

Banagrams Solver

Aayush Bajaj

July 10, 2026

Contents

1	the problem	1
1.1	formalisation	2
1.2	hardness	2
2	the haskell solver	2
2.1	the dictionary: multisets, not tries	2
2.2	the grid: a mutable board with an undo stack	3
2.3	the search loop	4
3	the anagram frame	4
4	tries: enumeration under constraints	5
5	a python reference solver	6
6	cost of the search	8
7	see also	8

bananagrams hands you a fistful of letter tiles and one instruction: arrange *all* of them into a connected crossword before anyone else does. this page documents the real solver living in this repo at `static/code/bananagrams/` — a haskell heuristic search in `haskell-imp/` plus a playable js incarnation served at `/code/bananagrams/` on this site — rather than a from-scratch design; a compact python reference solver (trie + backtracking) is developed at the end to make the algorithmic skeleton explicit.

the game ships as a banana-shaped bag of 144 tiles, hence the name. no board, no scores, no turns — first to use every tile shouts bananas and wins

1 the problem

scrabble intuitions mislead here, so start with the contrasts:

- **no board:** words go anywhere on an unbounded grid; there are no premium squares and no fixed centre.
- **no scores:** any legal layout wins. the objective is feasibility, not optimisation.
- **all tiles, exactly:** scrabble lets you sit on bad letters; bananagrams demands you place every tile you hold. the hand is a hard constraint, not a resource.
- **a race:** speed matters between humans. for a solver, only existence matters — find *one* legal layout.

1.1 formalisation

- **input:** a finite multiset M of letters over alphabet Σ , and a dictionary $D \subseteq \Sigma^*$.
- **output:** a finite set $S \subset \mathbb{Z}^2$ and a labelling $g : S \rightarrow \Sigma$ such that
 1. the multiset $\{g(p) : p \in S\}$ equals M — every tile used exactly once;
 2. S is 4-connected — one crossword, not an archipelago;
 3. every maximal horizontal and every maximal vertical run of length ≥ 2 in S , read left-to-right or top-to-bottom, spells a word of D .

condition 3 quantifies over *maximal* runs: placing **cat** flush against **s** silently creates **cats**, and the run that must be a word is the long one. a pleasant consequence of 2 and 3 together: whenever $|S| \geq 2$, every tile sits in at least one word, because connectivity gives each cell a neighbour and hence membership in some run of length ≥ 2 . no dangling singletons to legislate against.

1.2 hardness

related layout problems are provably hard: crossword-puzzle construction — given a dictionary and a grid pattern, fill the pattern so every across and down run is a word — is np-complete.¹ *bananagrams* swaps the fixed pattern for a free one but adds the exactly-once multiset constraint, and no polynomial algorithm is known; treating it as anything other than search is wishful. the *haskell* solver is accordingly an explicit *heuristic*, and says so in its module header:

this module implements a heuristic for finding a valid *bananagrams* layout given a list of characters and valid words. note that for simplicity the heuristic only explores layouts where words in the same direction are not adjacent, and therefore might fail to find a solution even if one exists (this is however unlikely in realistic examples).

that restriction is the classic completeness-for-speed trade: parallel adjacent words (as in a dense scrabble endgame) force validating a quadratic number of incidental cross-runs, while crossing-only layouts keep every new constraint local to one row or column sweep.

2 the haskell solver

three modules under `haskell-imp/`, each owning one concern:

module	owns
<code>Dictionary.hs</code>	which words are spellable, and where they can be anchored
<code>Grid.hs</code>	the mutable board: placement, conflicts, undo, candidates
<code>Bananagrams.hs</code>	the depth-first search loop over entries

(a fourth, `Log.hs`, wires `monad-log` severity-filtered logging to `stderr` so the search narrates itself.)

2.1 the dictionary: multisets, not tries

`Dictionary.hs` stores each word three ways at once — as `Text`, as a `Vector Char` for positional indexing, and as a `Multiset Char` for containment tests — in a vector

¹problem gp14, “crossword puzzle construction”, in m. r. garey and d. s. johnson, *computers and intractability: a guide to the theory of np-completeness*, w. h. freeman, 1979.

sorted by *descending length*:

```
1 spellable :: Hand -> Item -> Bool
2 spellable chars (Item _ _ mset) = mset `Multiset.isSubsetOf` chars
3
4 -- | Returns the best words to use first given the allowed characters.
5 firstWords :: Dictionary -> Hand -> [Text]
6 firstWords (Dictionary items) chars =
7   fmap itemTxt $ Vector.toList $ Vector.filter (spellable chars) items
```

so “which words can i spell” is a linear scan with an $O(|\Sigma|)$ multiset-subset test per word, longest words surfacing first — a greedy tile-burning heuristic. no trie here; the honest note is that the trie earns its keep for *enumeration under letter constraints* (see below), and at realistic hand sizes the flat scan was evidently fast enough.

the subtler routine is `matchingWords`: given the hand, a constraint map of already-placed characters indexed by offset from an anchor, and sweep bounds, it returns every `(word, offset)` alignment that (a) **matches** — each constrained position agrees with the word’s letter there, (b) is **disjoint** — the cells immediately before and after the word are unconstrained, so **cat** never silently extends into **cats**, and (c) is **spellable** — the word’s multiset fits inside the hand plus whatever board letters the alignment reuses. constraints double as free tiles: a reused letter costs nothing from the hand.

2.2 the grid: a mutable board with an undo stack

`Grid.hs` is the imperative heart, running in `ST` for in-place mutation with pure interfaces:

- **letters** live in an unboxed `STUArray s YX Word16` centred on the origin. each cell packs the character into the low byte and a small *reference count* into the byte above it — the count records how many entries cover the cell, which is exactly what makes undo of crossing words correct.
- **blocks** are a `Multiset YX` of forbidden cells: the cell before and after every entry, plus the diagonal neighbours flanking each freshly placed letter. blocks are what enforce “no same-direction adjacency” from the module comment — a multiset rather than a set, so removing one entry’s blocks never erases another’s.
- **changes** are a `Seq Change`, each recording the entry, the blocks it added, and the multiset of characters it actually consumed from the hand (a strict subset of the word when letters were reused). `setEntry` runs in two phases — a pure scan that either finds a **Conflict** (blocked cell, or a mismatched existing letter) or accumulates the **Change**, then a commit — and `unsetLastEntry` pops the sequence and reverses everything. the changes sequence *is* the backtracking stack.
- **candidates**: `extensionPoints` walks current entries and keeps letters covered by exactly one entry (crossings are spent); for each, **candidates** sweeps perpendicular until it hits a block or the array edge, collecting any set characters into the constraint map and returning a **Candidate** with root, orientation, constraints and offset bounds. word *expansions* (turning **cat** into **cats** in-place) are deliberately not candidates.

setChar adds 0x100 per covering entry and unsetChar subtracts it, so removing one of two crossing words leaves the shared letter intact

2.3 the search loop

`Bananagrams.hs` ties it together. `state` is a record of the dictionary, the hand as an `STRef`-wrapped `Multiset Char`, and the grid; `try` places an entry and subtracts the consumed characters, `backtrack` undoes the last entry and refunds them. the whole search is two mutually recursive functions threaded by `firstJust` — “return the first candidate that leads to a completed grid”:

```

1  continueSolve :: LoggableIO m => Bananagrams -> m (Maybe [Entry])
2  continueSolve b@(Bananagrams dict ref grid) = do
3    hand <- liftST $ readSTRef ref
4    if Multiset.null hand
5    then do
6      log0 Informational "Completed"
7      Just <$> liftST (currentEntries grid)
8    else do
9      log1 Debug "Remaining letters: {}" (Shown hand)
10     cands <- liftST $ candidates (Multiset.size hand + 2) grid
11     log1 Debug "Found {} candidates" (length cands)
12     let
13       tryCandidate cand@(Candidate yx orient chars bounds) = do
14         log1 Debug "Trying {}" (Shown cand)
15         firstJust (tryWord yx orient) (matchingWords dict hand chars bounds)
16       tryWord yx orient (word, off) = do
17         let entry = Entry word orient (yx - fromIntegral off * orientationYX
18           ↪ orient)
19         try b entry >> continueSolve b >=> maybe (backtrack b >> pure
20           ↪ Nothing) (pure . Just)
21     firstJust tryCandidate cands

```

read it as textbook depth-first backtracking wearing monadic clothes: **base case** — empty hand, harvest the entries; **recursive case** — enumerate candidate anchors, for each anchor enumerate matching (word, offset) pairs, place, recurse, and on failure `backtrack` and fall through to the next option. `startSolve` seeds the recursion by trying each spellable word as a horizontal first entry at the origin, longest first. the mutable grid plus undo stack means the search keeps a single board in memory, spending $O(\text{len})$ per node rather than copying state — the same discipline as any dancing-links or sudoku solver (Cormen, Thomas H. and Leiserson, Charles E. and Rivest, Ronald L. and Stein, Clifford, 2009).

3 the anagram frame

the recurring primitive is “can this word be spelled from these letters?”, and the right frame is letter counting. represent any string w by its count vector $c(w) \in \mathbb{N}^{26}$; then

$$w \text{ is spellable from hand } h \iff c(w) \leq c(h) \text{ componentwise,} \quad (1)$$

a multiset-containment test costing $O(|\Sigma|)$ regardless of word or hand length. the haskell hand is literally this object — a `Multiset Char` — and the search’s book-keeping is a tidy little ledger over it: placement subtracts (`Multiset.difference` with the *consumed* characters, not the whole word, since crossings reuse board letters), backtracking adds back (`<>`). the exactly-once tile constraint is enforced by construction: the hand can never go negative, and an empty hand is the success condition.

two neighbouring tricks worth knowing: the *canonical anagram key* (sort the letters;

tab and **bat** both become **abt**) turns exact-anagram lookup into a hash probe but cannot answer subset queries; and count vectors compose — the multiset of the whole grid is the sum over entries of consumed characters, which is how condition 1 of the formalisation stays true through arbitrary undo sequences.

4 tries: enumeration under constraints

membership (“is **aisle** a word?”) is cheap in any hashed set. the query that actually shapes a bananagrams solver is *enumeration*: “give me every word spellable from this hand” — and that is where the trie² earns its place (Sedgewick, Robert, 2001). store the dictionary as a rooted tree with one edge per letter, word ends marked; then depth-first walk the trie carrying the hand’s count vector, decrementing on descent and restoring on return, and *abandon a branch the instant a count would go negative*. one dead prefix prunes its entire subtree: if the hand holds no **q**, the whole **q**-subtree — every word starting with **q** — costs a single comparison. the number of nodes visited is exactly the number of spellable prefixes, which for small hands is a vanishing fraction of the dictionary; the flat scan pays $O(|\Sigma|)$ per word, spellable or not. knuth treats tries under digital searching (Knuth, Donald E., 1997); serious word-game engines go one step further and compress the trie into a dawg by merging shared suffixes.³

honesty about this repo: the haskell solver uses no trie at all (length-sorted vector + multiset tests, as above); the js demo builds a genuine prefix trie but then only calls its membership method, doing work a hash set would match; the python solver below finally uses the trie for what it is for.

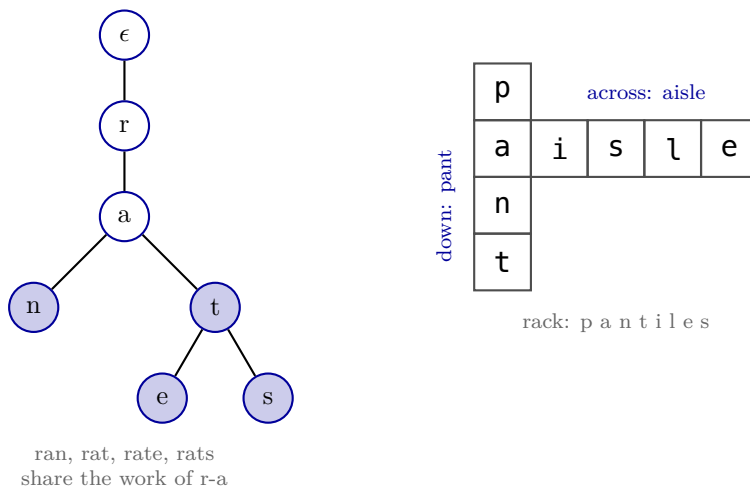


Figure 1: left: a trie fragment storing **ran**, **rat**, **rate** and **rats** — filled nodes mark word ends, and a hand with no **n** prunes the entire left branch in one step. right: the python reference solver’s actual output for the rack **pantiles** — **aisle** crossing **pant** at the shared **a**.

²e. fredkin, *trie memory*, communications of the acm 3(9), 1960 — from *re/trie/val*, which is also why nobody agrees on the pronunciation.

³a. w. appel and g. j. jacobson, *the world’s fastest scrabble program*, communications of the acm 31(5), 1988: the dictionary as a directed acyclic word graph (dawg), plus per-square cross-check sets — the standard playbook for move generation in crossword games.

5 a python reference solver

the same architecture in under a hundred lines, with the trie doing the enumeration this time. dictionary is `/usr/share/dict/words` filtered to lowercase alphabetic words of length 2 to 5 — placement is backtracking over anchored words, with every maximal cross-run validated against the dictionary after each placement:

the length cap is for taste as much as speed: with 8-letter words allowed, greedy longest-first happily played hypaton and quercite — web2 is full of arcana

```
1 """a tiny bananagrams reference solver: trie + counter walk + backtracking
   ↳ placement."""
2 from collections import Counter
3
4 MAXLEN = 5 # cap word length for speed
5
6 def build_trie(path="/usr/share/dict/words"):
7     root = {}
8     with open(path) as f:
9         for line in f:
10             w = line.strip()
11             if 2 <= len(w) <= MAXLEN and w.isalpha() and w.islower():
12                 node = root
13                 for ch in w:
14                     node = node.setdefault(ch, {})
15                 node["$"] = True
16     return root
17
18 def spellable(root, rack):
19     """every dictionary word spellable from the rack: walk the trie,
20     decrementing counts, and abandon a branch the moment a count hits zero."""
21     out, path = [], []
22     def walk(node):
23         if "$" in node:
24             out.append("".join(path))
25         for ch, child in node.items():
26             if ch != "$" and rack[ch] > 0:
27                 rack[ch] -= 1; path.append(ch)
28                 walk(child)
29                 path.pop(); rack[ch] += 1
30     walk(root)
31     return sorted(out, key=len, reverse=True) # longest first: greedy tile
   ↳ burn
32
33 def runs(grid):
34     """all maximal horizontal and vertical runs of length >= 2."""
35     for dr, dc in ((0, 1), (1, 0)):
36         for (r, c) in [p for p in grid if (p[0]-dr, p[1]-dc) not in grid]:
37             s = []
38             while (r, c) in grid:
39                 s.append(grid[(r, c)]); r += dr; c += dc
40             if len(s) >= 2:
41                 yield "".join(s)
42
43 def solve(tiles, words, wordset):
44     rack, grid = Counter(tiles), {}
45
46     def place(word, r, c, dr, dc):
47         """set word at (r,c) going (dr,dc); return new cells or None on
48         ↳ conflict."""
49         need, new = Counter(), []
50         for i, ch in enumerate(word):
51             p = (r + i*dr, c + i*dc)
52             if p in grid:
53                 if grid[p] != ch:
54                     return None
55             else:
56                 need[ch] += 1; new.append((p, ch))
```

```

56     if any(need[ch] > rack[ch] for ch in need):
57         return None # not spellable from the rack
58     if grid and len(new) == len(word):
59         return None # floats free: no intersection
60     if not new:
61         return None # places no tile
62     for p, ch in new:
63         grid[p] = ch
64         rack.subtract(need)
65     return new
66
67 def undo(new):
68     for p, ch in new:
69         del grid[p]
70         rack[ch] += 1
71
72 def backtrack():
73     if not +rack: # every tile used
74         return all(w in wordset for w in runs(grid))
75     anchors = list(grid.items()) or [(0, 0), None]
76     for (r, c), a in anchors:
77         for w in words:
78             for i, ch in enumerate(w):
79                 if a is not None and ch != a:
80                     continue
81                 for dr, dc in ((0, 1), (1, 0)):
82                     new = place(w, r - i*dr, c - i*dc, dr, dc)
83                     if new is None:
84                         continue
85                     if all(x in wordset for x in runs(grid)) and
86                         ↪ backtrack():
87                         return True
88                     undo(new)
89                     if a is None: # first word: one anchor offset is
90                         ↪ enough
91                         break
92                     if a is None:
93                         break
94     return False
95
96 if __name__ == "__main__":
97     tiles = "pantiles" # the 8-tile rack
98     trie = build_trie()
99     words = spellable(trie, Counter(tiles))
100     wordset = set(words)
101     print(f"{len(words)} spellable words, e.g. {words[:5]}")
102     grid = solve(tiles, words, wordset)
103     if grid is None:
104         print("no layout found")
105     else:
106         rs = [p[0] for p in grid]; cs = [p[1] for p in grid]
107         for r in range(min(rs), max(rs) + 1):
108             print("".join(grid.get((r, c), ".") for c in range(min(cs), max(cs)
109                 ↪ + 1)))

```

on the 8-tile rack pantiles it solves in well under a second:

380 spellable words, e.g. ['aisle', 'alien', 'alisp', 'alist', 'alite']

```

p....
aisle
n....
t....

```

structural correspondences with the haskell, module for module: `build_trie + spellable` play `Dictionary.hs` (with the trie upgrade); the `grid` dict, the `place` and `undo` pair and the `runs` validator play `Grid.hs` (with whole-grid run validation replacing the incremental blocks machinery — simpler, costlier, and it permits same-direction adjacency the haskell forbids); `backtrack` is `continueSolve`. one shared incompleteness both inherit: each new word must *cross* an existing one (`len(new) == len(word)` is rejected), so layouts requiring a word laid parallel and flush against another — sharing cross-runs but no cell — are unreachable for the python solver too. and note where the two draw the legality line differently: the haskell trusts `candidates + matchingWords` to only ever propose legal placements (a conflict from `setEntry` is a hard **error**, an invariant violation), while the python generates permissively and lets validation veto — generate-and-test versus constrain-and-generate, the recurring dial in [constraint programming](#).

6 cost of the search

the search tree over entry sequences has:

- **depth** at most $|M|$: every entry consumes at least one tile, and typically $\approx |M|/\bar{\ell}$ for mean placed-word length $\bar{\ell}$.
- **branching factor** = candidate placements per node: (anchors) $\times$ (matching words per anchor) $\times$ (offsets per word). anchors are bounded by tiles already placed; matching words by $|D|$; offsets by word length ℓ . the raw bound $O(|S| \cdot |D| \cdot \ell)$ per node is what the pruning machinery attacks:
 - the multiset-subset test eliminates almost all of D in $O(|\Sigma|)$ per word — and shrinks as the hand shrinks, so the tree narrows with depth;
 - constraint maps from the perpendicular sweep force agreement with already-placed letters, cutting surviving words by orders of magnitude at crowded anchors;
 - blocks (haskell) or whole-grid run validation (python) veto placements that manufacture non-words;
 - longest-first ordering empties the hand in fewer entries, shortening the tree rather than thinning it.
- **space**: $O(|M|)$ beyond the board — one undo record per entry on the path, since both implementations mutate a single grid in place instead of copying state per node.

none of this changes the worst case — exponential it remains, as the hardness discussion promised — but it relocates the exponent to pathological hands. on realistic racks the first descent rarely backtracks more than a handful of times, which is the empirical claim hiding inside the haskell comment’s “unlikely in realistic examples”.

7 see also

- [constraint programming](#) — the propagate-and-branch discipline this solver approximates by hand
- [sudoku](#) — same backtracking skeleton, constraints by row/column/box instead of dictionary

- [hashiwokakero](#) — another feasibility puzzle where connectivity is the awkward global constraint
- [peg solitaire](#) — depth-first search with undo over a mutable board, in the same spirit
- [the train game](#) — small-search-space cousin: enumerate expressions instead of layouts

references

Cormen, Thomas H. and Leiserson, Charles E. and Rivest, Ronald L. and Stein, Clifford (2009). *Introduction to Algorithms*, MIT Press.

Knuth, Donald E. (1997). *The Art of Computer Programming*, Addison-Wesley.

Sedgewick, Robert (2001). *Algorithms in C, Parts 1-4: Fundamentals, Data Structures, Sorting, Searching*, Addison-Wesley.