Don't touch my fish!

VILLANI Neven

Contents

1 Usage

1.1 `make` targets

By default, `make` builds the `pingouin` executable.
`make test` will run unit tests for all functions of `Priority` and `Bitset` that are part of the original interface, `make perf` will run basic benchmarks.

`make doc` and `make report` build the documentation and the `README.pdf` respectively.

Some other tools are

- `tester.py` which performs path length checks on all files in `problems/`
- `bencher.py` to produce benchmarks for the executable
- `reporter.py` to translate benchmark results to a formatted tabular

Note that source code has been moved to `src/`, the `Makefile` has been adapted in consequence.

1.2 Command-line arguments

For more information run `./pingouin -h`

`pingouin` will accept at most one positional argument being the name of the file that contains the problem. If this argument is not provided the problem will be read from `stdin`.

Named argument take the form `-[CATEGORY][FLAGS]`
where `CATEGORY` is one of `o` (optimizations), `d` (display), `h` (help).

Example (see the optimizations section for details):
There are three optimizations available: `1`, `X`, `T`.

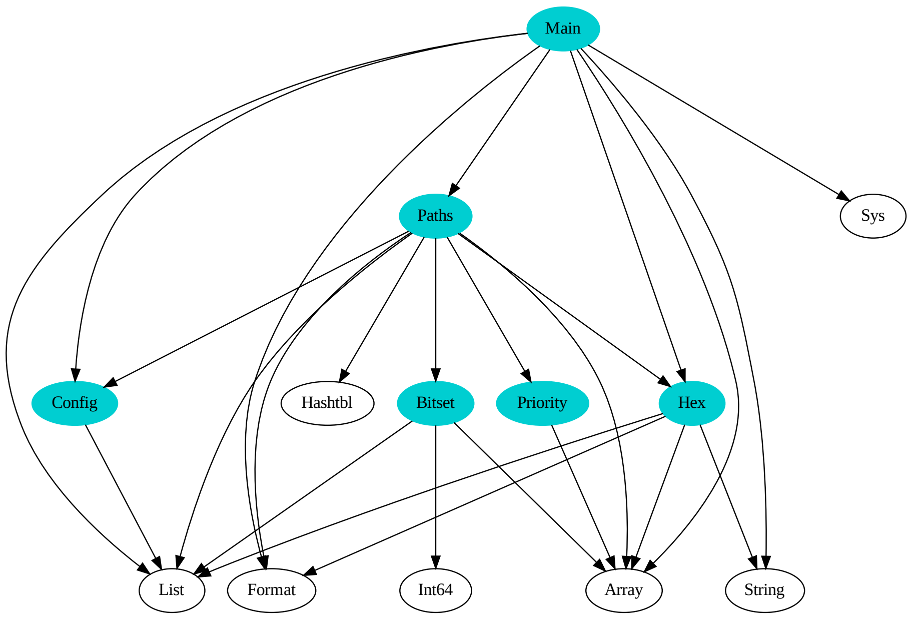
Running `./pingouin -o1X file` will enable optimizations `1` and `X` but disable `T`.
`d` will use its default configuration, as would have `o` were it not explicitly specified.

`h` is special in that its presence will cause other arguments to be ignored and the program will terminate after printing the help message.

2 Implementation

2.1 Structure

Executable dependencies



2.2 `Hex`

Directions were renamed to be more intuitive, and they were removed from the public interface to be accessible only through `all_directions` and the utility functions `opposite` as well as `neighbors`.

`read_grid` was made to be robust : it is able to read problems even if they contain extra lines before `<problem>` or after `</problem>`, and it is able to adapt itself depending on the parity of the line which features indentation.

2.3 `Bitset`

`Bitset.t` uses `Int64.t` as recommended, and makes use of bitwise operators for good performance.

Internally it uses mutable objects to avoid unnecessary copies but it exposes an immutable public interface.

Several utility functions were added to implement standard operations that benefit from not repeatedly converting from and to `int` and from avoiding creating copies.
Those are `setminus`, `union`, `intersect` which are self-explanatory both in name and in implementation as well as the more complex `transitive_closure` which takes a starting point and a `elt -> elt list` neighbor function and calculates the connected component of `start`.
`compare` is useful to sort sets by increasing cardinal.

2.4 `Priority`

`'a node` was made with a mutable key so that `decrease_key` does not invalidate the reference handed out during `insert`.
`find` searches the location of the node in the queue to be used with `member` and `remove`. `find` does not unnecessarily explore the rest of the heap when it finds a key greater than the one being searched, but no effort has been invested into any further improvement of performance, since the two functions `remove` and `decrease_key` are useless for my implementation.

`sift` and `trinkle` perform rotations rather than two-element permutations for some performance gain.

2.5 `Paths`

The finite type represents positions of $[0, \texttt{height}[\times [0, \texttt{width}[$ with the canonical bijection to $[0, \texttt{height} \times \texttt{width}[$.

`HMap` uses a static mutable `Hashtbl.t`, it is important to reset it before an exploration to properly record the positions that have actually been explored.

Keys contain an upper bound on the final length of the path as well as its current length.
Higher priority is given to configurations where the potential is greater, then those whose computation is the most advanced.

A configuration `(remaining_ice, current_position)` is added to the list of already explored configurations when it is time for its children to be added to the priority queue.
This allows proving that `decrease_key` is not required and that whenever a position is marked as "explored" it is indeed unnecessary to consider it again.

Assume `(ice, pos)` is explored and that it is encountered again.
This means that two paths were found that lead to the same configuration.
Since the first one was considered first in the priority queue it must have had higher potential at least up to the previous step. Since the configurations are now the same, one of the following must be true : (a) the two paths are not the same but have the same length, (b) the first path has greater length.
If (a) is the case then the path cannot be strictly improved, and if (b) is the case then the new path is worse than the previous one.
In both instances it is useless to calculate further.

An alternate way of interpreting the priority makes this fact obvious : sorting keys by decreasing potential then by decreasing advancement makes the strategy to be "explore the current path as long as it doesn't waste any space, switch to the next possible path as soon as going on with this path would require sacrificing part of the positions".
This fact explains the efficiency of the optimizations presented thereafter, of which some seem counterintuitive.

3 Optimizations

3.1 `1, X`

In many of the problems an optimal path, or at least a path close to the optimum, can be obtained by only using "single" moves (i.e. of the form `(d,1)`), or "extremal" moves (i.e. of the form either `(d,1)` or `(d,n)` when `(d,n+1)` is not allowed).

It has been observed (see benchmarks section) that searching for an optimal solution with such a restriction of moves sometimes yields a small increase in performance.
The added cost of exploring a whole tree of configurations is compensated by the fact that this tree smaller than the full tree of allowed moves enables removing many branches that are such terrible configurations that it is possible to do better even with more restrictions on allowed moves.

This is implemented in two variants that can be enabled with the option flags `-o1` ("single" moves) and `-oX` ("extremal" moves), or even `-o1X` (both, with "single" first and "extremal" next).

3.2 T

The above optimization (and the algorithm as a whole) is particularly effective when the optimal path is perfect, i.e. goes through all positions, because as soon as a perfect path is found the whole exploration can be stopped. This leads to the purpose of this optimization : to trim positions that are useless to reach in order to make it easier to eliminate branches of the exploration tree.

Procedure:

- find all positions that create a one-way split, i.e. have a position that satisfies the two following conditions: (a) it splits the board into two or more zones, and (b) it is impossible to go from a zone to another without stepping on the position.
- for each of these positions, select the zone that does not contain the starting point
- order these zones by inclusion
- calculate the best possible path for each zone (memoize the results for subsets) and delete all positions that are not part of the optimal path

`large3` is a good example of how this optimization operates.
Initial configuration:

```
  * * *   * *
 * # * * *
* * * *   *
 * * * *   *
  * * * *   * *
```

There are three positions that create a split as described, marked here:

```
  . . .   * .
 . . . . *
. . . .   .
 . . . .   .
  . . . .   * .
```

Two of them have optimal paths that are trivial to calculate:

```
  . . .   a b
 . . . . .
. . . .   .
 . . . .   .
  . . . .   . .
```

and

```
  . . .   . .
 . . . . .
. . . .   .
 . . . .   .
  . . . .   a b
```

The third uses the memoized calculations from the other two to quickly compute the best path

```
  . . .   * *
 . . . . a
. . . .   b
 . . . .   c
  . . . .   d e
```

The two unused positions are deemed useless, and once they are removed the main algorithm will have no difficulty quickly computing a path for the remaining positions, once again reusing the memoized results.

```
  e d c   . .
 f a b s t
h g q r   u
 i j p o   v
  k l m n   w x
```

This last computation is efficient is major part because the optimal path is perfect once the useless positions are deleted.

4 Benchmarks

4.1 Methodology

Each problem is timed with all combinations of optimization flags.
Timeout is set to 30 seconds, any execution error or timeout results in an ∞ for this combination.

Average is calculated on 5, 20 or 50 runs depending on execution time, ensuring that the displayed average has sufficient precision.

To avoid interference, all benchmarks are run consecutively on an otherwise inactive computer.

4.2 Results

	-o	-o1	-oX	-o1X	-oT	-o1T	-oXT	-o1XT
branches	∞	∞	∞	∞	1.64s	1.43s	2.27s	2.16s
choice	57ms	56ms	53ms	52ms	59ms	54ms	60ms	58ms
cutearly01	24ms	27ms	27ms	26ms	28ms	27ms	24ms	27ms
cutearly02	8ms	7ms	7ms	7ms	8ms	8ms	7ms	7ms
display	9ms	9ms	9ms	9ms	32ms	30ms	30ms	29ms
flake1	9ms	9ms	9ms	9ms	9ms	9ms	9ms	10ms
flake2v	60ms	57ms	58ms	58ms	58ms	59ms	58ms	56ms
flake3h	72ms	71ms	78ms	79ms	76ms	67ms	76ms	76ms
flake4hv	332ms	327ms	289ms	292ms	329ms	325ms	288ms	293ms
flake5h	13.4s	12.2s	11.7s	11.8s	13.8s	12.6s	11.9s	11.8s
flake6hv	∞	∞	∞	∞	∞	∞	∞	∞
large1	42ms	28ms	26ms	28ms	41ms	27ms	27ms	28ms
large12	19ms	21ms	19ms	20ms	19ms	21ms	21ms	26ms
large2	463ms	357ms	529ms	482ms	467ms	362ms	534ms	487ms
large3	2.65s	2.19s	3.56s	3.33s	10ms	8ms	9ms	8ms
large4	∞	∞	∞	∞	20ms	31ms	9ms	32ms
narrow	7ms	6ms	7ms	7ms	7ms	7ms	8ms	7ms
rotonde	44ms	45ms	53ms	57ms	50ms	51ms	59ms	59ms
rotonde2	30ms	120ms	26ms	120ms	28ms	122ms	25ms	119ms
scale1	64ms	54ms	64ms	64ms	67ms	56ms	64ms	63ms
scale2	19ms	19ms	8ms	8ms	17ms	19ms	9ms	8ms
scale3	8.65	7.78	5.53	5.52	8.75	7.84	5.54	5.51
scale4	1.02s	849ms	740ms	734ms	1.00s	845ms	747ms	741ms
scale5	104ms	88ms	97ms	98ms	105ms	85ms	100ms	98ms
simple1	7ms	7ms	7ms	7ms	7ms	7ms	7ms	7ms
simple2	6ms	7ms	7ms	7ms	6ms	6ms	7ms	7ms
simple3	7ms	7ms	6ms	7ms	6ms	6ms	7ms	7ms
simple4	7ms	6ms	6ms	6ms	6ms	7ms	6ms	7ms
trefoil	∞	∞	∞	∞	62ms	44ms	48ms	43ms
trimtriple	∞	∞	∞	∞	300ms	296ms	298ms	297ms

The main takeaway is that none of the optimizations are useless : `1` yields a small but noticeable improvement in performance on `flake5h`, `scale4`, `scale5`, `large1`, `large2`, `trefoil`; `X` is useful in the same way for `scale2`, `scale3`, `flake4hv`.
Finally, `T` provides quite spectacular improvements for `branches`, `large4`, `trefoil`, `trimtriple` and slightly less so for `large3`. Only in the pathological case `display` does `T` hinder performance.

Overall, these benchmarks suggest choosing `-o1XT` for best (and most consistent) results, which happens to be the default when `-o` is not specified.