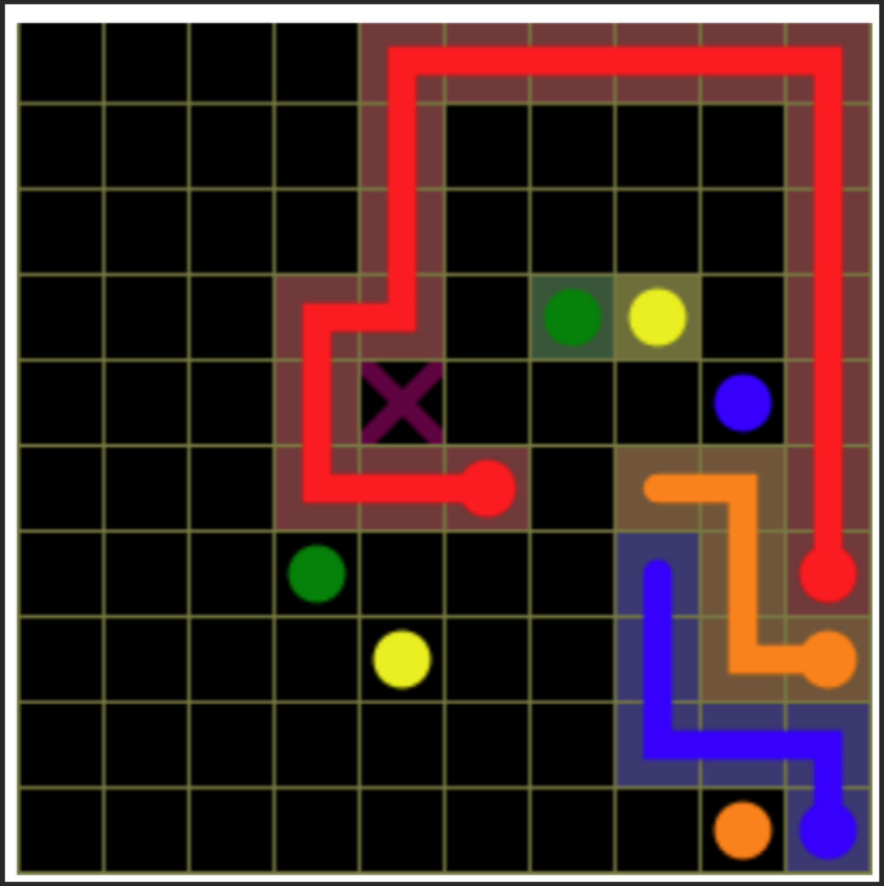
All edges have a weight of 1.

Your task is to find the path traversing the Flow Free Graph from the initial state (vertex) leading to a state (vertex) where all the grid cells contain a color, and therefore finding a solution to the puzzle. A path is a sequence of movements painting a new cell in the grid. You are going to use Dijkstra to find the shortest path first, along with some game-specific optimizations to speed up your algorithm.

The initial node (vertex) corresponds to the initial configuration (Algorithm 1, line 1). The algorithm selects a node to expand (generate the possible pipe extensions, line 5) along with an ordering of which is the *next color* the algorithm should try to connect (line 6). Therefore, when a node is expanded, it has at most 4 children (right, left, up, down). Once all the children have been added to the priority queue, the algorithm will pick again the next node to extend (line 5), as well as the *next color to consider*, until a solution is found. Different color orderings can be tried with the command line options (type `./flow -h` to see all options). Each ordering has a different impact on the number of nodes needed by Dijkstra to find a solution. By default, the search will expand a maximum of 1GB of nodes (line 12). This is checked by a variable named `max_nodes`.

```
GraphSearch(Graph, start)
node ← start
pq ← priority Queue Containing start node Only
while frontier ≠ empty
do
    node ← pq.pop()
    nextColor ← select next color in ordering
    for each move direction
    do
        if move direction for nextColor is valid
          then
              newNode ← applyMoveDirection(node)
              if number of nodes in pq ≥ max_nodes
                then
                    result = SEARCH_FULL
                    break

              if newNode is a dead end
                then
                    deleteNewNode
                    continue

              if newNode is thesolution
                then
                    solution = NewNode
                    result = SEARCH_SUCCESS
                    break

              pq.push(newNode)


free priority queue
return result
```

When you `applyMoveDirection` you have to create a new node that points to the parent, updates the grid resulting from applying the direction chosen, updates the priority of the node, and updates any other auxiliary data in the node. The priority of the node is given by the length of the path from the root node, i.e. how many grid cells have been painted. Check the file `node.h` as this function is already given.



A *dead-end* is a configuration for which you know a solution cannot exist. The figure above shows an example of a dead-end: the cell marked with X cannot be filled with any color. The problem is unsolvable with the current configuration, and we do not need to continue searching any of its successors. You can recognize dead-end cells by looking for a **free cell** in the grid that is **surrounded by three completed path segments (colored) or walls**. The current position of an in-progress pipe and goal position have to be treated as free space. Detecting unsolvable states early can prevent lots of unnecessary search. The search will eventually find a solution. If we recognize dead-ends and do not generate their successors (pruning), the search for a solution will likely take less time and memory, as you'll avoid exploring all possible successors of an unsolvable state.

You might have multiple paths leading to a solution. **Your algorithm should consider the possible actions in the following order**: left (0), right (1), up (2) or down (3).

Make sure you manage the memory well. When you finish running the algorithm, you have to free all the nodes from the memory, otherwise you will have memory leaks. You will notice that the algorithm can run out of memory fairly fast after expanding millions of nodes.

Check the files `utils.h` and `queue.h` where you'll find many of the functions in the algorithm already implemented. Other useful functions are located directly in the file `search.c`, which is the only file you need to edit to write your algorithm inside the function `game_dijkstra_search`. For dead-end detection, you need to look at `extensions.c`, function `game_check_deadends`. Look for the comment `FILL IN THE CODE`. All the files are in the folder `src/`.

Deliverable 1 - Dijkstra **Solver** source code

You are expected to hand in the source code for your solver, written in C. Obviously, your source code is expected to compile and execute flawlessly using the following makefile command:

`make`

generating an executable called

`flow`

Remember to compile using the optimization flag `gcc -O3` for doing your experiments, it will run twice as quickly as compiling with the debugging flag `gcc -g` (see `Makefile`). The provided Makefile compiles with the optimization flag by default, and with the debugging flag if you type `make debug=1`. For the submission, please **do not remove the `-g` option from your Makefile**, as our scripts need this flag for testing. Your program must not be compiled under any flags that prevent it from working under `gdb` or `valgrind`.

Your implementation should be able to solve the *regular* puzzles provided. To solve the *extreme* puzzles, you'll need further enhancements that go beyond the time for this assignment, but feel free to challenge yourself if you finish early and explore how you would solve the extreme puzzles.

The Code Base

You are given a base code. You can compile the code and execute the solver by typing `./flow <puzzleName>` . You are going to have to program your solver in the file `search.c`. Look at the file and implement the missing part in the function called `game_dijkstra_search`. Once you implement the search algorithm, go to the file called `extensions.c` and implement the function called `game_check_deadends`

You are given the structure of a node in `node.*` files, and also a priority queue `queues.*` implementation. Look into the `engine.*` and `utils.*` files to know about the functions you can call to perform the search.

In your final submission, you are free to change any file, but make sure the command line options remain the same.

Input

In order to execute your solver use the following command:

```
./flow [options]  <puzzleName1> ... <puzzleNameN>
```

for example:

```
./flow puzzles/regular_5x5_01.txt
```

Will run the solver for the regular 5 by 5 puzzle, and report if the search was *successful*, the *number of nodes generated* and the *time* taken. if you use `flag -q` (quiet) it will report the solutions more concisely. This option can be useful if you want to run several puzzles at once and study their performance.

If you append the option `-A` it will *animate* the solution found. If you append the option `-d` it will use the *dead-end detection* mechanism that you implemented. Feel free to explore the impact of the other options, specifically the *ordering* in which the colors are explored. By default, the color that has fewer free neighbors (most constrained), is the one that is going to be considered first.

All the options can be found if you use option `-h`:

```
$./flow -h
usage: flow_solver [ OPTIONS ] BOARD1.txt
BOARD2.txt [ ... ] ]

Display options:

  -q, --quiet              Reduce output
  -d, --diagnostics        Print diagnostics when search unsuccessful
  -A, --animation          Animate solution
  -F, --fast               Speed up animation 4x
  -C, --color              Force use of ANSI color
  -S, --svg                Output final state to SVG

Node evaluation options:

  -d, --deadends           dead-end checking

Color ordering options:

  -r, --randomize          Shuffle order of colors before solving
  -c, --constrained        Disable order by most constrained

Search options:

  -n, --max-nodes N        Restrict storage to N nodes
  -m, --max-storage N      Restrict storage to N MB (default 1024)

Help:

  -h, --help               See this help text
```

Output

Your solver will print the following information if option `-q` is used:

1. Puzzle Name
2. SearchFlag (see `utils.c`, line 65-68 to undestand the flags)
3. Total Search Time, in seconds
4. Number of generated nodes
5. A final Summary

For example, the output of your solver `./flow -q ../puzzles/regular_*` could be:

```
../puzzles/regular_5x5_01.txt s 0.000 18
../puzzles/regular_6x6_01.txt s 0.000 283
../puzzles/regular_7x7_01.txt s 0.002 3,317
../puzzles/regular_8x8_01.txt s 0.284 409,726
../puzzles/regular_9x9_01.txt s 0.417 587,332
5 total s 0.704 1,000,676
```

These numbers depend on your implementation of the search, the ordering you use, and whether you prune dead-ends. If we use dead-end pruning `./flow -q -d ../puzzles/regular_*` we get the following results

```
../puzzles/regular_5x5_01.txt s 0.000 17
../puzzles/regular_6x6_01.txt s 0.000 254
../puzzles/regular_7x7_01.txt s 0.001 2,198
../puzzles/regular_8x8_01.txt s 0.137 182,136
../puzzles/regular_9x9_01.txt s 0.210 279,287
5 total s 0.349 463,892
```

Remember that in order to get full marks, your solver has to solve at least the regular puzzles.