




WIKIPEDIA
The Free Encyclopedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

Print/export
Create a book
Download as PDF
Printable version


Languages 
Čeština
Deutsch
Español
فارسی
Français
한국어
Italiano
עברית
Nederlands
日本語
Polski
Português
Română
Русский
Српски / srpski
Українська
Tiếng Việt
中文

 Edit links

Create account Log in

Article **Talk**

Read Edit \ More

Search 

Backtracking

From Wikipedia, the free encyclopedia

*For the **line search** algorithm used in **unconstrained optimization**, see **Backtracking line search**.*

Backtracking is a general **algorithm** for finding all (or some) solutions to some **computational problems**, notably **constraint satisfaction problems**, that incrementally builds candidates to the solutions, and abandons each partial candidate *c* ("backtracks") as soon as it determines that *c* cannot possibly be completed to a valid solution.^{[1][2][3]}

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, since it cannot possibly be completed to a valid solution.

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 a large number of candidates with a single test.

Backtracking is an important tool for solving **constraint satisfaction problems**, such as **crosswords**, **verbal arithmetic**, **Sudoku**, and many other puzzles. It is often the most convenient (if not the most efficient^[*citation needed*]) technique for **parsing**, 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.

Contents [hide]

- 1 Description of the method
 - 1.1 Pseudocode
 - 1.2 Usage considerations
 - 1.3 Early stopping variants
- 2 Examples
 - 2.1 Constraint satisfaction
- 3 See also
- 4 References
- 5 Further reading
- 6 External links

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*.

Graph and tree search algorithms

α-β · A* · B* · **Backtracking** · Beam · Bellman–Ford · Best-first · Bidirectional · Borůvka · Branch & bound · BFS · British Museum · D* · DFS · Depth-limited · Dijkstra · Edmonds · Floyd–Warshall · Fringe search · Hill climbing · IDA* · Iterative deepening · Johnson · Jump point · Kruskal · Lexicographic BFS · Prim · SMA*

Listings

Graph algorithms · Search algorithms · List of graph algorithms

Related topics

Dynamic programming · Graph traversal · Tree traversal · Search games

v · t · e

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:

1. *root*(P): return the partial candidate at the root of the search tree.
2. *reject*(P, c): return *true* only if the partial candidate c is not worth completing.
3. *accept*(P, c): return *true* if c is a solution of P , and *false* otherwise.
4. *first*(P, c): generate the first extension of candidate c .
5. *next*(P, s): generate the next alternative extension of a candidate, after the extension s .
6. *output*(P, c): use the solution c of P , as appropriate to the application.

The backtracking algorithm reduces the problem to the call $bt(\text{root}(P))$, where *bt* is the following recursive procedure:

```
procedure bt(c)
  if reject(P, c) then return
  if accept(P, c) then output(P, c)
  s ← first(P, c)
  while s ≠ Λ do
    bt(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 *bt* 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.

Note that 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, denoted here by 'Λ', 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 is easily 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 [\[edit\]](#)

Typical examples are

- **Puzzles** such as [eight queens puzzle](#), [crosswords](#), [verbal arithmetic](#), [Sudoku](#), [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.

Below is an example for the [constraint satisfaction problem](#):

Constraint satisfaction [\[edit\]](#)

The general [constraint satisfaction problem](#) consists in finding a list of integers $x = (x[1], x[2], \dots, x[n])$, each in some range $\{1, 2, \dots, m\}$, 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 $c = (c[1], c[2], \dots, c[k])$, for any *k* between 0 and *n*, that are to be assigned to the first *k* variables $x[1], x[2], \dots, x[k]$. The root candidate would then be the empty list (). The *first* and *next* procedures would then be

```
function first(P,c)
  k ← length(c)
  if k = n
    then return Δ
    else return (c[1], c[2], ..., c[k], 1)

function next(P,s)
  k ← length(s)
  if s[k] = m
    then return Δ
    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 m^{n-k} *n*-tuples.

For example, if *F* is the [conjunction](#) of several boolean predicates, $F = F[1] \wedge F[2] \wedge \dots \wedge F[p]$, and each *F*[*i*] depends only on a small subset of the variables $x[1], \dots, x[n]$, then the *reject* procedure could simply check the terms *F*[*i*] that depend only on variables $x[1], \dots, x[k]$, 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 $x[1], \dots, x[k-1]$ 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

5	3	1	2	7	6	8	9	4
6	2	4	1	9	5	2		
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Sudoku puzzle solved by backtracking.



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](#)
- [Recursion \(computer science\)](#)

References [edit]

- ↑ Donald E. Knuth (1968). *The Art of Computer Programming*. Addison-Wesley.
- ↑ Thomas H. Cormen; Charles E. Leiserson; Ronald R. Rivest; Cliff Stein (1990). *Introduction to Algorithms*. McGraw-Hill.
- ↑ Gurari, Eitan (1999). [Backtracking algorithms "CIS 680: DATA STRUCTURES: Chapter 19: Backtracking Algorithms"](#) [↗].
- ↑ Rossi, Francesca; Beek, Peter Van; Walsh, Toby (August 2006). ["Constraint Satisfaction: An Emerging Paradigm"](#) [↗]. *Handbook of Constraint Programming* [↗]. *Foundations of Artificial Intelligence* [↗]. Amsterdam: Elsevier. p. 14. ISBN 978-0-444-52726-4. Retrieved 2008-12-30. "Bitner and Reingold credit Lehmer with first using the term 'backtrack' in the 1950s."

Further reading [edit]

- Gilles Brassard, Paul Bratley (1995). *Fundamentals of Algorithmics*. Prentice-Hall.

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
- [Sample Java Code](#) [↗], Sample code for backtracking of 8 Queens problem.

Categories: [Operations research](#) | [Search algorithms](#) | [Pattern matching](#)

This page was last modified on 25 June 2015, at 03:27.

Text is available under the [Creative Commons Attribution-ShareAlike License](#); additional terms may apply. By using this site, you agree to the [Terms of Use](#) and [Privacy Policy](#). Wikipedia® is a registered trademark of the [Wikimedia Foundation, Inc.](#), a non-profit organization.

[Privacy policy](#) [About Wikipedia](#) [Disclaimers](#) [Contact Wikipedia](#) [Developers](#) [Mobile view](#)

