# Chapter 3: Searching for Solutions

This chapter casts the problem of an agent deciding how to solve a goal as the problem of searching to find a path in a graph.

## 1 Problem Solving as Search

Searching in this chapter means searching in an internal representation for a path to a goal.

When an agent is given a problem, it is usually given only a description that lets it recognize a solution, not an algorithm to solve it. The existence of NP-complete problems, with efficient means to recognize solutions but no efficient methods for finding them, indicates that searching is a necessary part of solving problems.

Searching by humans:

- they use intuition to jump to solutions of difficult problems,
- they solve specific instances rather than general problems,
- they often look for satisficing solutions rather than optimal ones, and
- they find problems with little structure, or ones in which structure cannot be related to the physical world, difficult to solve.

The difficulty of search and the fact that humans are able to solve some search problems efficiently suggests that computer agents should exploit knowledge about special cases to guide them to a solution. This extra knowledge beyond the search space is called **heuristic knowledge**.

## 2 State Spaces

*Dimensions*: flat, states, indefinite horizon, fully observable, deterministic, goal directed, non-learning, single agent, offline, perfect rationality

A **state** contains all of the information necessary to predict the effects of an action and to determine whether a state satisfies the goal. State-space searching assumes:

- The agent has perfect knowledge of the state space and is planning for the case where it observes what state it is in: there is full observability.
- The agent has a set of actions that have known deterministic effects.
- The agent can determine whether a state satisfies the goal.

A **solution** is a sequence of actions that will get the agent from its current state to a state that satisfies the goal. [Another term is *policy*.]

A state-space problem consists of

- a set of states
- a distinguished state called the start state
- for each state, a set of actions available to the agent in that state
- an action function that, given a state and an action, returns a new state
- a goal specified as a Boolean function, goal(s), that is true when state s satisfies the goal, in which case we say that s is a goal state
- a criterion that specifies the quality of an acceptable solution

# 3 Graph Searching

In representing a state-space problem, the states are represented as nodes, and the actions as arcs (though there may be other ways of representing a problem as a graph).

## 3.1 Formalizing Graph Searching

A **directed graph** consists of

- a set $N$ of nodes and
- a set $A$ of arcs, where an arc is an ordered pair of nodes

Notation and terminology:
$\langle n_1, n_2 \rangle$: Arc outgoing from $n_1$ to $n_2$.
$n_2$ is a **neighbor** of $n_1$ if there is an arc from the latter to the former, i.e., $\langle n_1, n_2 \rangle \in \mathcal{A}$
Arcs may be **labeled** with such things as their weight, cost, etc.
**Path** from node $s$ to $g$ is a sequence of nodes $\langle n_0, n_1, \ldots, n_k \rangle$ such that $s = n_0$ and $g = n_k$, and $\langle n_{i-1}, n_i \rangle \in \mathcal{A}$.
Path $\langle n_0, n_1, \ldots, n_i \rangle$ is **initial path** of $\langle n_0, n_1, \ldots, n_k \rangle$ if $i \leq k$.
A **goal** is a Boolean function on nodes. $n$ is **goal node** if goal(n) is true.
A **cycle** is a nonempty path where the end node is the same as the start node: $\langle n_0, n_1, \ldots, n_k \rangle$ such that $n_0 = n_k$. A directed graph without any cycles is called a **directed acyclic graph (DAG)**.
A **tree** is a DAG where there is one node with no incoming arcs and every other node has exactly one incoming arc, called **root**. A **leaf** is a node with no outgoing arcs. Neighbors are often called **children**.

**Forward branching factor**: number of outgoing arcs from a node. **Backward branching factor**: number of incoming arcs to a node. Branching factors provide measures for complexity measurement of graph algorithms. When we discuss the time and space complexity of the search algorithms, we assume that the branching factors are bounded. The branching factor is an important key component in the size of the graph. If the forward branching factor for each node is **b**, and the graph is a tree, there are $b^n$ nodes that are $n$ arcs away from the start node.

# 4 A Generic Searching Algorithm

The intuitive idea behind the generic search algorithm, given a graph, a start node, and a goal predicate, is to explore paths incrementally from the start node. This is done by maintaining a frontier (or fringe) of paths from the start node. The frontier contains all of the paths that could form initial segments of paths from the start node to a goal node.

**Algorithm 1:** Search: generic graph searching algorithm

```
   Input:
 1 G: graph with nodes N and arcs A
 2 s: start node
 3 goal: Boolean function of nodes
   Output:
 4 path from s to a node for which goal is true
 5 or ⊥ if there are no solution paths
 6 Local Frontier: set of paths
 7 Frontier := {⟨s⟩}
 8 while Frontier ≠ {} do
 9 |   select and remove ⟨n_0, ..., n_k⟩ from Frontier;
10 |   if goal(n_k) then
11 |   |   return ⟨n_0, ..., n_k⟩
12 |   end
13 |   Frontier := Frontier ⋃{⟨n_0, ..., n_k, n⟩ : ⟨n_k, n⟩ ∈ A} return ⊥
14 end
```

The following are some features of algorithm 1 has:
• The path selected in line 9 defines the search strategy.
• If the procedure returns $\bot$ ("bottom") there are no solutions.
• The algorithm tests for goal after selecting the path from the frontier and not when it is added to the frontier, because: (a) the search should look for a lower-cost arc from a frontier node to the goal, and (b) determination of whether the node is a goal can be expensive and can be delayed if it is not necessary.

# 5 Uninformed Search Strategies

A problem determines the graph, the start node and the goal but not which path to select from the frontier. This is the job of a search strategy. A **search strategy** defines the order in which paths are selected from the frontier (line 9 in algorithm 1).

**Uninformed search strategies** do not take into account the location of the goal. Intuitively, these algorithms ignore where they are going until they find a goal and report success.

## 5.1 Breadth-First Search

In breadth-first search the frontier is implemented as a FIFO (first-in, first-out) queue. Thus, the path that is selected from the frontier is the one that was added earliest.

In breadth-first search each path on the frontier has either the same number of arcs or one more arc than the next element of the frontier that will be selected.

Suppose the branching factor of the search is $b$. If the next path to be selected on the frontier contains $n$ arcs, there are at least $b^{n+1}$ elements of the frontier. All of these paths contain $n$ or *n+1* arcs. Thus, both space and time complexities are exponential in the number of arcs of the path to a goal with the fewest arcs. This method is guaranteed, however, to find a solution if one exists and will find a solution with the fewest arcs.

Breadth-first search is useful when
• the problem is small enough so that space is not a problem (e.g., if you already need to store the graph) and
• you want a solution containing the fewest arcs.

It is a poor method when all solutions have many arcs or there is some heuristic knowledge available. It is not used very often for large problems where the graph is dynamically generated because of its exponential space complexity.

## 5.2 Depth-First Search

In depth-first search, the frontier acts like a LIFO (last-in, first-out) stack of paths. In a stack, elements are added and removed from the top of the stack. Using a stack means that the path selected and removed from the frontier at any time is the last path that was added.

This algorithm does not specify the order in which the paths to the neighbors are added to the frontier and it does not depend on the goal. The efficiency of the algorithm is sensitive to this ordering.

Implementing the frontier as a stack results in paths being pursued in a depth-first manner – searching one path to its completion before trying an alternative path. This method is said to involve **backtracking**: the algorithm selects a first alternative at each node, and it backtracks to the next alternative when it has pursued all of the paths from the first selection. Some paths may be infinite when the graph has cycles or infinitely many nodes, in which case a depth-first search may never stop.

If the branching factor is $b$ and the selected path on the frontier has $k$ arcs, there can be at most $k * (b - 1)$ other paths on the frontier. These are the up to *(b-1)* alternative paths from each node. Therefore, for depth-first search, the space used at any stage is linear in the number of arcs from the start to the current node.

Depth-first search is appropriate when
• space is restricted

• many solutions exist, perhaps with long paths, particularly for the case where nearly all paths lead to a solution or
• the order in which the neighbors of a node are added to the stack can be tuned so that solutions are found on the first try.

It is a poor method when
• it is possible to get caught in infinite paths, which occurs when the graph is infinite or when there are cycles in the graph
• solutions exist at shallow depth, because in this case the search may explore many long paths before finding the short solutions, or
• there are multiple paths to a node, for example, on a $n \times n$ grid, where all arcs go right or down, there are exponentially paths from the top-left node, but only $n^2$ nodes.

### 5.3 Iterative Deepening

Iterative Deepening combines the space efficiency of depth-first search with the optimality of breadth-first search by recomputing the elements of the breadth-first frontier rather than storing them. Each recomputation can be a depth-first search, which thus uses less space.

Iterative deepening repeatedly calls a depth-bounded searcher, a depth-first searcher that takes in an integer depth bound and never explores paths with more arcs than this depth bound. First the bound is 1, then 2, and so on. When a search with depth-bound $n$ fails to find a solution, it can throw away all of the previous computation and start again with a depth-bound of $n+1$. Eventually, it will find a solution if one exists, and, as it is enumerating paths in order of the number of arcs, a path with the fewest arcs will always be found first.

To ensure this method stops for finite graphs, it distinguishes between:
• failure because the depth bound was reached, where search is retired with a larger depth bound, and
• failure due to exhausting the search space, the whole search is failed.

### 5.4 Lowest-Cost-First Search

The simplest search method that is guaranteed to find a minimum cost path is lowest-cost-first search, which is similar to breadth-first search, but instead of expanding a path with the fewest number of arcs, it selects a path with the lowest cost. This is implemented by treating the frontier as a priority queue ordered by the cost function.

The bounded arc cost is used to guarantee the lowest-cost search will find a solution, when one exists, in graphs with finite branching factor. Without such a bound there can be infinite paths with a finite cost. For example, there could be nodes $n_0, n_1, \ldots$ with an arc $\langle n_{i-1}, n_i \rangle : \forall i > 0$ with cost $\frac{1}{2^i}$. Infinitely many paths of the form $\langle n_0, n_1, n_2, \ldots, n_k \rangle$ all have a cost of less than 1. If there is an arc from $n_0$ to a goal node with a cost equal to 1, it will never be selected. This is the basis of Zeno's paradox that Aristotle wrote about more than 2300 years ago.

**Algorithm 2:** Search: generic graph searching algorithm

```
   Input:
 1 G: graph with nodes N and arcs A
 2 s: start node
 3 goal: Boolean function of nodes
   Output:
 4 path from s to a node for which goal is true
 5 or ⊥ if there are no solution paths
 6 Local hit_depth_bound: Boolean
 7 bound: integer
 8 begin
 9 begin
10 |   procedure: depth_bounded_search(⟨n_0, ..., n_k⟩, b)
   |   Input:
11 |   ⟨n_0, ..., n_k⟩ : path
12 |   b: integer, b ≥ 0
   |   Output:
13 |   path to goal of length k+b if one exists
14 end
```

Like breadth-first search, lowest-cost-first search is typically exponential in both space and time. It generates all paths from the start that have a cost less than the cost of a solution.

# 6 Heuristic Search

A heuristic function *h(n)*, takes a node $n$ and returns a non-negative real number that is an estimate of the cost of the least-cost path from node $n$ to a goal node. The function *h(n)* is an **admissible heuristic** if *h(n)* is always less than or equal to the actual cost of a lowest-cost path from node $n$ to a goal.

A simple use of a heuristic function in depth-first search is to order the neighbors that are added to the stack representing the frontier. The neighbors can be added to the frontier so that the best neighbor is selected first. This is known as **heuristic depth-first search**. This search selects the locally best path, but it explores all paths from the selected path before it selects another path. Although it is often used, it suffers from the problems of depth-first search, and is not guaranteed to find a solution and may not find an optimal solution.

Another way to use a heuristic function is to always select a path on the frontier with the lowest heuristic value. This is called **greedy best-first search**. This method sometimes works well. However, it can follow paths that look promising because they appear (according to the heuristic function) close to the goal, but the path explored may keep getting longer.

### 6.1 $A^*$ Search

$A^*$ search uses both path cost, as in lowest-cost-first, and heuristic information, as in greedy best-first search, in its selection of which path to expand. For each path on the frontier, $A^*$ uses an estimate of the total path cost from the start node to a goal node constrained to follow that path initially. It uses *cost(p)*, the cost of the path found, as well as the heuristic function *h(p)*, the estimated path cost from the end of $p$ to the goal.

For any path $p$ on the frontier, define $f(p) = cost(p) + h(p)$. This is an estimate of the total path cost to follow path p then go to a goal node.

$A^*$ is implemented using a generic search algorithm, treating the frontier as a priority queue ordered by *f(p)*.

**Admissible search algorithm** is one which returns an optimal solution whenever a solution exists. Admissibility is guaranteed whenever certain conditions on graph and the heuristic hold.

*Proposition*: ($A^*$ admissibility) If there is a solution, $A^*$ using heuristic function h always returns an optimal solution, if

- the branching factor is finite (each node has a bounded number of neighbors),
- all arc costs are greater than some $\epsilon > 0$, and
- $h$ is an admissible heuristic, which means that *h(n)* is less than or equal to the actual cost of the lowest-cost path from node n to a goal node.

For proof of this proposition, the second condition, $\epsilon > 0$ means costs are bounded above zero, and this is a sufficient condition for $A^*$ to avoid suffering from the Zeno's paradox.

Admissibility of $A^*$ means that the first path chosen is optimal, but it does not mean that every partial path is best during the course of search.

**Iterative Deepening** $A^*$ performs repeated depth-bounded depth-first searches with the bounds specified by $f(start\_node)$ rather than number of arcs.

**Pattern database** maps the nodes of a simple problem into the heuristic value.

## 7 Pruning the Search Space

### 7.1 Cycle Pruning

**Cycle pruning** or **loop pruning** checks whether the last node on the path already appears earlier on the path from the start node to that node.

The compuational complexity of cycle pruning depends on the search method:
• For depth-first search (DFS), overhead can be a constant factor by storing elements of the current

path as a set
• For search strategies which store multiple paths (those with exponential space) this takes linear time in the length of the path searched. These algorithms cannot do better than searching up the partial path being considered, checking to ensure they do not add a node that already appears in the path.

## 7.2 Multiple-Path Pruning

**Multiple-path pruning** is implemented by maintaining an **explored set** (traditionally called **closed list**) of nodes that are at the end of paths that have been expanded. The explored set is initially empty. When a new path is selected, if the newest node is already in the explored set, the path is discarded, else the newest node is added to the explored set.

This approach does not guarantee a least-cost path. To guarantee this, we can:
• Make sure the first path to any node is lowest-cost and prune all subsequent paths to that node
• If a lower-cost path is found than the one already found, then remove all higher-cost paths to that node
• If a lower-cost path is found than the one already found, then the initial sections of all paths to that node are updated.

A **consistent heuristic** is a non-negative function $h(n)$ on node $n$ that satisfies the constraint $h(n) \leq cost(n, n') + h(n')$ for any two nodes $n$' and $n$, where $cost(n, n')$ is the cost of the least-cost path from $n$ to $n$'. Note that if $h(g) = 0$ for any goal $g$, then a consistent heuristic is never an overestimate of the cost of going from a node $n$ to a goal.

Consistency can be guaranteed if the heuristic function satisfies the monotone restriction: $h(n) \leq cost(n, n') + h(n')$ for any arc $\langle n, n' \rangle$. It is easier to check the monotone restriction as it only depends on the arcs, whereas consistency depends on all pairs of nodes. With the monotone restriction, the

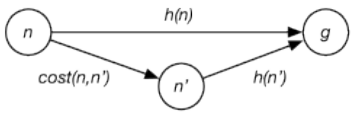


Figure 1: Triangle inequality: $h(n) \leq cost(n, n') + h(n')$

$f - values$ of the paths selected from the frontier are monotonically non-decreasing. That is, when the frontier is expanded, the $f - values$ do not get smaller.

*Proposition*: With a consistent heuristic, multiple-path pruning can never prevent $A^*$ search from finding an optimal solution.

Multiple-path pruning can be done in constant time, by setting a bit on each node to which a path has been found if the graph is explicitly stored, or using a hash function. This method is preferred for BFS because BFS stores all nodes are stored anyway, but cycle-pruning is used for DFS because using multple-path pruning for DFS makes it exponential in space. This method is also not appropriate for iterative deepening $A^*$, because of higher space requirement to store

explore set. $IDA^*$ is less effective for domains which have multiple paths to a node.

### 7.3 Summary of Search Strategies

| Strategy | Selection from frontier | Path found | Space |
|---|---|---|---|
| Breadth-first | First node added | Fewest arcs | Exponential |
| Depth-first | Last node added | No | Linear |
| Iterative deepening | – | Fewest arcs | Linear |
| Greedy best-first | Minimal *h(p)* | No | Exponential |
| Lowest-cost-first | Minimal *cost(p)* | Least cost | Exponential |
| $A^*$ | Minimal *cost(p)+h(p)* | Least cost | Exponential |
| $IDA^*$ | – | Least cost | Linear |

## 8 More Sophisticated Search

### 8.1 Branch and Bound

Depth-first branch-and-bound search combines the space saving of depth-first search with heuristic information for finding optimal paths. It is particularly applicable when there are many paths to a goal. Like $A^*$, the heuristic function is non-negative and less than equal to cost of lowest-cost path from $n$ to goal node.

The idea: maintain the lowest-cost path to a goal found so far, and its cost. Suppose this cost is bound. A new path $p$ is pruned when $cost(p) + h(p) \geq bound$. Then, a non-pruned path to a goal must be better than the previous best path. The new solution is remembered and bound is set to the cost of this new solution and search continues.

INSERT ALGORITHM HERE

This algorithm returns an optimal solution if there is a solution with cost less than the initial bound. It can be combined with iterative deepening to increase bound until either a solution is found or there is no solution.

Cycle pruning works well with depth-first branch-and-bound but multiple path pruning is not appropriate because storing the explored set defeats the space saving.

### 8.2 Direction of Search

The size of the search space of the generic search algorithm, for a given pruning strategy, depends on the path length and the branching factor.

If the following conditions hold:
• the set of goal nodes, $\{n : goal(n)\}$, is finite and can be generated • for any node $n$ the neighbors

of $n$ in the **inverse graph**, namely $\{n' : \langle n', n\rangle \in A\}$, can be generated
the graph search algorithm can either begin with the start node and search forward for a goal node, or begin with a goal node and search backward for the start node.

In **backward search**, the frontier start with the goal node and searches for the start node in the inverse graph, whereas in **forward search** it searches from start to the goal node in the original graph. A general principle is to search in the direction that has the smaller branching factor.

The idea of **bidirectional search** is to search forward from the start and backward from the goal simultaneously. It is a challenge to guarantee that the path found is optimal.

If the forward and backward branching factors of the search space are both $b$, and the goal is at depth $k$, then breadth-first search will take time proportional to $b^k$, whereas a symmetric bidirectional search will take time proportional to $2b^{k/2}$. This is an exponential saving in time, even though the time complexity is still exponential.

**Islands** in a search graph are positions which are constrained to be on the solution path. The use of inappropriate islands may make the problem more difficult (or even impossible to solve). Island search sacrifices optimality unless one is able to guarantee that the isands are on an optimal path.

## 8.3 Dynamic Programming

This is a general method for storing partial solutions to problems and retrieving them rather than recomputing them.

Dynamic programming can be used for finding paths in finite graphs by constructing a cost-to-goal function for nodes that gives the exact cost of a minimal-cost path from the node to a goal.

Dynamic programming has the advantage that it specifies what to do for every node, and so can be used given a goal for any starting position. It can be used to construct heuristic functions which can be used for $A^*$ and branch-and-bound searches.

Dynamic programming is useful when
• the goal nodes are explicit (the previous methods only assumed a function that recognizes goal nodes)
• a lowest-cost path is needed
• the graph is finite and small enough to be able to store the cost_to_goal value for each node
• the goal does not change very often, and
• the policy is used a number of times for each goal, so that the cost of generating the cost_to_goal values can be amortized over many instances of the problem.

The main problems with dynamic programming are that
• it only works when the graph is finite and the table can be made small enough to fit into memory
• an agent must recompute a policy for each different goal, and
• the time and space required is linear in the size of the graph, where the graph size for finite graphs is typically exponential in the path length.

# 9 Review

The main points:

- Many problems can be abstracted to the problem of finding paths in graphs.
- Breadth-first and depth-first searches can find paths in graphs without any extra knowledge beyond the graph.
- $A^*$ search can use a heuristic function that estimates the cost from a node to a goal. If graph is not pathological (see Proposition 3.2) and the heuristic is admissible, $A^*$ is guaranteed to find a lowest-cost path to a goal if one exists.
- Multiple-path pruning and cycle pruning can be used to make search more efficient.
- Iterative deepening and depth-first branch-and-bound searches can be used to find lowest-cost paths with less memory than methods, such as $A^*$, which store multiple paths.
- When graphs are small enough to store the nodes, dynamic programming records the actual cost of a lowest-cost path from each node to the goal, which can be used to find the next arc in an optimal path.