Package 'minex'

August 30, 2026

Title Automatically Reduce Failing R Scripts to a Minimal Reproducible Example

Version 0.2.0

Description Shrinks a failing R script to the smallest subset of statements that still triggers the same error, using the delta debugging algorithm of Zeller and Hildebrandt (2002) <doi:10.1109/32.988498>. Each candidate reduction is evaluated in a separate R process, so dependencies between statements and their side effects are respected. The result is a one-minimal example, in which removing any remaining statement makes the error disappear; this is the form most useful for bug reports and for questions on community forums. When no statement can be removed, because the failure is nested inside a function body, reduction continues within the surviving statements. A general delta debugging routine and a helper for reducing data frames to the rows that reproduce a failure are also provided.

License MIT + file LICENSE

Encoding UTF-8

Imports callr, stats, utils

Suggests clipr, ellmer, knitr, rmarkdown, spelling, testthat (>= 3.0.0)

VignetteBuilder knitr

Config/testthat/edition 3

Language en-US

URL https://github.com/DIGlabUAB/minex

BugReports https://github.com/DIGlabUAB/minex/issues

Config/roxygen2/version 8.0.0

NeedsCompilation no

Author Sandeep Bodduluri [aut, cph] (ORCID: <https://orcid.org/0000-0001-8201-5262>), Sumanth Chandrupatla [aut, cre, cph] (ORCID: <https://orcid.org/0000-0001-9522-4020>)

Maintainer Sumanth Chandrupatla <srchandr@uab.edu>

Repository CRAN

Date/Publication 2026-08-29 22:40:02 UTC

Contents

`ddmin` *Delta debugging*

Description

General implementation of the `ddmin` minimization algorithm of Zeller and Hildebrandt (2002). Given a collection of elements and a predicate that reports whether a subset still exhibits some behavior of interest, `ddmin()` returns a subset that is *one-minimal*: the predicate holds for it, but fails for every subset obtained by removing a single element.

Usage

```
ddmin(
  items,
  interesting,
  algorithm = c("cdd", "ddmin"),
  max_oracle_calls = Inf,
  verbose = FALSE,
  .info = NULL
)
```

Arguments

`items`	A list or atomic vector of elements to minimize.
`interesting`	A predicate applied to a subset of `items`, in the same form as `items`, returning a single logical. It should return `TRUE` when the subset still reproduces the behavior of interest.
`algorithm`	Character. The reduction strategy for the first phase, one of `"cdd"` (convergent delta debugging, the default) or `"ddmin"` (the classic block-halving loop). Both are followed by a verification sweep that removes any remaining removable single elements.

`max_oracle_calls`
: Numeric. An upper bound on the number of predicate evaluations. Must be at least 1. When the budget is exhausted the reduction stops early and returns the smallest set confirmed so far; the result is still guaranteed to reproduce the behavior but may not be one-minimal.

`verbose`
: Logical or character. If `TRUE`, report each reduction phase via `message()`. If `"trace"`, additionally record a per-call trace in `.info$trace`.

`.info`
: Optional environment. When supplied, `ddmin()` populates it with `oracle_calls` (the number of predicate evaluations), `complete` (whether the reduction finished within budget), and `trace` (a data frame when `verbose = "trace"`, otherwise `NULL`). Metadata flows only through this environment; the returned value itself carries no attributes.

Details

The algorithm partitions the current candidate into `n` blocks (starting with `n = 2`). It first tests whether any single block reproduces the behavior; if so it continues with that block. Otherwise it tests each complement (the candidate with one block removed) and continues with the first that reproduces. If neither succeeds the granularity is doubled, up to the point where each element sits in its own block, which guarantees one-minimality.

Results of the predicate are cached on the set of element indices, so an identical configuration is never evaluated twice.

Value

The one-minimal subset of `items`, in the original order.

References

Zeller A, Hildebrandt R (2002). "Simplifying and Isolating Failure-Inducing Input." *IEEE Transactions on Software Engineering*, 28(2), 183-200. doi:10.1109/32.988498

Zhang M, Xu Z, Tian Y, Cheng X, Sun C (2025). "Toward a Better Understanding of Probabilistic Delta Debugging." ICSE 2025. arXiv:2408.04735. https://arxiv.org/abs/2408.04735

See Also

`minex()` for the script-reduction front end and `reduce_rows()` for reducing data frames.

Examples

```
# Reduce a sentence to the single word a predicate depends on.
words <- strsplit("the quick brown fox", " ")[[1]]
ddmin(words, function(s) "fox" %in% s)

# When several elements are jointly required, all of them are kept.
nums <- 1:6
ddmin(nums, function(s) sum(s) >= 11 && 6 %in% s)
```

`explain_failure` *Explain a minex failure with an LLM*

Description

Hands a `minex()` result's minimal snippet to a language model and returns a structured explanation: why it fails, a fix (optionally executed to verify it is gone), a paste-ready bug report, and a diagnosis. Opt-in; requires the suggested `ellmer` package and a configured chat backend.

Usage

```
explain_failure(
  x,
  chat = NULL,
  verify_fix = TRUE,
  timeout = 60,
  session_info = TRUE,
  modes = c("explain", "fix", "report", "diagnose")
)
```

Arguments

`x`	A `minex_result` from `minex()`.
`chat`	An `ellmer` Chat object (e.g. `ellmer::chat_ollama()`). Defaults to `getOption("minex.chat")`. Used statelessly (cloned per call).
`verify_fix`	Logical. Run the proposed fix to check the failure is gone.
`timeout`	Seconds for the sandboxed fix execution.
`session_info`	Logical. Include `utils::sessionInfo()` in the bug report.
`modes`	Which sections to produce.

Details

Executing model output: with `verify_fix = TRUE` (default) the proposed fix is run in a fresh `callr` process with a timeout. `callr` isolates R session state, not the OS; a fix may still have side effects. Use `verify_fix = FALSE` for input you would not run yourself.

Value

A `minex_explanation` object.

Examples

```
## Not run:
# Write a failing script and reduce it, then explain the failure.
tmp <- tempfile(fileext = ".R")
writeLines("x <- c(1, 2, NA)\nif (is.na(mean(x))) stop('mean is NA')", tmp)
```

```
res <- minex(tmp, backend = "inprocess")
# Requires a configured chat backend (e.g. ellmer::chat_ollama()).
explain_failure(res, chat = ellmer::chat_ollama(model = "llama3.2"),
                verify_fix = FALSE)

## End(Not run)
```

`minex` *Minimize a failing R script to a reproducible example*

Description

Reduces a failing piece of R code to the smallest subset of its top-level statements that still triggers the same failure. The result is a *one-minimal* example: removing any remaining statement makes the failure disappear. This is the form requested when reporting bugs or asking for help, and the part of preparing such an example that is usually done by hand.

Usage

```
minex(
  file = NULL,
  code = NULL,
  clipboard = FALSE,
  oracle = NULL,
  condition = c("error", "warning", "message", "any"),
  match = c("message", "class", "both"),
  algorithm = c("cdd", "ddmin"),
  backend = c("callr", "inprocess"),
  timeout = 60,
  max_oracle_calls = Inf,
  verbose = FALSE,
  granularity = c("statement", "expression")
)
```

Arguments

`file`	Path to a file containing the R code to minimize. Used only when neither `code` nor `clipboard` is supplied.
`code`	A character vector of R source lines, or a single string. Takes precedence over both `clipboard` and `file`.
`clipboard`	Logical. If `TRUE`, read the R code to minimize from the system clipboard. Takes precedence over `file`, but is overridden by `code`. Input precedence is `code > clipboard > file`; supplying more than one source emits a warning and the highest-precedence source is used.
`oracle`	Optional predicate taking a character vector of statements and returning a single logical. When supplied, the target failure is not recorded automatically and `condition`, `match` and failure-point truncation are all bypassed; you are fully responsible for defining what counts as reproducing the failure.

`condition` Which kind of condition to target: `"error"` (the default), `"warning"`, `"message"`, or `"any"` (the most severe condition present, error > warning > message). Ignored when a custom `oracle` is supplied.

`match` How a candidate's failure must match the recorded one when no `oracle` is given: `"message"` (identical message, the default), `"class"` (shares a condition class) or `"both"`. Matching on the message is usually right, because an over-reduced fragment tends to fail with a different message (for example "object not found"). Can also be a function taking two condition summaries, `candidate` and `target` (each a list with `message` and `classes`), and returning a single logical, for custom matching logic.

`algorithm` The reduction strategy passed to `ddmin()`: `"cdd"` (convergent delta debugging, the default) or `"ddmin"` (the classic block-halving loop).

`backend` Either `"callr"` (evaluate each candidate in a fresh R process, the default and the only choice that fully isolates state) or `"inprocess"` (evaluate in the current session, faster but without isolation; a script's side effects other than options are not sandboxed). **Soundness caveat:** `"inprocess"` evaluates in an environment chained to the caller's global environment, so a candidate that references a name which happens to exist in the caller's workspace resolves it instead of raising `object not found`. Over-reduction can therefore spuriously still "succeed" against that tripwire. `backend = "callr"` (the default) evaluates in a clean process and is unaffected; prefer `"inprocess"` only for trusted, quick iteration.

`timeout` Maximum seconds allowed for a single `callr` evaluation.

`max_oracle_calls`
Numeric upper bound on the number of oracle evaluations in the reduction search, passed to `ddmin()` (and to the HDD pass when `granularity = "expression"`). When the budget is exhausted the reduction stops early with a warning; the result still reproduces the failure but may not be one-minimal. The one-time failure-point truncation probe is mandatory setup, exempt from this limit, but is still included in the reported `oracle_calls`. The budget bounds the whole call: when the statement pass escalates to `"expression"` (see `granularity`), the second pass runs on whatever the first left rather than on a fresh budget.

`verbose` Logical. If `TRUE`, report progress. If `"trace"`, also populate the result's `trace` with a per-oracle-call record. For `granularity = "expression"`, the HDD trace rows additionally carry `stmt_index` (which statement they reduced) and `level` (the HDD tree depth); the statement-level rows have `NA` in both. For `granularity = "statement"` the trace is unchanged from 0.2.0 (no `stmt_index`/`level` columns).

`granularity` `"statement"` (the default) reduces only at the level of top-level statements. `"expression"` additionally reduces *within* each surviving statement via HDD (hierarchical delta debugging): pipeline stages (`|>`, magrittr `%>%`) and positional call arguments can be dropped from a statement as long as the whole kept set still reproduces the target failure. When such a reduction makes an earlier statement redundant (for example a value dropped from a later call), that statement is swept out, so the result stays statement one-minimal.

When `granularity` is left at its default *and* statement-level reduction removes nothing, `minex()` retries once at `"expression"` rather than returning the input

unchanged. This is the common case for a script that is one function definition plus a call, where every top-level statement is load-bearing but the failure is nested inside the function body. The retried result carries `escalated_from = "statement"`, and its `code` is a simplification of the original statements rather than a subset of them. Passing `granularity = "statement"` explicitly suppresses the retry and reduces at statement level only. No retry happens when the statement pass stopped early against `max_oracle_calls`, since it has not then shown that nothing is removable.

Details

When no subset of the top-level statements can be removed – the usual case when the failure is nested inside a function body – `minex()` continues *within* the surviving statements instead of returning the input unchanged, so the reduced code is then a simplification of the original statements rather than a subset of them. See `granularity`.

By default `minex()` first runs the whole input to record the failure it produces (its condition message and class), then uses `ddmin()` to search for a minimal subset that reproduces it. Each candidate is evaluated in a separate R process so that dependencies between statements and their side effects are respected; removing a statement that a later one needs typically changes the error, and such a removal is therefore rejected.

Supply a custom `oracle` to minimize against any condition you can express as a predicate, rather than against the recorded failure. The oracle receives a character vector of statements and must return a single logical.

Value

An object of class `"minex_result"`: a list with the minimized `code` (a character vector of statements), the `original` statements, the statement counts `n_original` and `n_minimal`, the character counts `n_chars_original` and `n_chars_minimal` (`nchar()` of the code collapsed to a single string, before and after reduction), the number of `oracle_calls` (every predicate evaluation, including the failure-point truncation probe), the recorded `target` failure (or `NULL` for a custom oracle), the `granularity` setting used, and the `match` and `backend` settings.

When the statement pass escalated to `"expression"`, the result also carries `escalated_from = "statement"` and `coarse_oracle_calls`, the share of `oracle_calls` spent on the discarded statement-level pass. Both are absent otherwise, so `is.null(res$escalated_from)` distinguishes a result that was reduced at the granularity asked for from one that had to descend.

See Also

`ddmin()` for the underlying algorithm and `reduce_rows()` for reducing data frames.

Examples

```
# A failing script padded with irrelevant setup.
script <- c(
  "a <- 10",
  "b <- 20",
  "log('not a number')"
)
```

```
res <- minex(code = script, backend = "inprocess")
res
cat(as.character(res), "\n")

# A failure that genuinely depends on an earlier statement: both are kept.
script2 <- c(
  "x <- c(1, 2, NA)",
  "m <- mean(x)",
  "if (is.na(m)) stop('mean is NA')"
)
minex(code = script2, backend = "inprocess")


# The default backend runs candidates in fresh R processes.
minex(code = script)


## Not run:
# Reduce a failing script copied to the system clipboard:
minex(clipboard = TRUE)

## End(Not run)
```

`reduce_rows` *Reduce a data frame to the rows that reproduce a failure*

Description

Often a bug only shows up with a large data frame, even though a handful of rows is enough to trigger it. `reduce_rows()` applies `ddmin()` over the rows of `data` and returns the smallest subset for which `predicate` still holds, preserving the original row order. The result is typically small enough to paste into a bug report with `dput()`.

Usage

```
reduce_rows(
  data,
  predicate,
  algorithm = c("cdd", "ddmin"),
  max_oracle_calls = Inf,
  verbose = FALSE
)
```

Arguments

`data`	A data frame.
`predicate`	A function taking a data frame (a subset of `data`'s rows) and returning a single logical: `TRUE` when the subset still reproduces the failure of interest.

`algorithm` Character. The reduction strategy for the first phase, one of `"cdd"` (default) or `"ddmin"`. Passed through to `ddmin()`.

`max_oracle_calls` Numeric. An upper bound on the number of predicate calls. Passed through to `ddmin()`.

`verbose` Logical. If `TRUE`, report progress.

Value

A data frame containing the one-minimal subset of rows.

See Also

`ddmin()`, `minex()`.

Examples

```
df <- data.frame(id = 1:6, value = c(3, 8, 999, 2, 5, 7))
# The failure: any value greater than 100.
reduce_rows(df, function(d) any(d$value > 100))
```

Index