Don't touch my fish!

Villani Neven

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 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
- `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).

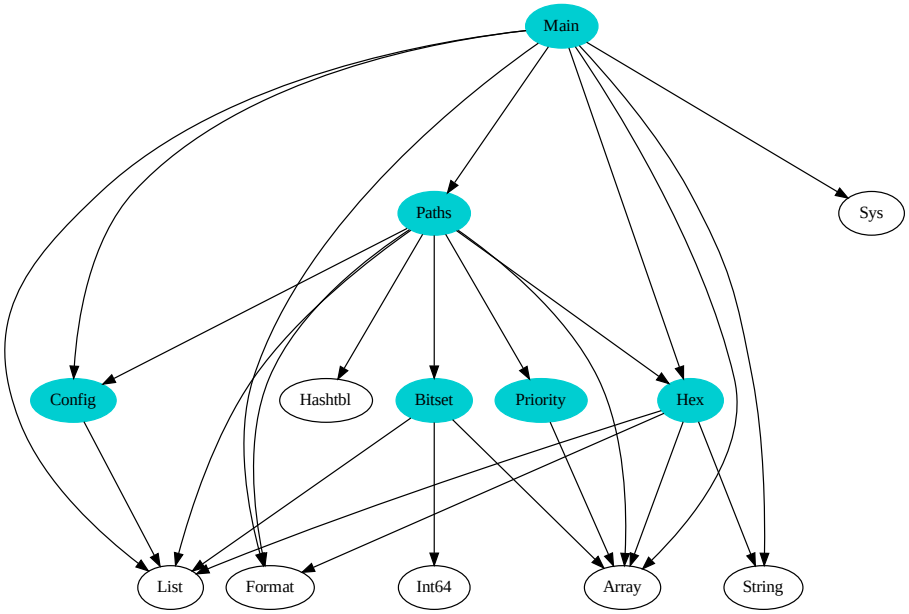
Example (see the optimizations section for details) :
There are three optimizations available : `1`, `X`, `T`.
Running `./pingouin -o1X file` will enable optimizations `1` and `X` but disable `T`.
`d` will use its default configuration, as would have `o` were it not explicitly specified.

`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

	-o	-o1	-oX	-o1X	-oT	-o1T	-oXT	-o1XT
branches	∞	∞	∞	∞	1.50s	1.19s	1.88s	1.80s
choice	54ms	50ms	49ms	47ms	54ms	57ms	60ms	55ms
cutearly01	26ms	27ms	26ms	26ms	24ms	24ms	28ms	26ms
cutearly02	8ms	7ms	7ms	7ms	9ms	8ms	7ms	7ms
display	9ms	8ms	8ms	8ms	20ms	20ms	21ms	20ms
flake1	8ms	9ms	9ms	9ms	8ms	9ms	9ms	9ms
flake2v	57ms	58ms	51ms	50ms	54ms	56ms	59ms	58ms
flake3h	70ms	66ms	77ms	74ms	67ms	63ms	79ms	72ms
flake4hv	299ms	316ms	272ms	276ms	302ms	308ms	274ms	278ms
flake5h	11.4s	9.95	9.40	9.39	11.3s	10.4s	9.51	9.52
flake6hv	∞	∞	∞	∞	∞	∞	∞	∞
large1	41ms	28ms	28ms	28ms	41ms	24ms	23ms	24ms
large12	15ms	15ms	19ms	19ms	18ms	18ms	19ms	23ms
large2	435ms	345ms	500ms	463ms	454ms	351ms	503ms	467ms
large3	2.23s	1.82s	3.03s	2.72s	10ms	8ms	9ms	8ms
large4	∞	∞	∞	∞	21ms	34ms	10ms	31ms
narrow	9ms	6ms	6ms	6ms	7ms	7ms	7ms	7ms
rotonde	41ms	48ms	55ms	54ms	49ms	51ms	56ms	58ms
rotonde2	90ms	179ms	28ms	119ms	91ms	182ms	29ms	118ms
scale1	67ms	56ms	63ms	62ms	67ms	56ms	65ms	63ms
scale2	22ms	21ms	9ms	7ms	18ms	21ms	10ms	9ms
scale3	6.86	6.44	4.46s	4.43s	6.93	6.51	4.50s	4.46s
scale4	928ms	795ms	716ms	692ms	921ms	804ms	707ms	694ms
scale5	102ms	77ms	90ms	93ms	99ms	77ms	95ms	96ms
simple1	7ms	8ms	6ms	6ms	6ms	6ms	6ms	6ms
simple2	6ms	6ms	6ms	6ms	6ms	6ms	6ms	6ms
simple3	6ms	6ms	7ms	7ms	6ms	6ms	6ms	6ms
simple4	6ms	6ms	6ms	6ms	6ms	6ms	6ms	6ms
trefoil	∞	∞	∞	∞	62ms	46ms	46ms	45ms
trimtriple	∞	∞	∞	∞	306ms	294ms	294ms	298ms

The first takeaway is that none of the optimizations are useless : `1` yields a noticeable improvement in performance on `flake5h`, `scale4`, `scale5`, `large1`, `large2`, `trefoil` ; `X` is useful 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 quite unambiguously suggest choosing `-o1XT` for best (and most consistent) results, which happens to be the default when `-o` is not specified.