

# Notes on Basic Search <sup>1</sup>

## Basic Search Methods

Algorithm 1 is the basic (template) pseudocode for the different basic search methods. For a partial path  $N$ ,  $\text{tail}(N)$  is the last element in the list representation of the partial path, which corresponds to the last node in the path that starts at  $S$ .<sup>2</sup>

---

**Algorithm 1** Basic Search

---

```
1: Initialize  $Q = ((S))$ 
2: while  $Q$  is not empty do
3:   Pick some path  $N$  from  $Q$ 
4:   if  $\text{tail}(N)$  is the goal node then
5:     return  $N$ 
6:   end if
7:   remove  $N$  from  $Q$ 
8:   for all children  $c$  of  $\text{tail}(N)$  in the search tree do
9:     extend the partial path  $N$  to  $c$ 
10:  end for
11:  Add extensions of  $N$  somewhere in  $Q$ 
12: end while
13: Signal failure
```

---

For the most part, what changes between the different basic search methods is how steps 3 and 11 of the algorithm are implemented; that is,

1. *which partial path  $N$*  to pick from  $Q$ , and
2. *where to insert* the extended paths from  $N$  in  $Q$ .

All basic search methods, except for some so called informed/heuristic search methods (like best-first search and beam search), pick the first partial path in  $Q$ . As we will see, the most variation is in where the extended paths are inserted (and in some cases, also how).

If, in addition, we use an *extended list*  $E$ , then the only difference is that (1) we add  $\text{tail}(N)$  to  $E$  after  $N$  is extended; and (2) after we remove  $N$  from  $Q$ , we first check whether  $\text{tail}(N)$  is in  $E$ . If  $\text{tail}(N)$  is in  $E$ , then we continue at the top of the while loop without extending  $N$ . Otherwise, we continue with the remaining steps of the while loop (extending  $N$  and adding extensions to  $Q$ ).

To implement a search method without *backtracking/backups* (*BT*), instead of adding all the extensions of  $N$  to  $Q$ , only *one* is added.

The difference between *informed or heuristic* and *uninformed or blind* search is that in informed search, for each node  $n$ , we assign to  $n$  a heuristic value that is an estimate of the minimum number

---

<sup>1</sup>These notes are primarily based on the recitation notes of Kimberle Koile and the slides of Tomás Lozano-Perez from previous terms, as well as the books *Artificial Intelligence (Third Edition)* by Patrick Henry Winston and *Artificial Intelligence: A Modern Approach (Second Edition)* by Stuart Russell and Peter Norvig. Original date: September 24, 2004; Last updated: November 6, 2006.

<sup>2</sup>Note that we could as well have used a different representation, and usually practically more useful, where we represent the path as an inverted list where the head of the list is the last node in the path and  $S$  is the tail of the list.

of steps to the goal from  $n$ , and use those heuristic values to decide the order in which *those* extensions of  $n$  should themselves be extended (*i.e.*, among *only* the set of extensions of  $n$ , which should be extended first). In general, we want to pick paths whose tail (*i.e.*, last node in the path) has low heuristic value.

*Depth-first search (DFS)* and *breadth-first search (BFS)* are uninformed search methods. DFS inserts the extended paths to the *front* of  $Q$ , while BFS inserts them to the *end* of  $Q$ .

*Hill climbing search (HC)*, *best-first search (BestFS)*, and *beam search (Beam)* are informed search methods. HC sorts the extensions of  $N$  by the heuristic values of their last node in the path (*i.e.*, the tail of the path) before inserting them to the *front* of  $Q$ . BestFS and Beam work a little different. BestFS picks the partial path  $N$  in  $Q$  whose tail has the *best heuristic value* and inserts the extensions of  $N$  *anywhere* in  $Q$ .

Beam has a width parameter  $K$  which determines how many partial paths it will extend at each level. Beam without BT, the typical implementation, picks *all* ( $K$ ) partial paths in  $Q$ , extends them all, then selects the best  $K$  extensions in terms of the heuristic value of the tail of the path and *replaces*  $Q$  with those  $K$  partial path extensions. Beam with BT picks the first  $K$  elements in  $Q$ , extends them all, then sort the extensions in terms of the heuristic value of the tail of the path and inserts *all* the sorted extensions to the front of  $Q$ . The implementation of Beam with BT is naturally more elaborate and requires more space than the typical implementation since it needs to keep track, at each level, of those paths that were ignored earlier on at previous depths (*i.e.*, higher up in the search tree).

The following table summarizes the differences in the implementation of the basic search methods using the template pseudocode given in Algorithm 1.

Method	Step 3	Step 11
Blind		
DFS	first	front
DFS (no BT)	first	replace w/ one
BFS	first	end
Heuristic		
HC	first	sorted extensions to front
HC (no BT)	first	replace w/ best
BestFS	best	anywhere
Beam( $K$ )	first $K$	sorted extensions to front
Beam( $K$ ) (no BT)	all	replace w/ best $K$

## Computation and Quality Guarantees

Note also that only BFS is guaranteed to find a path to the goal with the minimum number of nodes. BFS is also always guaranteed to find a path to the goal (*i.e.*, it is complete), even if the search tree is infinite. For search trees of finite depth, DFS and HC *with BT*, as well as BestFS, are also guaranteed to find a path to the goal.

## Time and Space Complexity

The basic search methods also differ in *worst-case running time and space*. Those values are given in terms of the *branching factor* ( $b$ ) and *depth* ( $d$ ) of the corresponding search tree. From a worst-

case complexity standpoint, the branching factor is *usually* maximum number of descendants of any node in the search tree, although the “average” branching size of all the nodes in the search tree is also used. The branching factor is assumed constant for the purpose of the worst-case analysis. The *total number of extended nodes* throughout the execution of the algorithm (roughly) characterizes the *worst-case running time*. Similarly, the *maximum size of  $Q$*  (*i.e.*, the number of elements or partial paths) at any time during the execution of the algorithm (roughly) characterizes the *space complexity*.

The following table summarizes both the guarantees and complexity properties of the basic search methods. The entries are illustrative rather than exact, since the main objective is to distinguish between those methods that exhibit *exponential explosion* and those that do not. Therefore, the worst-case time and space are given in terms of the number of extended nodes and the maximum size of  $Q$ , respectively.<sup>3</sup> The bounds given ignore constant<sup>4</sup> and  $\log$ <sup>5</sup> factors, and assume that the corresponding search tree is “regular” in the sense that every node has the same number of children  $b$  (*i.e.*, finite branching factor) and every path from the starting node to the leaves has the same finite length  $d$  (*i.e.*, finite depth).<sup>6</sup> The worst-case occurs when the goal is lowest (*i.e.*, depth  $d$ ) and most to the right in the search tree. Finally, note that if there exists a path of infinite length in the search tree, then only BFS and BestFS can guarantee to output a path to the goal if one exists.

Search Method	Guarantee Path	Min Length	Worst Time	Worst Space
Blind				
DFS	yes	no	$b^d$	$bd$
DFS (no BT)	no	no	$d$	1
BFS	yes	yes	$b^d$	$b^d$
Heuristic				
HC	yes	no	same as DFS	
HC (no BT)	no	no	same as DFS (no BT)	
BestFS	yes	no	same as BFS	
Beam( $K$ )	yes	no	$b^d$	$bdK$
Beam( $K$ ) (no BT)	no	no	$dK$	$K$

<sup>3</sup>This ignores, for instance, that it takes time and space to extend partial paths and that a partial path in the queue takes space proportional to its length.

<sup>4</sup>They are in so called “big- $O$ ” notation without the  $O$ .

<sup>5</sup>Assumes that sorting takes constant time, which is false.

<sup>6</sup>This is how the worst-case running time and space bounds given in the table in the text change if we use a formal analysis and remove the assumptions, except for the one on  $b$ . Let  $m$  be the length of the longest path to *any* node in the search tree and  $d$  the shortest path to the goal in the search tree. Assume  $m$  is infinite if there is a path of infinite length in the search tree and  $d$  is infinite if the goal is not in the search tree. DFS with BT takes  $O(b^m)$  time and  $O(m^2b)$  space, while without BT it takes  $O(m)$  time and  $O(m)$  space. BFS takes  $O(b^{d+1})$  time and  $O(b^d)$  space, but it takes  $O(b^m)$  time and  $O(b^m)$  space if  $d$  is infinite. HC with BT takes  $O(b^m \log(b))$  time and  $O(m^2b)$  space, while without BT it takes  $O(mb \log(b))$  time and  $O(mb)$ . BestFS takes  $O(b^d)$  time and  $O(b^d d)$  space, but if  $d$  is infinite, then it takes  $O(b^m)$  time and  $O(mb^m)$  space. Beam( $K$ ) with BT takes  $O(b^m \log(b))$  time and  $O(mKb)$  space, while without BT it takes  $O(mKb \log(b))$  time and  $O(mKb)$  space. Note that if  $b$  is infinite, then both the time and space of all the algorithms are infinite, except DFS without BT when  $d$  is finite.