Solving problems by searching

In which we see how an agent can find a sequence of actions that achieves its goals when no single action will do

Piotr Fulmański

Instytut Nauk Ekonomicznych i Informatyki,
Państwowa Wyższa Szkoła Zawodowa w Płocku, Polska

October 25, 2015

Table of contents

Basic definitions

Our discussion of problem solving begins with precise definitions of **problems** and their **solutions**. We give several examples to illustrate these definitions.

We then describe several general-purpose search algorithms that can be used to solve these problems.

Uninformed vs. informed search algorithms

- We will see several **uninformed** search algorithms – algorithms that are given no information about the problem other than its definition. Although some of these algorithms can solve any solvable problem, none of them can do so efficiently.
- **Informed** search algorithms, on the other hand, can do quite well given some guidance on where to look for solutions.

World state

World state depends on the problem but generally we can say that it is a very small part of our (real) world relevant for thinking about the problem we have and the way we want o solve it.

Goal formulation

Goal formulation, based on the current situation and the agent's performance measure, is the first step in problem solving. We will consider a goal to be a set of world states – exactly those states in which the goal is satisfied. **The agent's task is to find out how to act, now and in the future, so that it reaches a goal state.** Before it can do this, it needs to decide what sort of actions and states it should consider.

Problem formulation

Problem formulation is the process of deciding what **actions** and **states** to consider, given a goal.

Assumptions

Assumptions before we formulate the problem and try to solve it

- For now, we assume that the environment is **observable**, so the agent always knows the current state.
- We also assume the environment is **discrete**, so at any given state there are only finitely many actions to choose from.
- We will assume the **environment is known**, so the agent knows which states are reached by each action.
- Finally, we assume that the **environment is deterministic**, so each action has exactly one outcome.

Under these assumptions, **the solution to any problem is a fixed sequence of actions**.

Assumptions

Assumptions before we formulate the problem and try to solve it

- For now, we assume that the environment is **observable**, so the agent always knows the current state.
- We also assume the environment is **discrete**, so at any given state there are only finitely many actions to choose from.
- We will assume the **environment is known**, so the agent knows which states are reached by each action.
- Finally, we assume that the **environment is deterministic**, so each action has exactly one outcome.

Under these assumptions, **the solution to any problem is a fixed sequence of actions**.

Assumptions

Assumptions before we formulate the problem and try to solve it

- For now, we assume that the environment is **observable**, so the agent always knows the current state.
- We also assume the environment is **discrete**, so at any given state there are only finitely many actions to choose from.
- We will assume the **environment is known**, so the agent knows which states are reached by each action.
- Finally, we assume that the **environment is deterministic**, so each action has exactly one outcome.

Under these assumptions, **the solution to any problem is a fixed sequence of actions**.

Assumptions

Assumptions before we formulate the problem and try to solve it

- For now, we assume that the environment is **observable**, so the agent always knows the current state.
- We also assume the environment is **discrete**, so at any given state there are only finitely many actions to choose from.
- We will assume the **environment is known**, so the agent knows which states are reached by each action.
- Finally, we assume that the **environment is deterministic**, so each action has exactly one outcome.

Under these assumptions, **the solution to any problem is a fixed sequence of actions**.

Assumptions

Assumptions before we formulate the problem and try to solve it

- For now, we assume that the environment is **observable**, so the agent always knows the current state.
- We also assume the environment is **discrete**, so at any given state there are only finitely many actions to choose from.
- We will assume the **environment is known**, so the agent knows which states are reached by each action.
- Finally, we assume that the **environment is deterministic**, so each action has exactly one outcome.

Under these assumptions, **the solution to any problem is a fixed sequence of actions**.

Assumptions

Assumptions before we formulate the problem and try to solve it

- For now, we assume that the environment is **observable**, so the agent always knows the current state.
- We also assume the environment is **discrete**, so at any given state there are only finitely many actions to choose from.
- We will assume the **environment is known**, so the agent knows which states are reached by each action.
- Finally, we assume that the **environment is deterministic**, so each action has exactly one outcome.

Under these assumptions, **the solution to any problem is a fixed sequence of actions**.

formulate–state–execute

- The process of looking for a sequence of actions that reaches the goal is called **search**.
- A search algorithm takes a problem as input and returns a **solution** in the form of action sequence.
- Once a solution is found, the actions it recommends can be carried out. This is called the **execution** phase.

Thus, we have a simple **formulate–state–execute** design pattern for the agent.

formulate–state–execute

- The process of looking for a sequence of actions that reaches the goal is called **search**.
- A search algorithm takes a problem as input and returns a **solution** in the form of action sequence.
- Once a solution is found, the actions it recommends can be carried out. This is called the **execution** phase.

Thus, we have a simple **formulate–state–execute** design pattern for the agent.

formulate–state–execute

- The process of looking for a sequence of actions that reaches the goal is called **search**.
- A search algorithm takes a problem as input and returns a **solution** in the form of action sequence.
- Once a solution is found, the actions it recommends can be carried out. This is called the **execution** phase.

Thus, we have a simple **formulate–state–execute** design pattern for the agent.

formulate–state–execute

- The process of looking for a sequence of actions that reaches the goal is called **search**.
- A search algorithm takes a problem as input and returns a **solution** in the form of action sequence.
- Once a solution is found, the actions it recommends can be carried out. This is called the **execution** phase.

Thus, we have a simple **formulate–state–execute** design pattern for the agent.

formulate–state–execute

- The process of looking for a sequence of actions that reaches the goal is called **search**.
- A search algorithm takes a problem as input and returns a **solution** in the form of action sequence.
- Once a solution is found, the actions it recommends can be carried out. This is called the **execution** phase.

Thus, we have a simple **formulate–state–execute** design pattern for the agent.

Goal after goal

Goal after goal

After formulating a goal and a problem to solve, the agent calls a search procedure to solve it. It then uses the solution to guide its actions, doing whatever the solution recommends as the next thing to do – typically, the first action of the sequence – and then removing that step from the sequence. Once the solution has been executed, the agent will formulate a new goal.

Simple problem solving agent (SPSA)

```
globar vars:
problem - a problem formulation
state - some description of the current world state

function SPSA(state){
local vars:
  goal - a goal, initially empty
  seq - an action sequence, initially empty

  do{
    action := SPSA_AUX(state)
    if(action is not null)
      state := EXECUTE_ACTION(action)
      // returns percept instead of simply state
      // in case of result which is nondeterministic
  }while(action in not NULL)
}
```

Simple problem solving agent

An auxiliary function (SPSA_AUX)

```
function SPSA_AUX(state) return an action {
  if (seq is empty) then {
    goal := FORMULATE_GOAL(state)
    problem := FORMULATE_PROBLEM(state,goal)
    seq := SEARCH(problem)

    if (seq = null) then
      return a null action
  }
  action := FIRST(seq)
  seq := REST(seq)
  return action
}
```

open-loop

Notice that while **the agent** is executing the solution sequence it **ignores its percepts when choosing an action** because it knows in advance what they will be. An agent that carries out its plans with its eyes closed, so to speak, must be quite certain of what is going on. Control theorists call this an **open-loop** system, because ignoring the percepts breaks the loop between agent and environment.

Problem

A **problem** can be defined formally by five components

- The **initial state** that the agent starts in.
- A description of the possible **actions** available to the agent. Given a particular state s, function `ACTION(`s`)` returns the set of actions that can be executed in s. We say that each of these actions is **applicable** in s.
- A description of what each action does; the formal name for this is the **transition model**, specified by a function `RESULT(`s`,`a`)` that returns the state that results from doing action a in state s. We also use the term **successor** to refer to any state reachable from a given state by a single action.

Problem

A **problem** can be defined formally by five components

- The **initial state** that the agent starts in.
- A description of the possible **actions** available to the agent. Given a particular state s, function `ACTION(`s`)` returns the set of actions that can be executed in s. We say that each of these actions is **applicable** in s.
- A description of what each action does; the formal name for this is the **transition model**, specified by a function `RESULT(`s`,`a`)` that returns the state that results from doing action a in state s. We also use the term **successor** to refer to any state reachable from a given state by a single action.

Problem

A **problem** can be defined formally by five components

- The **initial state** that the agent starts in.
- A description of the possible **actions** available to the agent. Given a particular state s, function `ACTION(`s`)` returns the set of actions that can be executed in s. We say that each of these actions is **applicable** in s.
- A description of what each action does; the formal name for this is the **transition model**, specified by a function `RESULT(`s`,`a`)` that returns the state that results from doing action a in state s. We also use the term **successor** to refer to any state reachable from a given state by a single action.

Problem

A **problem** can be defined formally by five components

- The **initial state** that the agent starts in.
- A description of the possible **actions** available to the agent. Given a particular state s, function `ACTION(`s`)` returns the set of actions that can be executed in s. We say that each of these actions is **applicable** in s.
- A description of what each action does; the formal name for this is the **transition model**, specified by a function `RESULT(`s`,`a`)` that returns the state that results from doing action a in state s. We also use the term **successor** to refer to any state reachable from a given state by a single action.

Problem

All the above together: **the initial state, actions, and transition model implicitly define the state space** of the problem – the set of all states reachable from the initial state by any sequence of actions. The state space forms the directed network or graph in which the nodes are states and the links between nodes are actions. A **path** in the state space is a sequence of states connected by a sequence of actions.

Problem

- The **goal test**, which determines whether a given state is a goal state. Sometimes there is an explicit set of possible goal states, and the test simply checks whether the given state is one of them. Sometimes the goal is specified by an abstract property rather than an explicitly enumerated set of states. For example, in chess, the goal is to reach a state called „checkmate", where the opponent's king is under attack and can't escape.
- A **path cost** function that assigns a numeric cost to each path. The problem-solving agent chooses a cost function that reflects its own performance measure.

A **solution** to a problem is an action sequence that leads from the initial state to a goal state. Solution quality is measured by the path cost function, and an **optimal solution** has the lowest path cost among all solutions.

Problem

- The **goal test**, which determines whether a given state is a goal state. Sometimes there is an explicit set of possible goal states, and the test simply checks whether the given state is one of them. Sometimes the goal is specified by an abstract property rather than an explicitly enumerated set of states. For example, in chess, the goal is to reach a state called „checkmate", where the opponent's king is under attack and can't escape.
- A **path cost** function that assigns a numeric cost to each path. The problem-solving agent chooses a cost function that reflects its own performance measure.

A **solution** to a problem is an action sequence that leads from the initial state to a goal state. Solution quality is measured by the path cost function, and an **optimal solution** has the lowest path cost among all solutions.

Problem

- The **goal test**, which determines whether a given state is a goal state. Sometimes there is an explicit set of possible goal states, and the test simply checks whether the given state is one of them. Sometimes the goal is specified by an abstract property rather than an explicitly enumerated set of states. For example, in chess, the goal is to reach a state called „checkmate", where the opponent's king is under attack and can't escape.
- A **path cost** function that assigns a numeric cost to each path. The problem-solving agent chooses a cost function that reflects its own performance measure.

A **solution** to a problem is an action sequence that leads from the initial state to a goal state. Solution quality is measured by the path cost function, and an **optimal solution** has the lowest path cost among all solutions.

Problem

- The **goal test**, which determines whether a given state is a goal state. Sometimes there is an explicit set of possible goal states, and the test simply checks whether the given state is one of them. Sometimes the goal is specified by an abstract property rather than an explicitly enumerated set of states. For example, in chess, the goal is to reach a state called „checkmate", where the opponent's king is under attack and can't escape.
- A **path cost** function that assigns a numeric cost to each path. The problem-solving agent chooses a cost function that reflects its own performance measure.

A **solution** to a problem is an action sequence that leads from the initial state to a goal state. Solution quality is measured by the path cost function, and an **optimal solution** has the lowest path cost among all solutions.

Abstraction

Every time we propose a formulation of the problem, even it seems reasonable, it is still a **model** – an abstract mathematical description – and not the real thing. Among others we have to leave out of our state description all these considerations which are irrelevent to the problem. The process of removing detail from a representation is called **abstraction**. In addition to abstracting the state description, we must abstract the actions themselves.

Validity and usefulness

- The abstraction is **valid** if we can expand any abstract solution into a solution in the more detailed world.
- The abstraction is **useful** if carrying out each of the action in the solution is easier than the oryginal problem.

The choice of a good abstraction thus involves removing as much detail as possible while retaining validity and ensuring that the abstract actions are easy to carry out. Were it not for the ability to construct useful abstractions, intelligent agents would be completely swamped by the real world.

Validity and usefulness

- The abstraction is **valid** if we can expand any abstract solution into a solution in the more detailed world.
- The abstraction is **useful** if carrying out each of the action in the solution is easier than the oryginal problem.

The choice of a good abstraction thus involves removing as much detail as possible while retaining validity and ensuring that the abstract actions are easy to carry out. Were it not for the ability to construct useful abstractions, intelligent agents would be completely swamped by the real world.

Validity and usefulness

- The abstraction is **valid** if we can expand any abstract solution into a solution in the more detailed world.
- The abstraction is **useful** if carrying out each of the action in the solution is easier than the oryginal problem.

The choice of a good abstraction thus involves removing as much detail as possible while retaining validity and ensuring that the abstract actions are easy to carry out. Were it not for the ability to construct useful abstractions, intelligent agents would be completely swamped by the real world.

Validity and usefulness

- The abstraction is **valid** if we can expand any abstract solution into a solution in the more detailed world.
- The abstraction is **useful** if carrying out each of the action in the solution is easier than the oryginal problem.

The choice of a good abstraction thus involves removing as much detail as possible while retaining validity and ensuring that the abstract actions are easy to carry out. Were it not for the ability to construct useful abstractions, intelligent agents would be completely swamped by the real world.

Toy problems

- vacuum cleaner
- 8-puzzle
- 8-queens problem

state, initial state, actions, transition model, goal test, path cost

Toy problems

Toy problems

- vacuum cleaner
- 8-puzzle
- 8-queens problem

state, initial state, actions, transition model, goal test, path cost

Toy problems

Toy problems

- vacuum cleaner
- 8-puzzle
- 8-queens problem

state, initial state, actions, transition model, goal test, path cost

Toy problems

- vacuum cleaner
- 8-puzzle
- 8-queens problem

state, initial state, actions, transition model, goal test, path cost

Real-world problems

- route-finding problem
- traveling salesperson problem
- automatic assembly sequencing

state, initial state, actions, transition model, goal test, path cost

Real-world problems

Real-world problems

- route-finding problem
- traveling salesperson problem
- automatic assembly sequencing

state, initial state, actions, transition model, goal test, path cost

Real-world problems

Real-world problems

- route-finding problem
- traveling salesperson problem
- automatic assembly sequencing

state, initial state, actions, transition model, goal test, path cost

Real-world problems

Real-world problems

- route-finding problem
- traveling salesperson problem
- automatic assembly sequencing

state, initial state, actions, transition model, goal test, path cost

Search tree

Having formulated some problems, we now need to solve them. A solution is an action sequence, so search algorithms work by considering various possible action sequences. The possible action sequences starting at the initial state from a **search tree** with the initial state at the root; the branches are actions and the **nodes** correspond to states in the state space of the problem.

Search tree

The root node of he tree corresponds to the initial state. Then we need to consider taking various actions. We do this by **expanding** the current state; that is applaying each legal action to the current state, thereby **generating** a new set of states. We add new **branches** from the **parent node** leading to new **child nodes**. Node with no children (at this moment) in the tree is a **leaf node**[a]. The set of all leaf nodes available for expansion at any given point is called the **frontier**.

After that we must choose which of these new possibilities to consider further.

---

[a] We say „at this moment" because applying legal action to leaf would generate a new set of states for which this leaf would be a parent.

The essence of search

This is the essence of search – following up one option now and putting the others aside for leater, in case the first choice does not lead to a solution.

General tree-search algorithm

General tree-search algorithm

Now we present the general tree-search algorithm. Search algorithms all share this basic structure; they vary primarily according to how they choose which state to expand next – the so-called **search strategy**.

Remark

Tree search is not exactly about searching already existing tree but about searching method which forms tree.
We can say this, because tree is generated by searching method; tree is generated dynamically while searching method works.

General tree-search algorithm

```
function treeSearch(problem) return a solution, or failure
{
  initialize the frontier using the initial state of problem
  loop
  {
    if (the frontier is empty) than return failure
    choose a leaf node and remove it from the frontier
    if (the node contains a gola state) then
      return the corresponding solution
    expand the chosen node, adding the resulting nodes
      to the frontier
  }
}
```

Redundant paths

Redundant paths exists whenever there is more than one way to get from one state to another. Sometimes redundant paths leads to infinite loop. The way to avoid exploring redundant paths is to remember where one has been. To do this we augment the `treeSearch` algorithm with a data structure called the **explored set**, which remembers every expanded node. Newly generated nodes that match previously generated nodes – ones in the explored set or the frontier – can be discarded instead of being added to the frontier. The new algorithm is called `graphSearch`.

Graph search

```
function graphSearch(problem) return a solution, or failure
{
  initialize the frontier using the initial state of problem
  initialize the explored set to be empty
  loop
  {
    if (the frontier is empty) than return failure
    choose a leaf node and remove it from the frontier
    if (the node contains a gola state) then
      return the corresponding solution
    add the node to the explored set
    expand the chosen node, adding the resulting nodes
      to the frontier only if not in the frontier
      or explored set
  }
}
```

Infrastructure for searching algorithms

Search algorithms required a data structure to keep track of the search tree or graph that is being constructed. For each node `n` of the tree, we have a structure that contains four components:

- `n.state`: the state in the state space to which the node corresponds;
- `n.parent`: the node in the search tree that generated this node;
- `n.action`: the action that was applied to the parent to generate the node;
- `n.pathCost`: the cost, traditionally denoted by $g(n)$, of the path from the initial state to the node, as indicated by the parent pointers.

We also use `n.childNode(action)` function which generates and returns the resulting child node, that is nodes for which the node `n` is a parent and are generated when `action` is applied to node `n`.

Measuring problem solving performance

Measuring problem solving performance

We can evaluate an algorithm's performance in four ways

- **completeness**: is the algorithm guaranteed to find a solution when there is one?
- **optimality**: does the strategy find the optimal solution (**optimal solution** is such a solution which has the lowest path cost among all solutions)?
- **time complexity**: how long does it take to find a soltion?
- **space complexity**: how much memory is needed to perform the search?

Measuring problem solving performance

Measuring problem solving performance

We can evaluate an algorithm's performance in four ways

- **completeness**: is the algorithm guaranteed to find a solution when there is one?
- **optimality**: does the strategy find the optimal solution (**optimal solution** is such a solution which has the lowest path cost among all solutions)?
- **time complexity**: how long does it take to find a soltion?
- **space complexity**: how much memory is needed to perform the search?

Measuring problem solving performance

Measuring problem solving performance

We can evaluate an algorithm's performance in four ways

- **completeness**: is the algorithm guaranteed to find a solution when there is one?
- **optimality**: does the strategy find the optimal solution (**optimal solution** is such a solution which has the lowest path cost among all solutions)?
- **time complexity**: how long does it take to find a soltion?
- **space complexity**: how much memory is needed to perform the search?

Measuring problem solving performance

Measuring problem solving performance

We can evaluate an algorithm's performance in four ways

- **completeness**: is the algorithm guaranteed to find a solution when there is one?
- **optimality**: does the strategy find the optimal solution (**optimal solution** is such a solution which has the lowest path cost among all solutions)?
- **time complexity**: how long does it take to find a soltion?
- **space complexity**: how much memory is needed to perform the search?

Measuring problem solving performance

Measuring problem solving performance

We can evaluate an algorithm's performance in four ways

- **completeness**: is the algorithm guaranteed to find a solution when there is one?
- **optimality**: does the strategy find the optimal solution (**optimal solution** is such a solution which has the lowest path cost among all solutions)?
- **time complexity**: how long does it take to find a soltion?
- **space complexity**: how much memory is needed to perform the search?

Uninformed and informed search strategies

Uninformed search strategies

Now we cover several strategies that comes under the heading of **uninformed search** (also called **blind search**). The term means that the strategies have no additional information about states beyond that provided in the problem definition. **All they can do is generate successors and distinguish a goal state from a non-gola state. All search strategies are distinguished by the order in which nodes are expanded.**

Informed search strategies

In contrary, strategies that know whether one non-goal state is „more promising" than other are called **informed search** or **heuristic search** strategies.

Uninformed and informed search strategies

Uninformed search strategies

Now we cover several strategies that comes under the heading of **uninformed search** (also called **blind search**). The term means that the strategies have no additional information about states beyond that provided in the problem definition. **All they can do is generate successors and distinguish a goal state from a non-gola state. All search strategies are distinguished by the order in which nodes are expanded.**

Informed search strategies

In contrary, strategies that know whether one non-goal state is „more promising" than other are called **informed search** or **heuristic search** strategies.

Breadth-first search

Breadth-first search is a simple strategy in which the root node is expanded first, than all the successors of the root node are expanded next, than their successors, and so on. In general, all the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded.

Breadth-first search is an instance of the general graph-search algorithm in which the shallowest unexpanded node is chosen for expansion. This is achieved very simply by using FIFO queue for the frontier. Thus, new nodes (which are always deeper than their parents) go to the back of the queue, and old nodes, which are shallower than the new nodes, get expanded first. There is one slight tweak on the general graph-search algorithm, which is that the goal test is applied to each node when it is generated rather than it is selected for expansion.

Breadth-first search

```
function breadthFirstSearch(problem) return a solution, or failure
{
  node.state := problem.initialState
  if (problem.goalTest(node.state) = true) than
    return solution(node)
  frontier := a FIFO queue with node as the only element
  explored := an empty set
  loop{
    if (the frontier is empty) than return failure
    //choose a leaf node and remove it from the frontier
    node := pop(frontier)
    // chooses the shallowest node in frontier
    explored.add(node.state)

    for each action in problem.actions(node.state){
      child := node.childNode(action)
      if (child.state is not in explored or frontier) then
      {
        if (problem.goalTest(child.state)) then
          return solution(child)
        frontier.add(child)
      }
    }
  }
}
```

Uniform-cost search

Uniform-cost search

When all step cost are equal, breadth-first search is optimal because it always expands the shallowest unexpanded node. By a simple extension, we can find an algorithm that is optimal with any step-cost function. Instead of expanding the shallowest node, **uniform-cost search** expands the node n with the lowest path cost $g(n)$. This is done by storing the frontier as a priority queue ordered by g.

Uniform-cost search

```
function uniformCostSearch(problem) return a solution, or failure
{
  node.state := problem.initialState
  node.pathCost := 0
  if (problem.goalTest(node.state) = true) than
    return solution(node)
  frontier := a priority queue ordered by pathCost,
              with node as the only element
  explored := an empty set
  loop{
    if (the frontier is empty) than return failure
    //choose a leaf node and remove it from the frontier
    node := pop(frontier)
    /* chooses the lowest-cost node in frontier
       and remove it from the frontier */
    if (problem.goalTest(node.state)) then
      return solution(node)
    explored.add(node.state)

    for each action in problem.actions(node.state){
      child := node.childNode(action)
      if (child.state is not in explored or frontier) then
      {
        frontier.add(child)
      }
      else if child.state is in frontier with higher pathCost then
      {
        replace the frontier node with child
      }
    }
```

Uniform-cost search – differences

In addition on the ordering of the queue by path cost there are two other significant differences from breadth-first search.

- The first is that the goal test is applied to a node when it is selected for expansion (as in the generic graph-search algorithm) rather than when it is first generated.
- The second difference is that a test is added in case a better path is found to a node currently on the frontier.

Depth-first search

Depth-first search always expands the deepest node in the current frontier of the search tree. The search proceeds immediately to the deepest level of the search tree, where the nodes have no successors. As those nodes are expanded, they are dropped from the frontier, so then the search „backs up" to the next deepest node that still has unexplored successor. **The depth-first search algorithm is an instance of the graph-search algorithm which uses LIFO queue. A LIFO queue means that the most recently generated node is chosen for expansion.** This must be the deepest unexpanded node because it is one deeper than its parent – which, in turn, was the deepest unexpanded node when it was selected.

Depth-first search

So, to implement `depthFirstSearch` algorith it is enought to take previously described `breadthFirstSearch` algorithm and replace FIFO by LIFO.
This „detail” changes significantly the behaviour of the algorithm.

Depth-limited search

The embarrassing failure of depth-first search in infinite state spaces can be alleviated by supplying depth-first search with a predetermined dept limit *limit*. That is, nodes at depth *limit* are treated as if they have no successors. This approach is called **depth-limited search**.

Depth-limited search

```
function depthLimitedSearch(problem) return a solution, or failure/cutoff
  return recursiveDLS(problem.getInitialState(),problem, limit)

function recursiveDLS(node, problem, limit) return a solution, or failure/cutoff
{
  if (problem.goalTest(node.state) = true) than
    return solution(node)
  else if limit=0 then
    return cutoff
  else {
    cutoffOccurred := false
    for each action in problem.actions(node.state)
    {
      child := node.childNode(action)
      result := recursiveDLS(child, problem, limit-1)
      if (result = cutoff)
        then cutoffOccurred := true
      else if (result != failure) then
        return result
    }
    if (cutoffOccurred = true) then
      return cutoff
    else
      return failure
  }
}
```

Iterative deepening depth-first search

Iterative deepening depth-first search

Iterative deepening depth-first search (or iterative deepening search) is a general strategy, often use in combination with depth-first tree search, that finds the best depth limit. It does this by gradually increasing the limit – first 0, then 1, then 2, and so on – until a goal is found.
The iterative deepening combines the benefits of depth-first and breadth-first search.

Iterative deepening depth-first search

```
function iterativeDeepeningSearch(problem) return a solution, or failure
{
  for depth := 0 to infinity do
  {
    result := depthLimitedSearch(problem, depth)
    if (result != cutoff) then return result
  }
}
```

Bidirectional search

The idea behind bidirectional search is to run two simultaneous searches – one forward from the initial state and the other backward from the goal – hoping that the two searches meet in the middle.
Thers is one very important problem. Consider the question of what we mean by „the goal" in searching „backward from the goal". For the 8-puzzle there is just one goal so the backward search is very much like the forward search. If there are several explicitly listed goal states than we can construct a new dummy goal state whose immediate predecessors are all the actual goal states. But if the goal is an abstract description, such as the goal that „no queen attacks another queen" in the n-queens problem, than bidirectional search is difficult to use.

Comparing uninformed search strategies

	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative-Deepening	Bidirectional
Complete	Yes	Yes	No	No	Yes	Yes
Time	$O(b^{d})$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^{m})$	$O(b^{l})$	$O(b^{d})$	$O(b^{d/2})$
Space	$O(b^{d})$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(bl)$	$O(bd)$	$O(b^{d/2})$
Optimal	Yes	Yes	No	No	Yes	Yes

b – the branching factor;
d – the depth of the shallowest solution;
m – the maximum depth of the search tree;
l – the depth limit
Explanations for superscript:
a – complete if b is finite;
b – complete if step costs $> \epsilon$ for positive F ;
' optimal if step costs are all identical;
d if both directions use breadth-first search.

Informed (heuristic) search strategies

Now we show how an informed search strategy – one that uses problem-specific knowledge beyond the definition of the problem itself – can find solution more efficiently than can an uninformed strategy.
A general approach we consider is called **best-first search**. Best-first search is an instance of the general `TREE-SEARCH` or `GRAPH-SEARCH` algorithm in which a node is selected for expansion based on an **evaluation function**, $f(n)$. The evaluation function is construed as a cost estimate, so the node with the lowest evaluation is expanded first. **The implementation of the best-first graph search is identical to that for uniform-cost search, except for the use of f instead of g to order the priority queue**.

Heuristic function

Heuristic function

The choice of f determines the search strategy. Most best-first algorithms include as a component of f a **heuristic function**, denoted $h(n)$; $h(n)$ = estimated cost of the cheapest path from the state at node n to a goal state.
Heuristic (from the Greek: „find" or „discover") is an adjective for experience-based techniques that help in problem solving, learning and discovery. A heuristic method is used to rapidly come to a solution that is hoped to be close to the best possible answer, or „optimal solution".

Greedy best-first search

Greedy best-first search

Greedy best-first search tries to expand the node that is closest to the goal, on the grounds that this is likely to lead to a solution quickly. At each step algorithm tries to get as close to the goal as it can. Thus, it evaluates nodes by using just the heuristic function; that is, $f(n) = h(n)$.

A* search: minimizing the total estimated solution cost

A*

The most widely known form of best-first search is called **A* search**. It evaluates nodes by combining $g(n)$, the cost to rach the node, and $h(n)$, he cost to get from the node to the goal:

$$f(n) = g(n) + h(n).$$

Since $g(n)$ gives the path cost from the start node to node n, and $h(n)$ is the estimated cost of the cheapest path from n to the goal, we have $f(n) =$ estimated cost of cheapest solution through n.
Thus if we are trying to find the cheapest solution, a reasonable thing to try first is the node with the lowest value of $g(n) + h(n)$. It turns out that this strategy is more than just reasonable: provided that the heuristic function $h(n)$ satisfies certain conditions (see next slide), A* search is both complete and optimal. **The A* algorith is identical to uniform-cost-search except that A* uses $g + h$ instead of g.**

Definition

A heuristic $h(n)$ is admissible if for every node n, $h(n) \leq h^*(n)$, where $h^*(n)$ is the true cost to reach the goal state from n.

An admissible heuristic never overestimates the cost to reach the goal, i.e., it is optimistic. Example: $h_{SLD}(n)$ – straight-line distace, never overestimates the actual road distance

Theorem

*If $h(n)$ is admissible, A * using TREE-SEARCH is optimal.*

Definition

A heuristic is consistent if for every node n, every successor n' of n generated by any action a

$$h(n) \leq cost(n, a, n') + h(n')$$

Theorem

*If $h(n)$ is consistent, A * using GRAPH-SEARCH is optimal.*

Comparision of algorithm

Previous slides introduced a following costs

- $g(n)$ — cost so far to reach n;
- $h(n)$ — estimated cost from n to goal;
- $f(n) = g(n) + h(n)$ — estimated total cost of path through n to goal.

As we have seen all search algorithms discussed so far share this basic structure; they follow TREE-SEARCH or GRAPH-SEARCH schema but vary according to how they choose which state to expand next – the so-called **search strategy**.

- Breadth-First Search uses FIFO
- Depth-First Search uses LIFO
- Uniform Cost Search uses PQ (priority queue) with function $f(n) = g(n)$ used to calculate priority value for node n.
- Best First Search (Greedy Best-First Search variant) uses PQ with $f(n) = h(n)$.
- Best First Search (A* variant) uses PQ with $f(n) = g(n) + h(n)$.