

Lecture 02: Blind Search

TBA

Contents

1	Overview	1
2	Basic Search Problem	2
3	Evaluation Criteria	2
4	Search Algorithms	3
4.1	Generic Search	3
4.2	Breadth-First Search	3
4.3	Depth-First Search	4
4.4	Iterative Deepening	5
4.5	Iterative Broadening	6
4.6	Bidirectional Search	7
4.7	Summary	8
5	Example	8

1 Overview

Our first series of lectures is concerned with the following topics in the core area of search:

1. blind, or exhaustive, search: depth-first, breadth-first, iterative deepening, iterative broadening, and bidirectional search
2. heuristic, or informed, search, in which the search is guided by a domain-specific heuristic evaluation function: greedy search, beam search, A* and IDA* search
3. local search, in which search proceeds by looking in neighborhoods of the current best goal for an improvement: hill climbing (gradient descent), simulated annealing, genetic algorithms
4. constraint satisfaction problems, like satisfiability, in which all goals that satisfy the given constraints are equally valid: backtracking, constraint propagation
5. constrained optimization, in which there are constraints to be satisfied and an objective function to be optimized: mathematical programming
6. adversarial search, in which the objective is to discover a winning strategy in a game: minimax algorithm, $\alpha\beta$ pruning

2 Basic Search Problem

A **basic search problem** is a 4-tuple $\langle X, S, G, \delta \rangle$, where

- X is a set of states
- $S \subseteq X$ is a nonempty set of **start** states
- $G \subseteq X$ is a nonempty set of **goal** states
- $\delta : X \rightarrow 2^X$ is a state transition function
 $\delta(x)$ is the set of successor states of x

Example: As an example, consider the sliding tiles puzzle, where the objective is to slide numbered tiles into numerical order, given an $n \times m$ board with $n \times m - 1$ tiles and one blank space. The states X describe the locations of the numbers and the blank space. The set of start states $S = X$. The set of goal states $G \subseteq X$ includes only that state in which the tiles are in row-major order: i.e, 1 to n in the first row, $n + 1$ to $2n$ in the second row, \dots , and finally, $n(m - 1) + 1$ to $nm - 1$ in the last row, with the blank square in the bottom-right-most corner. The transition function describes the change in state that results from moving the blank space left, right, up, or down. (See Figure 1).

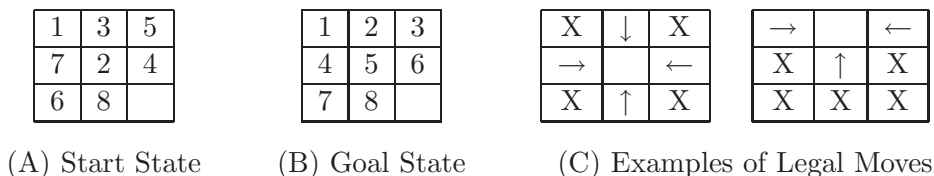


Figure 1: Sliding Tiles Puzzle ($n = m = 3$). (A) Start State. (B) Goal State. (C) Examples of Legal moves: An arrow indicates the direction in which a tile can be moved. An “X” means that a tile cannot be moved.

Exercise: Formalize the following version of the **All the King’s Digits** puzzle as a basic search problem: Given the sequence of digits 0123456789, insert the symbols $(,), +, -, \cdot, /$ between the digits such that the resulting expression evaluates to 100.

One possible solution to this puzzle is:

$$0 + 1 + 2 + 3 + 4 + 5 + 6 + 7 + (8 \cdot 9) = 100$$

Can you find others?

3 Evaluation Criteria

Before discussing a variety of algorithms designed to solve search problems, it is worth mentioning how we will evaluate such algorithms. The following criteria are typically used for this evaluation:

- time complexity: how much time is required to find solution?

- space complexity: how much space is required to find solution?
- completeness: is the algorithm guaranteed to find solution, if one exists?
- optimality: does it find an optimal solution (e.g., a shortest path), if multiple solutions exists?

The analyses and algorithms presented in this lecture depend on the assumption that the search space is a **tree** of (possibly infinite) depth d with finite branching factor b . Each **node** in this tree represents a state in the search space. The **depth** of a tree is defined as the maximum number of links connecting the root node to any other node in the tree. The **branching factor** of a tree is defined as the maximum number of immediate successors (i.e., children) of any node in the tree.



In a tree, the immediate successors of a node (other than the leaves) are called its **children**; the (unique) immediate predecessor of a node (other than the root) is called its **parent**; and the nodes with whom it shares its parent are called its **siblings**.

4 Search Algorithms

In this section, we discuss five blind search algorithms (for trees). They are all instances of the generic search algorithm (for trees) which is described first.

4.1 Generic Search

During a search, the **fringe** (or **frontier**) is the collection of nodes waiting to be visited, usually stored as a stack, a queue, or a priority queue. Nodes on the fringe are called **open**. After nodes are deleted from the fringe, they are labeled **closed**.

Search spaces can be formulated as trees or graphs. Below, we formulate generic search algorithms for both trees and graphs; the difference between them is that the tree algorithm does not maintain a list of closed nodes because nodes in a tree are never reencountered during a search.

The remainder of the (blind search) algorithms in this lecture specialize the generic search algorithm for trees (Table 1). We defer a detailed discussion of graph search until the lectures on heuristic search (Lectures 03 and 04).

4.2 Breadth-First Search

The main idea of breadth-first search (BFS) is to visit all nodes at depth i before visiting those at depth $i + 1$: i.e., after visiting a node at the next level, visit its siblings before visiting its children. BFS is implemented by storing the set of open nodes as a **queue**, and accessing its entries in a first-in-first-out fashion.

BFS is complete: it is guaranteed to find a solution, if one exists. Moreover, BFS is optimal: it always finds a goal node of minimal distance from the start. That's the good news. The bad news

SEARCH(X, S, G, δ)	
Inputs	search problem
Output	(path to) goal node or failure
Initialize	$O = S$ is the set of open nodes
while (O is not empty) do	
1. delete some node $n \in O$	
2. if $n \in G$, return (path to) n	
3. insert $\delta(n)$ into O	
fail	

Table 1: Generic Search Algorithm for Trees.

SEARCH(X, S, G, δ)	
Inputs	search problem
Output	(path to) goal node or failure
Initialize	$O = S$ is the set of open nodes $C = \emptyset$ is the queue of closed nodes
while (O is not empty) do	
1. delete some node $n \in O$	
2. if $n \in G$, return (path to) n	
3. insert $\delta(n) \setminus C$ into O	
4. insert n into C	
fail	

Table 2: Generic Search Algorithm for Graphs.

is, in the worst case, BFS visits every node, which takes time as follows:

$$1 + b + b^2 + \dots + b^d = \sum_{i=0}^d b^i = \frac{b^{d+1} - 1}{b - 1} = O(b^d)^1$$

In terms of space, BFS maintains a list of all nodes at all depths: at depth d the length of this list is b^d . Hence, the space complexity of BFS is exponential in d . In summary, both the time and space complexities of BFS are exponential in d .

4.3 Depth-First Search

Like BFS, depth-first-search (DFS) can also be viewed as proceeding level by level; however, after visiting a node at the next level, DFS visits its children before visiting its siblings. DFS is imple-

¹Let $S = 1 + b + b^2 + \dots + b^d$. Then $bS = b + b^2 + \dots + b^d + b^{d+1}$. So $(b - 1)S = bS - S = b^{d+1} - 1$, from which it follows that $S = \frac{b^{d+1} - 1}{b - 1}$.

BFS(X, S, G, δ)	
Inputs	search problem
Output	(path to) goal node or failure
Initialize	$O = S$ is the queue of open nodes
while (O is not empty) do	
1. delete <i>first</i> node $n \in O$	
2. if $n \in G$, return (path to) n	
3. append $\delta(n)$ to <i>back</i> of O	
fail	

Table 3: Breadth-First Search.

mented by storing the set of open nodes as a **stack**, and accessing its entries in a last-in-first-out fashion.

DFS(X, S, G, δ)	
Inputs	search problem
Output	(path to) goal node or failure
Initialize	$O = S$ is the stack of open nodes
while (O is not empty) do	
1. delete <i>first</i> node $n \in O$	
2. if $n \in G$, return (path to) n	
3. prepend $\delta(n)$ to <i>front</i> of O	
fail	

Table 4: Depth-First Search.

Like BFS, the time complexity of DFS is $O(b^d)$. It is exponential in d because in the worst-case DFS visits every node. The space complexity of DFS, however, is linear in d , where d is the length of longest path. Since at most b nodes are stored at each of the d depths, the space complexity of DFS is $O(bd)$.

DFS is neither complete nor optimal. Given knowledge base $\{\phi \rightarrow \phi, \phi\}$ and formula ϕ , DFS could fail to prove ϕ by forever visiting $\phi \rightarrow \phi$. Using DFS, a robot that intends to visit its neighbor to the west, but starts out searching to the east, travels all the way around the world before finding its neighbor!

4.4 Iterative Deepening

Iterative deepening (ID) is a search algorithm with the space requirements of DFS—it requires memory linear in d —and the performance properties of BFS—it is complete and (asymptotically) optimal. The main idea of iterative deepening is to repeatedly search in depth-first fashion, over

subgraphs of depth 0, depth 1, depth 2, and so on, until a goal is found.

ID(X, S, G, δ)	
Inputs	search problem
Output	(path to) goal node
Initialize	$c = 0$ is the cut-off depth $O = S$ is the stack of open nodes
while (1) do	
1. while (O is not empty) do	
(a) delete <i>first</i> node $n \in O$	
(b) if $n \in G$, return (path to) goal n	
(c) if $\text{depth}(n) \leq c$	
i. prepend $\delta(n)$ to <i>front</i> of O	
2. increment c , $O = S$	

Table 5: Iterative Deepening.

ID is optimal and complete: it is guaranteed to find a goal if one exists—specifically, an optimal goal if multiple solutions exist (as long as $c = 1$). It iteratively performs depth-first searches; thus, its space complexity is that of DFS, namely $O(bd)$.

Like DFS and BFS, the time complexity of ID is $O(b^d)$: *i.e.*, in the worst-case it is exponential in d . ID visits nodes at depth 0 $d + 1$ times, at depth 1 d times, ..., and at depth d 1 time. Thus, the total time required is given by:

$$1 + \underbrace{1 + b}_{\text{depth 1}} + \underbrace{1 + b + b^2}_{\text{depth 2}} + \dots + \underbrace{1 + b + b^2 + \dots + b^d}_{\text{depth } d} = O(1 + b + b^2 + \dots + b^d) = O(b^d)$$

4.5 Iterative Broadening

The idea of iterative broadening (IB) is analogous to that of iterative deepening, but IB iteratively broadens its search, whereas ID iteratively deepens it. IB performs depth-first searches on subgraphs of breadth 1, breadth 2, breadth 3, and so on, until a goal node is reached.

IB is not complete: it is susceptible to following infinite paths within its breadth limit as it conducts depth-first search. Neither is IB optimal.

In the worst-case, its space complexity equals that of DFS, and the time it requires is $O(b^d)$. Specifically, the number of nodes IB visits is given by:

$$d + \underbrace{1 + 2 + \dots + 2^d}_{\text{breadth 2}} + \dots + \underbrace{1 + b + \dots + b^d}_{\text{breadth } b} = O(1^d + 2^d + \dots + b^d) = O((b + 1)^{d+1})$$

since

$$\frac{b^{d+1}}{d+1} = \int_0^b x^d dx \leq \sum_{c=1}^b c^d \leq \int_1^{b+1} x^d dx = \frac{(b+1)^{d+1} - 1}{d+1}$$

IB(X, S, G, δ)	
Inputs	search problem
Output	(path to) goal node
Initialize	$c = 1$ is the cut-off breadth $O = S$ is the stack of open nodes
while (1) do	
1. while (O is not empty) do	
(a) delete <i>first</i> node $n \in O$	
(b) if $n \in G$, return (path to) goal n	
(c) if $\text{breadth}(n) \leq c$	
i. prepend c nodes in $\delta(n)$ to <i>front</i> of O	
2. increment c , $O = S$	

Table 6: Iterative Broadening.

4.6 Bidirectional Search

A **bidirectional search problem** is a 5-tuple $\langle X, S, G, \delta, \gamma \rangle$, where $\langle X, S, G, \delta \rangle$ is a basic search problem and $\gamma : X \rightarrow 2^X$ is an “inverse” transition function, meaning $\gamma(x)$ is the set of predecessors of x . In bidirectional search, as the name suggests, search proceeds simultaneously in two directions: both forwards from an initial state and backwards from a goal state. Search terminates where and when the two searches meet. However, not all basic search problems are easily posed as bidirectional search problems: e.g., it is nontrivial to generate predecessors of the set of checkmate configurations in the game of chess.

Bidirectional search is typically implemented as two breadth-first searches. Like BFS, bidirectional BFS is optimal and complete. Bidirectional search improves upon BFS in terms of complexity, however. The time complexity of bidirectional BFS is exponential in $d/2$. In the worst case, each direction of bidirectional BFS visits every node through depth $d/2$, which for two directions requires $O(2b^{d/2}) = O(b^{d/2})$. Its space complexity is also $O(b^{d/2})$, since it maintains list of all nodes all at depths i , and the number of nodes at depth $d/2 = b^{d/2}$.

Island-driven search is a generalization of bidirectional search in which a series of intermediate goals, or islands, is identified between the start and goal nodes. The islands reduce the one large search problem to m small search problems. If the islands are spaced evenly between the start and goal nodes, the number of nodes visited is $mb^{d/m} \ll b^d$. An obvious difficulty with this approach is the identification of islands through which an optimal search path traverses. Consequently, it is difficult to guarantee optimality.

4.7 Summary

Criteria	DFS	BFS	ID	IB	biBFS
Time	$O(b^d)$	$O(b^d)$	$O(b^d)$	$O((b + 1)^{(d+1)})$	$O(b^{d/2})$
Space	$O(bd)$	$O(b^d)$	$O(bd)$	$O(bd)$	$O(b^{d/2})$
Completeness	NO	YES	YES	NO	YES
Optimality	NO	YES	YES	NO	YES

1. IB is preferred if the branching factor is large: the search space is wide (possibly infinite), particularly if goals are known to be deep
2. ID is preferred if the depth is large: the search space is deep (possibly infinite), particularly if goals are known to be shallow
3. DFS is preferred if the maximum depth of the goal nodes is known: if this depth is n , we can modify DFS to search only to depth n

5 Example

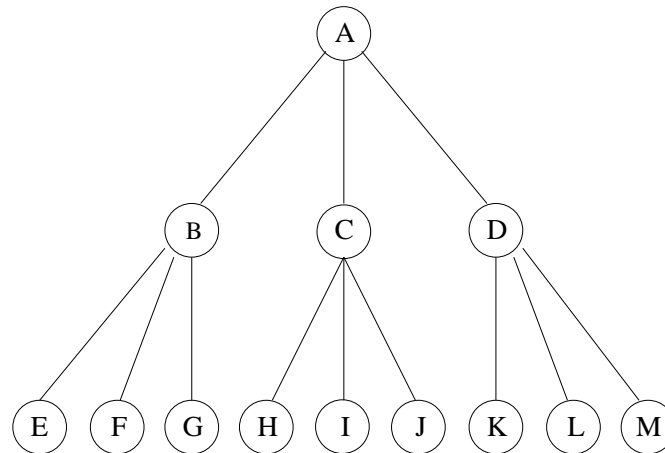


Figure 2: (a) BFS: ABCDEFGHIJKLM. (b) DFS: ABEFGCHIJDKLM. (c) ID: AABCD-ABEFGCHIJDKLM. (d) IB: ABEABEFCHIABEFGCHIJDKLM.