backtracking

Search algorithm

Search algorithm
.mw-parser-output .hatnote{font-style:italic}.mw-parser-output div.hatnote{padding-left:1.6em;margin-bottom:0.5em}.mw-parser-output .hatnote i{font-style:normal}.mw-parser-output .hatnote+link+.hatnote{margin-top:-0.5em}For the line search algorithm used in unconstrained optimization, see Backtracking line search.
Backtracking is a class of algorithms for finding solutions to some computational problems, notably constraint satisfaction problems, that incrementally builds candidates to the solutions, and abandons a candidate ("backtracks") as soon as it determines that the candidate cannot possibly be completed to a valid solution.[1]
The classic textbook example of the use of backtracking is the eight queens puzzle, that asks for all arrangements of eight chess queens on a standard chessboard so that no queen attacks any other. In the common backtracking approach, the partial candidates are arrangements of k queens in the first k rows of the board, all in different rows and columns. Any partial solution that contains two mutually attacking queens can be abandoned.
Backtracking can be applied only for problems which admit the concept of a "partial candidate solution" and a relatively quick test of whether it can possibly be completed to a valid solution. It is useless, for example, for locating a given value in an unordered table. When it is applicable, however, backtracking is often much faster than brute-force enumeration of all complete candidates, since it can eliminate many candidates with a single test.
Backtracking is an important tool for solving constraint satisfaction problems,[2] such as crosswords, verbal arithmetic, Sudoku, and many other puzzles. It is often the most convenient technique for parsing,[3] for the knapsack problem and other combinatorial optimization problems. It is also the basis of the so-called logic programming languages such as Icon, Planner and Prolog.
Backtracking depends on user-given "black box procedures" that define the problem to be solved, the nature of the partial candidates, and how they are extended into complete candidates. It is therefore a metaheuristic rather than a specific algorithm – although, unlike many other meta-heuristics, it is guaranteed to find all solutions to a finite problem in a bounded amount of time.
The term "backtrack" was coined by American mathematician D. H. Lehmer in the 1950s.[4] The pioneer string-processing language SNOBOL (1962) may have been the first to provide a built-in general backtracking facility.
Description of the method[edit]
The backtracking algorithm enumerates a set of partial candidates that, in principle, could be completed in various ways to give all the possible solutions to the given problem. The completion is done incrementally, by a sequence of candidate extension steps.
Conceptually, the partial candidates are represented as the nodes of a tree structure, the potential search tree. Each partial candidate is the parent of the candidates that differ from it by a single extension step; the leaves of the tree are the partial candidates that cannot be extended any further.
The backtracking algorithm traverses this search tree recursively, from the root down, in depth-first order. At each node c, the algorithm checks whether c can be completed to a valid solution. If it cannot, the whole sub-tree rooted at c is skipped (pruned). Otherwise, the algorithm (1) checks whether c itself is a valid solution, and if so reports it to the user; and (2) recursively enumerates all sub-trees of c. The two tests and the children of each node are defined by user-given procedures.
Therefore, the actual search tree that is traversed by the algorithm is only a part of the potential tree. The total cost of the algorithm is the number of nodes of the actual tree times the cost of obtaining and processing each node. This fact should be considered when choosing the potential search tree and implementing the pruning test.
Pseudocode[edit]
In order to apply backtracking to a specific class of problems, one must provide the data P for the particular instance of the problem that is to be solved, and six procedural parameters, root, reject, accept, first, next, and output. These procedures should take the instance data P as a parameter and should do the following:
root(P): return the partial candidate at the root of the search tree.
reject(P,c): return true only if the partial candidate c is not worth completing.
accept(P,c): return true if c is a solution of P, and false otherwise.
first(P,c): generate the first extension of candidate c.
next(P,s): generate the next alternative extension of a candidate, after the extension s.
output(P,c): use the solution c of P, as appropriate to the application.
The backtracking algorithm reduces the problem to the call backtrack(root(P)), where backtrack is the following recursive procedure:
procedure backtrack(P, c) is
if reject(P, c) then return
if accept(P, c) then output(P, c)
s ← first(P, c)
while s ≠ NULL do
backtrack(P, s)
s ← next(P, s)
Usage considerations[edit]
The reject procedure should be a boolean-valued function that returns true only if it is certain that no possible extension of c is a valid solution for P. If the procedure cannot reach a definite conclusion, it should return false. An incorrect true result may cause the backtrack procedure to miss some valid solutions. The procedure may assume that reject(P,t) returned false for every ancestor t of c in the search tree.
On the other hand, the efficiency of the backtracking algorithm depends on reject returning true for candidates that are as close to the root as possible. If reject always returns false, the algorithm will still find all solutions, but it will be equivalent to a brute-force search.
The accept procedure should return true if c is a complete and valid solution for the problem instance P, and false otherwise. It may assume that the partial candidate c and all its ancestors in the tree have passed the reject test.
The general pseudo-code above does not assume that the valid solutions are always leaves of the potential search tree. In other words, it admits the possibility that a valid solution for P can be further extended to yield other valid solutions.
The first and next procedures are used by the backtracking algorithm to enumerate the children of a node c of the tree, that is, the candidates that differ from c by a single extension step. The call first(P,c) should yield the first child of c, in some order; and the call next(P,s) should return the next sibling of node s, in that order. Both functions should return a distinctive "NULL" candidate, if the requested child does not exist.
Together, the root, first, and next functions define the set of partial candidates and the potential search tree. They should be chosen so that every solution of P occurs somewhere in the tree, and no partial candidate occurs more than once. Moreover, they should admit an efficient and effective reject predicate.
Early stopping variants[edit]
The pseudo-code above will call output for all candidates that are a solution to the given instance P. The algorithm can be modified to stop after finding the first solution, or a specified number of solutions; or after testing a specified number of partial candidates, or after spending a given amount of CPU time.
Examples[edit]
A Sudoku solved by backtracking.
Examples where backtracking can be used to solve puzzles or problems include:
Puzzles such as eight queens puzzle, crosswords, verbal arithmetic, Sudoku[nb 1], and Peg Solitaire.
Combinatorial optimization problems such as parsing and the knapsack problem.
Logic programming languages such as Icon, Planner and Prolog, which use backtracking internally to generate answers.
The following is an example where backtracking is used for the constraint satisfaction problem:
Constraint satisfaction[edit]
The general constraint satisfaction problem consists in finding a list of integers x = (x[1], x[2], …, x[n]), each in some range {1, 2, …, m}, that satisfies some arbitrary constraint (boolean function) F.
For this class of problems, the instance data P would be the integers m and n, and the predicate F. In a typical backtracking solution to this problem, one could define a partial candidate as a list of integers c = (c[1], c[2], …, c[k]), for any k between 0 and n, that are to be assigned to the first k variables x[1], x[2], …, x[k]. The root candidate would then be the empty list (). The first and next procedures would then be
function first(P, c) is
k ← length(c)
if k = n then
return NULL
else
return (c[1], c[2], ..., c[k], 1)
function next(P, s) is
k ← length(s)
if s[k] = m then
return NULL
else
return (s[1], s[2], ..., s[k − 1], 1 + s[k])
Here length(c) is the number of elements in the list c.
The call reject(P, c) should return true if the constraint F cannot be satisfied by any list of n integers that begins with the k elements of c. For backtracking to be effective, there must be a way to detect this situation, at least for some candidates c, without enumerating all those mn − k n-tuples.
For example, if F is the conjunction of several boolean predicates, F = F[1] ∧ F[2] ∧ … ∧ F[p], and each F[i] depends only on a small subset of the variables x[1], …, x[n], then the reject procedure could simply check the terms F[i] that depend only on variables x[1], …, x[k], and return true if any of those terms returns false. In fact, reject needs only check those terms that do depend on x[k], since the terms that depend only on x[1], …, x[k − 1] will have been tested further up in the search tree.
Assuming that reject is implemented as above, then accept(P, c) needs only check whether c is complete, that is, whether it has n elements.
It is generally better to order the list of variables so that it begins with the most critical ones (i.e. the ones with fewest value options, or which have a greater impact on subsequent choices).
One could also allow the next function to choose which variable should be assigned when extending a partial candidate, based on the values of the variables already assigned by it. Further improvements can be obtained by the technique of constraint propagation.
In addition to retaining minimal recovery values used in backing up, backtracking implementations commonly keep a variable trail, to record value change history. An efficient implementation will avoid creating a variable trail entry between two successive changes when there is no choice point, as the backtracking will erase all of the changes as a single operation.
An alternative to the variable trail is to keep a timestamp of when the last change was made to the variable. The timestamp is compared to the timestamp of a choice point. If the choice point has an associated time later than that of the variable, it is unnecessary to revert the variable when the choice point is backtracked, as it was changed before the choice point occurred.
See also[edit]
Ariadne's thread (logic)
Backjumping
Backward chaining
Enumeration algorithm
Sudoku solving algorithms
Notes[edit]
.mw-parser-output .reflist{font-size:90%;margin-bottom:0.5em;list-style-type:decimal}.mw-parser-output .reflist .references{font-size:100%;margin-bottom:0;list-style-type:inherit}.mw-parser-output .reflist-columns-2{column-width:30em}.mw-parser-output .reflist-columns-3{column-width:25em}.mw-parser-output .reflist-columns{margin-top:0.3em}.mw-parser-output .reflist-columns ol{margin-top:0}.mw-parser-output .reflist-columns li{page-break-inside:avoid;break-inside:avoid-column}.mw-parser-output .reflist-upper-alpha{list-style-type:upper-alpha}.mw-parser-output .reflist-upper-roman{list-style-type:upper-roman}.mw-parser-output .reflist-lower-alpha{list-style-type:lower-alpha}.mw-parser-output .reflist-lower-greek{list-style-type:lower-greek}.mw-parser-output .reflist-lower-roman{list-style-type:lower-roman}
^ See Sudoku solving algorithms.
References[edit]
^ .mw-parser-output cite.citation{font-style:inherit;word-wrap:break-word}.mw-parser-output .citation q{quotes:"\"""\"""'""'"}.mw-parser-output .citation:target{background-color:rgba(0,127,255,0.133)}.mw-parser-output .id-lock-free a,.mw-parser-output .citation .cs1-lock-free a{background:url("//upload.wikimedia.org/wikipedia/commons/6/65/Lock-green.svg")right 0.1em center/9px no-repeat}.mw-parser-output .id-lock-limited a,.mw-parser-output .id-lock-registration a,.mw-parser-output .citation .cs1-lock-limited a,.mw-parser-output .citation .cs1-lock-registration a{background:url("//upload.wikimedia.org/wikipedia/commons/d/d6/Lock-gray-alt-2.svg")right 0.1em center/9px no-repeat}.mw-parser-output .id-lock-subscription a,.mw-parser-output .citation .cs1-lock-subscription a{background:url("//upload.wikimedia.org/wikipedia/commons/a/aa/Lock-red-alt-2.svg")right 0.1em center/9px no-repeat}.mw-parser-output .cs1-ws-icon a{background:url("//upload.wikimedia.org/wikipedia/commons/4/4c/Wikisource-logo.svg")right 0.1em center/12px no-repeat}.mw-parser-output .cs1-code{color:inherit;background:inherit;border:none;padding:inherit}.mw-parser-output .cs1-hidden-error{display:none;color:#d33}.mw-parser-output .cs1-visible-error{color:#d33}.mw-parser-output .cs1-maint{display:none;color:#3a3;margin-left:0.3em}.mw-parser-output .cs1-format{font-size:95%}.mw-parser-output .cs1-kern-left{padding-left:0.2em}.mw-parser-output .cs1-kern-right{padding-right:0.2em}.mw-parser-output .citation .mw-selflink{font-weight:inherit}Gurari, Eitan (1999). "CIS 680: DATA STRUCTURES: Chapter 19: Backtracking Algorithms". Archived from the original on 17 March 2007.
^ Biere, A.; Heule, M.; van Maaren, H. (29 January 2009). Handbook of Satisfiability. IOS Press. ISBN 978-1-60750-376-7.
^ Watson, Des (22 March 2017). A Practical Approach to Compiler Construction. Springer. ISBN 978-3-319-52789-5.
^ Rossi, Francesca; van Beek, Peter; Walsh, Toby (August 2006). "Constraint Satisfaction: An Emerging Paradigm". Handbook of Constraint Programming. Amsterdam: Elsevier. p. 14. ISBN 978-0-444-52726-4. Retrieved 30 December 2008.
Further reading[edit]
.mw-parser-output .refbegin{font-size:90%;margin-bottom:0.5em}.mw-parser-output .refbegin-hanging-indents>ul{margin-left:0}.mw-parser-output .refbegin-hanging-indents>ul>li{margin-left:0;padding-left:3.2em;text-indent:-3.2em}.mw-parser-output .refbegin-hanging-indents ul,.mw-parser-output .refbegin-hanging-indents ul li{list-style:none}@media(max-width:720px){.mw-parser-output .refbegin-hanging-indents>ul>li{padding-left:1.6em;text-indent:-1.6em}}.mw-parser-output .refbegin-columns{margin-top:0.3em}.mw-parser-output .refbegin-columns ul{margin-top:0}.mw-parser-output .refbegin-columns li{page-break-inside:avoid;break-inside:avoid-column}
Gilles Brassard, Paul Bratley (1995). Fundamentals of Algorithmics. Prentice-Hall. ISBN 9780133350685.
External links[edit]
HBmeyer.de, Interactive animation of a backtracking algorithm
Solving Combinatorial Problems with STL and Backtracking, Article and C++ source code for a generic implementation of backtracking
.mw-parser-output .hlist dl,.mw-parser-output .hlist ol,.mw-parser-output .hlist ul{margin:0;padding:0}.mw-parser-output .hlist dd,.mw-parser-output .hlist dt,.mw-parser-output .hlist li{margin:0;display:inline}.mw-parser-output .hlist.inline,.mw-parser-output .hlist.inline dl,.mw-parser-output .hlist.inline ol,.mw-parser-output .hlist.inline ul,.mw-parser-output .hlist dl dl,.mw-parser-output .hlist dl ol,.mw-parser-output .hlist dl ul,.mw-parser-output .hlist ol dl,.mw-parser-output .hlist ol ol,.mw-parser-output .hlist ol ul,.mw-parser-output .hlist ul dl,.mw-parser-output .hlist ul ol,.mw-parser-output .hlist ul ul{display:inline}.mw-parser-output .hlist .mw-empty-li{display:none}.mw-parser-output .hlist dt::after{content:": "}.mw-parser-output .hlist dd::after,.mw-parser-output .hlist li::after{content:" · ";font-weight:bold}.mw-parser-output .hlist dd:last-child::after,.mw-parser-output .hlist dt:last-child::after,.mw-parser-output .hlist li:last-child::after{content:none}.mw-parser-output .hlist dd dd:first-child::before,.mw-parser-output .hlist dd dt:first-child::before,.mw-parser-output .hlist dd li:first-child::before,.mw-parser-output .hlist dt dd:first-child::before,.mw-parser-output .hlist dt dt:first-child::before,.mw-parser-output .hlist dt li:first-child::before,.mw-parser-output .hlist li dd:first-child::before,.mw-parser-output .hlist li dt:first-child::before,.mw-parser-output .hlist li li:first-child::before{content:" (";font-weight:normal}.mw-parser-output .hlist dd dd:last-child::after,.mw-parser-output .hlist dd dt:last-child::after,.mw-parser-output .hlist dd li:last-child::after,.mw-parser-output .hlist dt dd:last-child::after,.mw-parser-output .hlist dt dt:last-child::after,.mw-parser-output .hlist dt li:last-child::after,.mw-parser-output .hlist li dd:last-child::after,.mw-parser-output .hlist li dt:last-child::after,.mw-parser-output .hlist li li:last-child::after{content:")";font-weight:normal}.mw-parser-output .hlist ol{counter-reset:listitem}.mw-parser-output .hlist ol>li{counter-increment:listitem}.mw-parser-output .hlist ol>li::before{content:" "counter(listitem)"\a0 "}.mw-parser-output .hlist dd ol>li:first-child::before,.mw-parser-output .hlist dt ol>li:first-child::before,.mw-parser-output .hlist li ol>li:first-child::before{content:" ("counter(listitem)"\a0 "}.mw-parser-output .navbox{box-sizing:border-box;border:1px solid #a2a9b1;width:100%;clear:both;font-size:88%;text-align:center;padding:1px;margin:1em auto 0}.mw-parser-output .navbox .navbox{margin-top:0}.mw-parser-output .navbox+.navbox,.mw-parser-output .navbox+.navbox-styles+.navbox{margin-top:-1px}.mw-parser-output .navbox-inner,.mw-parser-output .navbox-subgroup{width:100%}.mw-parser-output .navbox-group,.mw-parser-output .navbox-title,.mw-parser-output .navbox-abovebelow{padding:0.25em 1em;line-height:1.5em;text-align:center}.mw-parser-output .navbox-group{white-space:nowrap;text-align:right}.mw-parser-output .navbox,.mw-parser-output .navbox-subgroup{background-color:#fdfdfd}.mw-parser-output .navbox-list{line-height:1.5em;border-color:#fdfdfd}.mw-parser-output .navbox-list-with-group{text-align:left;border-left-width:2px;border-left-style:solid}.mw-parser-output tr+tr>.navbox-abovebelow,.mw-parser-output tr+tr>.navbox-group,.mw-parser-output tr+tr>.navbox-image,.mw-parser-output tr+tr>.navbox-list{border-top:2px solid #fdfdfd}.mw-parser-output .navbox-title{background-color:#ccf}.mw-parser-output .navbox-abovebelow,.mw-parser-output .navbox-group,.mw-parser-output .navbox-subgroup .navbox-title{background-color:#ddf}.mw-parser-output .navbox-subgroup .navbox-group,.mw-parser-output .navbox-subgroup .navbox-abovebelow{background-color:#e6e6ff}.mw-parser-output .navbox-even{background-color:#f7f7f7}.mw-parser-output .navbox-odd{background-color:transparent}.mw-parser-output .navbox .hlist td dl,.mw-parser-output .navbox .hlist td ol,.mw-parser-output .navbox .hlist td ul,.mw-parser-output .navbox td.hlist dl,.mw-parser-output .navbox td.hlist ol,.mw-parser-output .navbox td.hlist ul{padding:0.125em 0}.mw-parser-output .navbox .navbar{display:block;font-size:100%}.mw-parser-output .navbox-title .navbar{float:left;text-align:left;margin-right:0.5em}.mw-parser-output .navbar{display:inline;font-size:88%;font-weight:normal}.mw-parser-output .navbar-collapse{float:left;text-align:left}.mw-parser-output .navbar-boxtext{word-spacing:0}.mw-parser-output .navbar ul{display:inline-block;white-space:nowrap;line-height:inherit}.mw-parser-output .navbar-brackets::before{margin-right:-0.125em;content:"[ "}.mw-parser-output .navbar-brackets::after{margin-left:-0.125em;content:" ]"}.mw-parser-output .navbar li{word-spacing:-0.125em}.mw-parser-output .navbar a>span,.mw-parser-output .navbar a>abbr{text-decoration:inherit}.mw-parser-output .navbar-mini abbr{font-variant:small-caps;border-bottom:none;text-decoration:none;cursor:inherit}.mw-parser-output .navbar-ct-full{font-size:114%;margin:0 7em}.mw-parser-output .navbar-ct-mini{font-size:114%;margin:0 4em}vteParsing algorithmsTop-down
Earley
LL
Recursive descent
Tail recursive
Bottom-up
Precedence
Simple
Operator
Shunting-yard
LR
Simple
Look-ahead
Canonical
Generalized
CYK
Recursive ascent
Shift-reduce
Mixed, other
Combinator
Chart
Left corner
Statistical
Related topics
PEG
Definite clause grammar
Deterministic parsing
Dynamic programming
Memoization
Parser generator
LALR
Parse tree
AST
Scannerless parsing
History of compiler construction
Comparison of parser generators
Operator-precedence grammar
vteWell-known computer science algorithmsCategories
Minimax
Sorting
Search
Streaming
Paradigms
Backtracking
Brute-force search
Divide and conquer
Dynamic programming
Greedy
Sweep line
Recursion
Other
Binary search
Breadth-first search
Depth-first search
Topological sorting
List of algorithms
<!--
NewPP limit report
Parsed by mw1385
Cached time: 20230523214537
Cache expiry: 1814400
Reduced expiry: false
Complications: [vary‐revision‐sha1, show‐toc]
CPU time usage: 0.228 seconds
Real time usage: 0.297 seconds
Preprocessor visited node count: 950/1000000
Post‐expand include size: 29486/2097152 bytes
Template argument size: 986/2097152 bytes
Highest expansion depth: 12/100
Expensive parser function count: 2/500
Unstrip recursion depth: 1/20
Unstrip post‐expand size: 32242/5000000 bytes
Lua time usage: 0.135/10.000 seconds
Lua memory usage: 4661287/52428800 bytes
Number of Wikibase entities loaded: 0/400
-->
<!--
Transclusion expansion time report (%,ms,calls,template)
100.00% 255.239 1 -total
33.85% 86.389 2 Template:Reflist
27.75% 70.832 2 Template:Navbox
25.23% 64.402 1 Template:Parsers
22.08% 56.351 1 Template:Cite\_web
18.06% 46.091 1 Template:Short\_description
10.09% 25.760 2 Template:Pagetype
6.89% 17.580 4 Template:Cite\_book
6.41% 16.350 1 Template:For
5.11% 13.047 1 Template:Use\_dmy\_dates
-->
<!-- Saved in parser cache with key enwiki:pcache:idhash:238867-0!canonical and timestamp 20230523214537 and revision id 1143884685. Rendering was triggered because: page-view
-->

References:

Similar Topics:

Image Links: