Artificial Intelligence

3. Classical Search, Part I: Basics, and Blind Search

Got a Problem? Gotta Solve It!

Jana Koehler Álvaro Torralba

SAARLAND UNIVERSITY
COMPUTER SCIENCE

Summer Term 2019

Thanks to Prof. Hoffmann for slide sources

Agenda

Disclaimer

So far, we had a nice philosophical chat about "intelligence" et al.

As of today, we look at technical work.

Naturally, we don't start with the most complex action-decision framework. We start with the *simplest* possible one . . .

(Despite that simplicity, it's highly relevant in practice!)

A (Classical Search) Problem

→ Problem: Find a route to Moscow.

- Starting from an initial state ... (SB)
- ... apply actions ... (Using a road segment)
- ... to reach a goal state. (Moscow)
- Performance measure:

Another (Classical Search) Problem (The "15-Puzzle")

→ Problem: Move tiles to transform left state into right state.

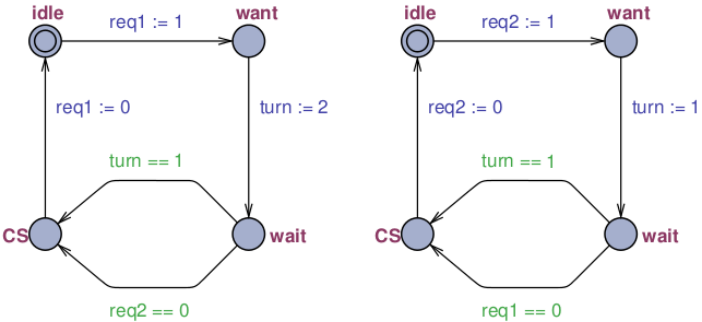
9	2	12	6
5	7	14	13
3	4	1	11
15	10	8	

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

- Starting from an initial state ... (Left)
- ... apply actions ... (Moving a tile)
- ... to reach a goal state. (Right)
- Performance measure: Minimize summed-up action costs. (Each move has cost 1, so we minimize the number of moves)

Yet Another (Classical Search) Problem

→ Problem: Finding bugs in software artifacts.



Classical Search Problems

. . . restrict the agent's environment to a very simple setting:

- Finite numbers of states and actions (in particular: discrete).
- Single-agent (nobody else around).
- Fully observable (agent knows everything).
- Deterministic (each action has only one outcome).
- Static (if the agent does nothing, the world doesn't change).

→ All of these restrictions can be removed, and a lot of work in AI considers such more general settings. We will talk about some of this in later chapters (but not in the present one).

→ Classical search problems are one of the simplest classes of action choice problems an agent can be facing. Despite that simplicity, classical search problems are very important in practice (see also next slide).

→ And despite that "simplicity", these problems are computationally hard! Typically harder than **NP** . . .

Examples of Classical Search Problems

Just to name a few:

- Route planning (e.g. Google Maps).
- Puzzles (Rubic's Cube, 15-Puzzle, Towers of Hanoi ...).
- Detecting bugs in software and hardware.
- Non-player-characters in computer games.
- Travelling Salesman Problem (TSP). Actions = moves in the graph.
- Robot assembly sequencing. Planning of the assembly of complex objects. Actions = robot activities.
- Attack planning. Finding a hack into a secured network. Used for regular security testing. Actions = exploits.
- Query optimization in databases. Actions = rewriting operations.
- Sequence alignment in Bioinformatics. Actions = re-alignment operations.
- Natural language sentence generation. Actions = add another word to a partial sentence.

Our Agenda for This Topic

→ Our treatment of the topic "Classical Search" consists of Chapters 4 and 5.

- **This Chapter:** Basic definitions and concepts; blind search.

  → Sets up the framework. Blind search is ideal to get our feet wet. It is not wide-spread in practice, but it is among the state of the art in certain applications (e.g., software model checking).

- **Chapter 5:** Heuristic functions and informed search.

  → Classical search algorithms exploiting the problem-specific knowledge encoded in a heuristic function. Typically much more efficient in practice.

Our Agenda for This Chapter

- **What (Exactly) Is a "Problem":** How are they formally defined?

  → Get ourselves on firm ground.

- **How To Put the Problem Into the Computer:** How are problems specified?

  → There are 3 fundamentally different methods, and the choice we make has a huge impact on practice. (The search algorithms we introduce here work for all 3 in principle.)

- **Basic Concepts of Search:** What are search spaces?

  → Sets the stage for the consideration of search strategies.

- **(Non-Trivial) Blind Search Strategies:** How to guarantee optimality? How to make the best use of time and memory?

  → Blind search serves to get started, and is used in some applications.

→ Some implementation details, as well as plain breadth-first search and depth-first search, are moved to the "Lookup Section" and won't be discussed.

Before We Begin

→ To precisely specify how we solve search problems algorithmically, we first need a formal definition.

That definition really is quite simple:

- The underlying base concept are state spaces.
- State spaces are (annotated) graphs.
- Paths to goal states correspond to solutions.
- Cheapest such paths correspond to optimal solutions.

State Spaces

Every problem Π specifies a state space Θ: (Exactly how Π specifies Θ is the subject of the next section)

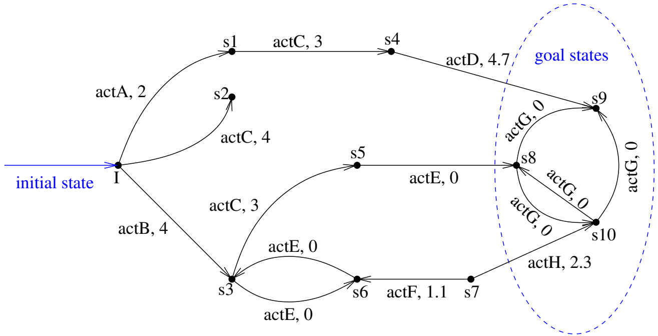
Definition (State Space). *A state space is a 6-tuple $\Theta = (S, A, c, T, I, S^G)$ where:*

- *S is a finite set of states.*
- *A is a finite set of actions.*
- *$c : A \mapsto \mathbb{R}_0^+$ is the cost function.*
- *$T \subseteq S \times A \times S$ is the transition relation. We require that T is deterministic, i.e., for all $s \in S$ and $a \in A$, there is at most one state s' such that $(s, a, s') \in T$. If such (s, a, s') exists, then a is applicable to s.*
- *$I \in S$ is the initial state.*
- *$S^G \subseteq S$ is the set of goal states.*

We say that Θ has the transition (s, a, s') if $(s, a, s') \in T$. We also write $s \xrightarrow{a} s'$, or $s \rightarrow s'$ when not interested in a.
We say that Θ has unit costs if, for all $a \in A$, $c(a) = 1$.

State Spaces: Illustration

Directed labeled graphs + mark-up for initial state and goal states:



- Does this Θ have unit costs?
- Which actions are applicable to the initial state?
- Is T deterministic?

State Spaces Terminology

Some commonly used terms:

- s' successor of s if $s \to s'$; s predecessor of s' if $s \to s'$.
- s' reachable from s if there exists a sequence of transitions:

$$s = s_0 \xrightarrow{a_1} s_1, \ldots, s_{n-1} \xrightarrow{a_n} s_n = s'$$

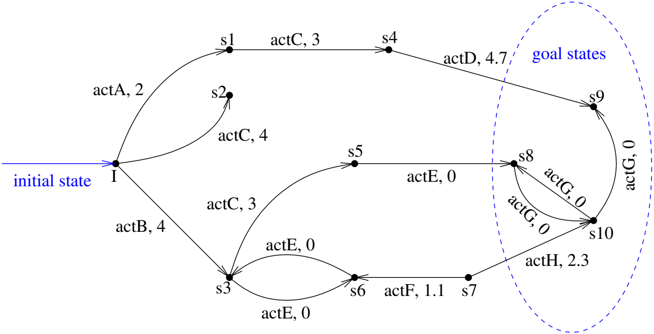
  - $n = 0$ possible; then $s = s'$.
  - $a_1, \ldots, a_n$ is called path from s to s'.
  - $s_0, \ldots, s_n$ is also called path from s to s'.
  - The cost of that path is $\sum_{i=1}^{n} c(a_i)$.
- s' reachable (without reference state) means reachable from I.
- s is solvable if some $s' \in S^G$ is reachable from s; else, s is a dead end.

Definition (State Space Solutions). *Let $\Theta = (S, A, c, T, I, S^G)$ be a state space, and let $s \in S$. A solution for s is a path from s to some $s' \in S^G$. The solution is optimal if its cost is minimal among all solutions for s. A solution for I is called a solution for Θ. If a solution exists, then Θ is solvable.*

$\rightarrow$ Unsolvable Θ do occur naturally! E.g., in debugging "unsolvable" =

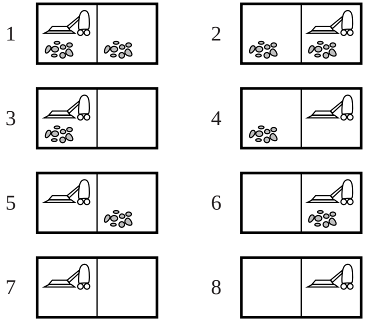
State Spaces: Illustration, ctd.

Directed labeled graphs + mark-up for initial state and goal states:



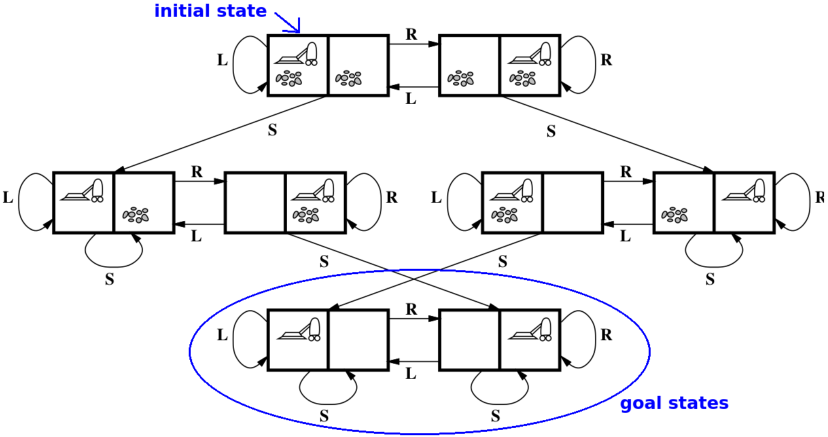
- Are all states in Θ reachable?
- Are all states in Θ solvable?
- What are the optimal solutions for Θ?

Example Vacuum Cleaner



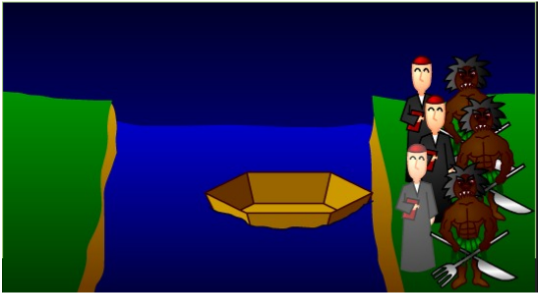
- Starting from state 1 (dirty!) ...
- ... go right(R), left (L), or suck (S) ...
- ... to clean the apartment.
- Performance measure: Minimize number of actions.

Example Vacuum Cleaner: State Space



Example Missionaries and Cannibals

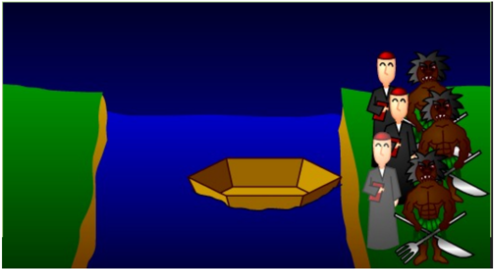
→ Problem: Cross the river without being eaten.



- Starting with everybody on the right bank ...
- ... use the boat which carries ≤ 2 people ...
- ... to get everybody to the left bank.
- If, at any point in time, missionaries are outnumbered by cannibals on either bank, then ... game over.

Example Missionaries and Cannibals: Clarifications

→ Problem: Cross the river without being eaten.



- We consider only the states at the end of each boat ride, not the situation during the boat ride.
- At the end of each move, everybody leaves the boat (in other words, any people left in the boat count as being on the river bank); and the game is over in case that results in more C than M.
- Moves after which the game would be over are disallowed, i.e., these actions are not applicable.

Questionnaire

Question!

For which of these problems can a solvable state space Θ contain a reachable dead end?

(A): Route Planning

(B): 15-Puzzle

(C): Debugging

(D): Missionaries and Cannibals

Example Route Planning: State Space

- State set S: $\{at(x) \mid x$ city in Europe$\}$.
- Action set A: $\{move(x, y) \mid x, y$ linked by a road segment$\}$.
- Cost function c: Maps each $move(x, y)$ to the length of the road segment.
- Transition relation T: $\{(at(x), move(x, y), at(y)) \mid x, y$ linked by a road segment$\}$.
- Initial state I: $at(\text{SB})$.
- Goal states S^G: $\{at(\text{Moscow})\}$.

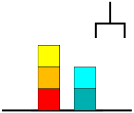
15-Puzzle: States are position assignments to all tiles, actions accordingly.

Software debugging: States are value assignments to all variables (including the program counter PC), actions are program commands (e.g., "Goto 10" becomes $PC := 10$).

Example Missionaries and Cannibals: State Space

- State set S: Triples (M, C, B) with $0 \leq M, C \leq 3$, $0 \leq B \leq 1$. Here, M, C, and B respectively represent the number of missionaries, cannibals, and boats currently on the right bank.
- Initial state I: $(3, 3, 1)$.
- Goal states S^G: $\{(0, 0, 0), (0, 0, 1)\}$.
- Cost function c: Unit 1.
- Action set A:
- Transition relation T: Accordingly.

So, Why All the Fuss? Example Blocksworld



- n blocks, 1 hand.
- A single action either takes a block with the hand or puts a block we're holding onto some other block/the table.

blocks	states	blocks	states
1	1	9	4596553
2	3	10	58941091
3	13	11	824073141
4	73	12	12470162233
5	501	13	202976401213
6	4051	14	3535017524403
7	37633	15	65573803186921
8	394353	16	1290434218669921

$\rightarrow$ State spaces may be huge. In particular, the state space is typically exponentially large in the size of its specification via the problem Π (up next).

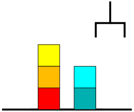
$\rightarrow$ In other words: Search problems typically are computationallly hard (e.g., optimal Blocksworld solving is **NP**-complete).

Questionnaire

Questionnaire, ctd.

Why Am I Talking About This?

Remember the Blocksworld? 16 blocks, 1290434218669921 states.



- n blocks, 1 hand.
- A single action either takes a block with the hand or puts a block we're holding onto some other block/the table.

Π **vs.** Θ**:** Π is the description of the problem ("A single action either takes a ..."), and Θ is the state space corresponding to this description. (Similar for software debugging etc.)

→ Huge state spaces Θ can often be specified by small problem descriptions Π. It is thus important to distinguish the two.

→ So the question becomes: What are suitable "problem descriptions"?

Option 1: Blackbox Description

→ The blackbox description of a problem Π is an API (a programming interface) providing functionality allowing to construct the state space:

Blackbox Description of a Problem

- InitialState(): Returns the initial state of the problem.
- GoalTest(s): Returns a Boolean, "true" iff state s is a goal state.
- Cost(a): Returns the cost of action a.
- Actions(s): Returns the set of actions that are applicable to state s.
- ChildState(s, a): Requires that action a is applicable to state s, i.e., there is a transition $s \xrightarrow{a} s'$. Returns the outcome state s'.

- "Specifying the problem" = programming the API.
- Huge state spaces can be specified with little program code.

→ The API does not provide the search with any knowledge about the problem, other than the bare essentials needed to generate the state space. Hence the name "blackbox", as opposed to: up next.

Option 2: Declarative/Whitebox Description

→ The declarative description of Π comes in a problem description language:

Declarative Description of a Problem

There are many ways to do this. Here's one:

- P: Set of Boolean variables (propositions).
- I: Subset of P, indicating which propositions are true in the initial state.
- G: Subset of P, where s is a goal state iff $G \subseteq s$.
- A: Set of actions a, each with precondition pre_a, add list add_a, and delete list del_a; a applicable to s iff $pre_a \subseteq s$, outcome state is $(s \cup add_a) \setminus del_a$.
- c: Maps each $a \in A$ to its cost $c(a)$.

- This language is called "STRIPS"; we'll get back to it in **Chapter 14**.
- "Specifying the problem" = writing STRIPS. The computer then inputs that description and can generate the state space.

→ Declarative descriptions are *strictly more powerful* than blackbox ones. They allow to implement the API, and much more (e.g. analyze/simplify the problem).

Option 3: Explicit Description

→ The explicit description describes Π simply in terms of its state space:

Explicit Description of a Problem

$\Pi = \Theta$: We simply input the state space graph (in some representation).

- "Specifying the problem" = writing down the state space.
- Impossible for large state spaces.
- Can be solved easily, *in the size of the state space*: Dijkstra's algorithm.

→ Explicit descriptions do not have the ability to compactly describe large state spaces.

→ They are used if state spaces are "small" (only 100000s of states) and runtime is very limited. This is typically the case in route planning. A prominent application is in Video games, where routes for all non-player agents must be computed in microseconds.

So What?

→ Declarative descriptions enable general (classical search) problem solving:

(some new classical search problem)

describe problem in generic language $\mapsto$ use off-the-shelf solver

(its solution)

- Little programming effort, easy to adapt to changes.
- Core topic of FAI group; will be covered in **Chapters 14 and 15**.
- In this and the next chapter, we assume the blackbox description. Explicit descriptions will only be used in (some) illustrative examples.
- In principle, the search strategies we will discuss can be used with *any* problem description that allows to implement the blackbox API.

Questionnaire

Questionnaire, ctd.

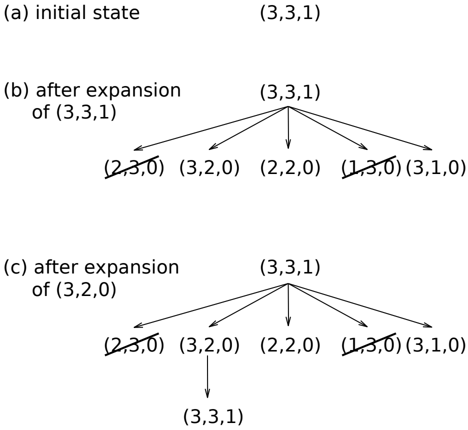
Question!

(A) In the blackbox description of route planning, what does ChildState(s, a) return?
(B) In the blackbox description of debugging, what does Actions(s) return?

Search Illustration

How to "search"? Start at the initial state. Then, step-by-step, expand a state by generating its successors ...

→ Search space.



Search Terminology

Search node n: Contains a *state* reached by the search, plus information about how it was reached.

Path cost $g(n)$: The cost of the path reaching n.

Optimal cost g^*: The cost of an optimal solution path. For a state s, $g^*(s)$ is the cost of a cheapest path reaching s.

Node expansion: Generating all successors of a node, by applying all actions applicable to the node's state s. Afterwards, the *state* s itself is also said to be expanded.

Search strategy: Method for deciding which node is expanded next.

Open list: Set of all *nodes* that currently are candidates for expansion. Also called frontier.

Closed list: Set of all *states* that were already expanded. Used only in graph search, not in tree search (up next). Also called explored set.

Tree Search vs. Graph Search

Duplicate Elimination:

- Maintain a closed list.
- Check for each generated state s' whether s' is in the closed list. If so, discard s'.

Tree Search:

- . . . is another word for "don't use duplicate elimination".
- Search space is "tree-like": We do not consider the possibility that the same state may be reached from more than one predecessor.
- The same state may appear in many search nodes.
- Main advantage: lower memory consumption (no closed list needed).

Graph Search:

- . . . is another word for "use duplicate elimination".
- Search space is "graph-like": We do consider said possibility.

Generic Tree Search Procedure

```
function TREE-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  loop do
     if the frontier is empty then return failure
     choose a leaf node and remove it from the frontier
     if the node contains a goal state then return the corresponding solution
     expand the chosen node, adding the resulting nodes to the frontier
```

- This is merely a *guideline* for tree search!
- Concrete algorithms often differ in the details, for efficiency reasons.

Generic Graph Search Procedure

```
function GRAPH-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  initialize the explored set to be empty
  loop do
     if the frontier is empty then return failure
     choose a leaf node and remove it from the frontier
     if the node contains a goal state then return the corresponding solution
     add the node's state to the explored set
     expand the chosen node, adding the resulting nodes to the frontier
        only if node's state not in the explored set
```

- This is merely a *guideline* for graph search!
- Concrete algorithms often differ in the details, for efficiency reasons.

Criteria for Evaluating Search Strategies

Guarantees:

Completeness: Is the strategy guaranteed to find a solution when there is one?

Optimality: Are the returned solutions guaranteed to be optimal?

Complexity:

Time Complexity: How long does it take to find a solution? (Measured in generated states.)

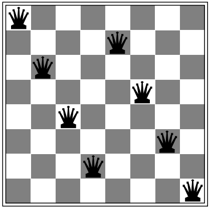
Space Complexity: How much memory does the search require? (Measured in states.)

Typical state space features governing complexity:

Branching factor b: How many successors does each state have?

Goal depth d: The number of actions required to reach the shallowest goal state.

Questionnaire



- Chess board, numbering the 8 columns $C_1, \ldots, C_8$ from left to right.
- 8 queens $Q_1, \ldots, Q_8$, each Q_i to be placed "in its own" column C_i.
- We fill the columns left to right, i.e., the actions allow to place Q_i somewhere in C_i, provided all of $Q_1, \ldots, Q_{i-1}$ have already been placed.
- Goal: Placement where no queens attack each other.

Question!

Tree search always terminates in?

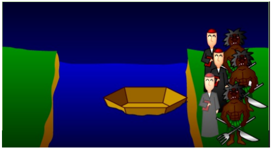
(A): 15-Puzzle.

(B): Missionaries and Cannibals.

(C): Vacuum Cleaning.

(D): 8-Queens.

Questionnaire, ctd.



- 3 missionaries, 3 cannibals. Boat holds ≤ 2.
- Never leave k missionaries alone with $> k$ cannibals.
- States: (M, C, B) numbers on right bank.

Question!

Which are successor states of $(1,1,0)$ in Missionaries and Cannibals?

(A): $(1,1,1)$.
(B): $(2,2,1)$.
(C): $(3,3,1)$.
(D): $(2,1,1)$.

Preliminaries

Blind search vs. informed search:

- Blind search does not require any input beyond the problem API.

  Pros and Cons: Pro: No additional work for the programmer. Con: It's not called "blind" for nothing . . . same expansion order regardless what the problem actually is. Rarely effective in practice.

- Informed search requires as additional input a heuristic function h (**Next Chapter**) that maps states to estimates of their goal distance.

  Pros and Cons: Pro: Typically more effective in practice. Con: Somebody's gotta come up with/implement h.

  $\rightarrow$ Note: In planning, h is generated automatically from the declarative problem description (**Chapters 14 and 15**).

Preliminaries, ctd.

Blind search strategies covered:

- Breadth-first search, depth-first search.
- Uniform-cost search. Optimal for non-unit costs.
- Iterative deepening search. Combines advantages of breadth-first search and depth-first search.

Blind search strategy not covered:

- Bi-directional search. Two separate search spaces, one forward from the initial state, the other backward from the goal. Stops when the two search spaces overlap.

Content I will not talk about:

- Breadth-first search and depth-first search.
- The pseudo-code in what follows will use some basic functions.

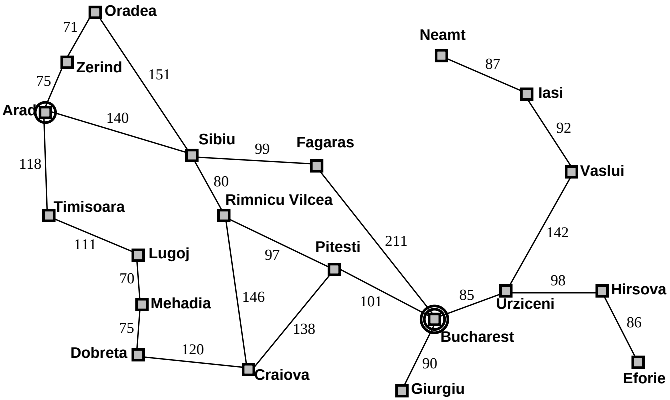
→ Both are in the "Lookup Section". I strongly recommend you read that section. Post any questions you may have in Moodle.

Uniform-Cost Search: Pseudo-Code

function Uniform-Cost Search*(problem)* **returns** a solution, or failure
  node $\leftarrow$ a node n with n.*State*=*problem.InitialState*
  frontier $\leftarrow$ a priority queue ordered by ascending g, only element n
  explored $\leftarrow$ empty set of states
  loop do
    if *Empty?(frontier)* **then return** failure
    $n \leftarrow$ *Pop(frontier)*
    if *problem.GoalTest*(n.*State*) **then return** *Solution*(n)
    explored $\leftarrow$ *explored*$\cup n$.State
    for each *action* a **in** *problem.Actions*(n.*State*) **do**
      $n' \leftarrow$ *ChildNode*(*problem*,n,a)
      if n'.State$\notin$ [*explored* $\cup$ States(*frontier*)] **then** *Insert*(n', $g(n')$, *frontier*)
      else if ex. $n'' \in$*frontier* s.t. n''.State= n'.State and $g(n') < g(n'')$ **then**
        replace n'' in *frontier* with n'

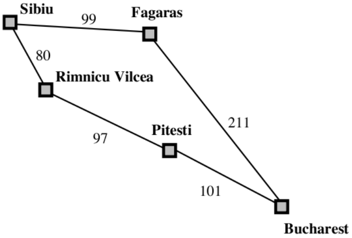
- Goal test at node-expansion time.
- Duplicates in frontier replaced in case of cheaper path.

Russel & Norvig's Example: Route Planning in Romania



Arad	366
Bucharest	0
Craiova	160
Drobeta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Route Planning in Romania: Uniform-Cost Search



Search protocol:

Uniform-Cost Search: Guarantees and Complexity

Lemma. *Uniform-cost search is equivalent to Dijkstra's algorithm on the state space graph.* (Obvious from the definition of the two algorithms.)

→ The only differences are: (a) we generate only a part of that graph incrementally, whereas Dijkstra inputs and processes the whole graph; (b) we stop when we reach any goal state (rather than a fixed target state given in the input).

Theorem. *Uniform-cost search is optimal.* (Because Dijkstra's algorithm is optimal.)

- Completeness:
- Time complexity: $O(b^{1+\lfloor g^*/\epsilon \rfloor})$ where g^* denotes the cost of an optimal solution, and ϵ is the positive cost of the cheapest action.
- Space complexity: Same as time complexity.

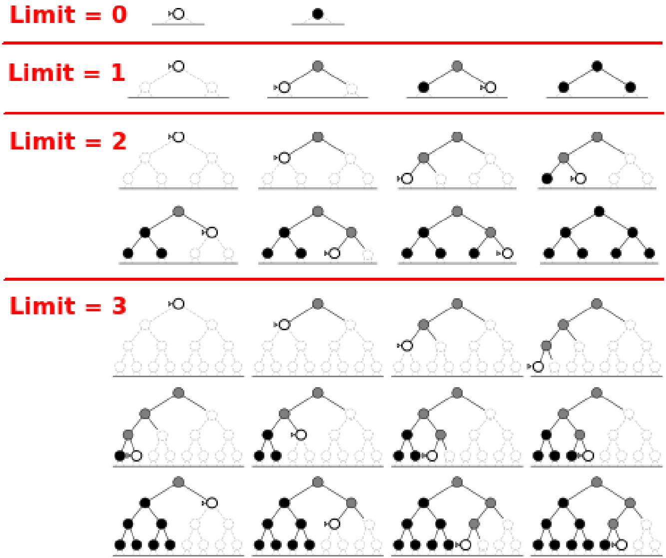
Iterative Deepening Search: Pseudo-Code

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
   for depth = 0 to ∞ do
      result ← DEPTH-LIMITED-SEARCH(problem, depth)
      if result ≠ cutoff then return result
```

```
function Depth-Limited Search(problem, limit) returns a solution, or failure/cutoff
  node ← a node n with n.state=problem.InitialState
  return Recursive-DLS(node, problem, limit)

function Recursive-DLS(n, problem, limit) returns a solution, or failure/cutoff
  if problem.GoalTest(n.State) then return the empty action sequence
  if limit = 0 then return cutoff
  cutoffOccured ← false
  for each action a in problem.Actions(n.State) do
     n′ ← ChildNode(problem,n,a)
     result ← Recursive-DLS(n′, problem, limit−1)
     if result = cutoff then cutoffOccured ← true
        else if result ≠ failure then return a ∘ result
  if cutoffOccured then return cutoff else return failure
```

Iterative Deepening Search: Illustration



Iterative Deepening Search: Guarantees and Complexity

"Iterative Deepening Search=
Keep doing the same work over again until you find a solution."

BUT: Optimality? Completeness? Space complexity?

Time complexity:

Breadth-First-Search	$b + b^2 + \cdots + b^{d-1} + b^d \in O(b^d)$
Iterative Deepening Search	$(d)b + (d-1)b^2 + \cdots + 3b^{d-2} + 2b^{d-1} + 1b^d \in O(b^d)$

Example: $b = 10, d = 5$

Breadth-First Search	$10 + 100 + 1,000 + 10,000 + 100,000 = 111,110$
Iterative Deepening Search	$50 + 400 + 3,000 + 20,000 + 100,000 = 123,450$

→ IDS combines the advantages of breadth-first and depth-first search. It is the preferred blind search method in large state spaces with unknown solution depth.

→ Videos illustrating vs. depth-first search: `http://movingai.com/dfid.html`

Blind Search Strategies: Overview

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[a]	Yes[a,b]	No	No	Yes[a]	Yes[a,d]
Optimal?	Yes[c]	Yes	No	No	Yes[c]	Yes[c,d]
Time	$O(b^d)$	$O(b^{1+\lfloor g^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^l)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor g^*/\epsilon \rfloor})$	$O(bm)$	$O(bl)$	$O(bd)$	$O(b^{d/2})$

b	finite branching factor
d	goal depth
m	maximum depth of the search tree
l	depth limit
g^*	optimal solution cost
$\epsilon > 0$	minimal action cost

Footnotes:

[a] if b is finite

[b] if action costs $\geq \epsilon > 0$

[c] if action costs are unit

[d] if both directions use breadth-first search

Questionnaire

→ "Search tree": Tree generated by taking the initial state as the root, then keeping to expand states *without* duplicate elimination. (= The search space underlying any tree search.)

Question!

What is the size of the search tree in 8-Queens? (You may use a pocket calculator :-)

(A): 40320

(B): 371955

(C): 16777216

(D): 19173961

Question!

What about the 15-Puzzle?

Implementation: What Is a Search Node?

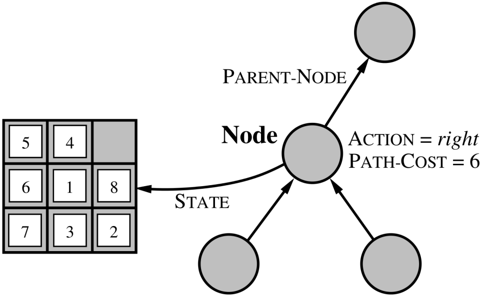
Data Structure for Every Search Node n

n.State: The state (from the state space) which the node contains.

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: $g(n)$, the cost of the path from the initial state to the node (as indicated by the parent pointers).



Implementation, ctd: Operations on Search Nodes

Operations on Search Nodes

Solution(n): Returns the path to node n. (By backchaining over the n.Parent pointers and collecting n.Action in each step.)

ChildNode(problem,n,a): Generates the node n' corresponding to the application of action a in state n.State. That is:
n'.State:=problem.ChildState(n.State, a);
n'.Parent:= n; n'.Action:= a;
n'.PathCost:= n.PathCost+problem.Cost(a).

Implementation, ctd: Operations for the Open List

Operations for the Open List

Empty?(frontier): Returns true iff there are no more elements in the open list.

Pop(frontier): Returns the first element of the open list, and removes that element from the list.

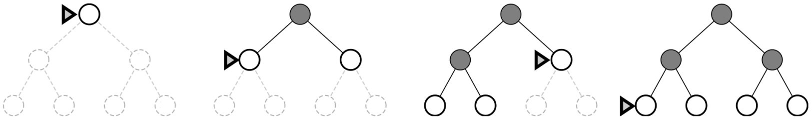
Insert(element, frontier): Inserts an element into the open list.

→ Crucial point: *Where* "Insert(element, frontier)" inserts the new element. Different implementations yield different search strategies.

Breadth-First Search: Illustration and Guarantees

Strategy: Expand nodes in the order they were produced (FIFO frontier).

Illustration:



Guarantees:

- Completeness: Yes.
- Optimality: Yes, for unit action costs. Breadth-first search always finds a shallowest goal state. If costs are not unit, this is not necessarily optimal.

Breadth-First Search: Pseudo-Code

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
     if EMPTY?(frontier) then return failure
     node ← POP(frontier)   /* chooses the shallowest node in frontier */
     add node.STATE to explored
     for each action in problem.ACTIONS(node.STATE) do
        child ← CHILD-NODE(problem, node, action)
        if child.STATE is not in explored or frontier then
           if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
           frontier ← INSERT(child, frontier)
```

- Duplicate check against explored set *and* frontier: No need to re-generate a state already in the (current) last layer.
- Goal test at node-generation time (as opposed to node-expansion time): We already know this is a shortest path so can just as well stop.

Breadth-First Search: Complexity

Time Complexity: Say that b is the maximal branching factor, and d is the goal depth (depth of shallowest goal state).

- Upper bound on the number of generated nodes: $b + b^2 + b^3 + \cdots + b^d$: In the worst case, the algorithm generates all nodes in the first d layers.
- So the time complexity is $O(b^d)$.
- And if we were to apply the goal test at node-expansion time, rather than node-generation time: $O(b^{d+1})$ because then we'd generate the first $d + 1$ layers in the worst case.

Space Complexity: Same as time complexity since all generated nodes are kept in memory.

Breadth-First Search: Example Data

Setting: $b = 10$; 10000 nodes/second; 1000 bytes/node.

Yields data: (inserting values into previous equations)

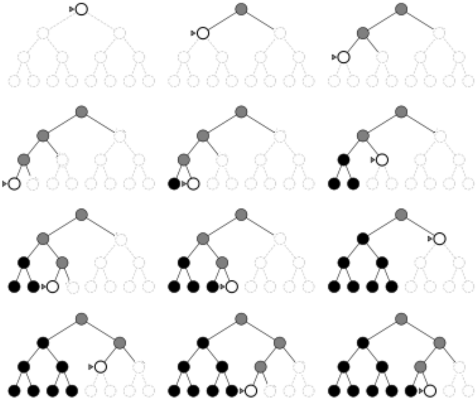
Depth	Nodes	Time		Memory	
2	110	.11	milliseconds	107	kilobytes
4	11110	11	milliseconds	10.6	megabytes
6	10^6	1.1	seconds	1	gigabyte
8	10^8	2	minutes	103	gigabytes
10	10^{10}	3	hours	10	terabytes
12	10^{12}	13	days	1	petabyte
14	10^{14}	3.5	years	99	petabytes

$\rightarrow$ The critical resource here is memory. (In my own experience, breadth-first search typically exhausts RAM within a few minutes.)

Depth-First Search: Illustration

Strategy: Expand the most recent nodes in (LIFO frontier).

Illustration: (Nodes at depth 3 are assumed to have no successors)



Depth-First Search: Pseudo-Code

Typically implemented as a recursive function: (Root call on a search node for the initial state of the problem)

function Recursive Depth-First Search(*n, problem*) **returns** a solution, or failure
  if *problem.GoalTest(n.State)* **then return** the empty action sequence
  for each *action a* **in** *problem.Actions(n.State)* **do**
    $n' \leftarrow$ *ChildNode(problem,n,a)*
    result $\leftarrow$ Recursive Depth-First Search(*n', problem*)
    if *result* $\neq$ *failure* **then return** $a \circ$ *result*
  return *failure*

$\rightarrow$ **Note:** Here (and everywhere else), as we loop across *problem.Actions(n.State)*, we generate that set (the actions applicable to the state) *only once* and store it: Finding the applicable actions typically consumes non-negligible runtime.

Depth-First Search: Guarantees and Complexity

Guarantees:

- Optimality: No. After all, the algorithm just "chooses some direction and hopes for the best". (Depth-first search is a way of "hoping to get lucky".)
- Completeness: No, because search branches may be infinitely long: No check for cycles along a branch!

  $\rightarrow$ Depth-first search is complete in case the state space is acyclic. If we do add a cycle check, it becomes complete.

Complexity:

- Space: Stores nodes and applicable actions on the path to the current node. So if m is the maximal depth reached, the complexity is $O(b\,m)$.
- Time: If there are paths of length m in the state space, $O(b^m)$ nodes can be generated. Even if there are solutions of depth 1!

  $\rightarrow$ If we happen to choose "the right direction" then we can find a length-l solution in time $O(b\,l)$ regardless how big the state space is.

Summary

- Classical search problems require to find a path of actions leading from an initial state to a goal state.
- They assume a single-agent, fully-observable, deterministic, static environment. Despite this, they are ubiquitous in practice.
- A problem can be described via its blackbox API, or declaratively, or explicitly. Each method allows to generate the problem's state space.
- For blackbox and declarative descriptions, the state space is exponentially larger than the size of the description, and deciding whether a solution exists is computationally hard (**NP** and beyond).
- Search strategies differ (amongst others) in the order in which they expand search nodes, and in the way they use duplicate elimination. Criteria for evaluating them are completeness, optimality, time complexity, and space complexity.
- Uniform-cost search is optimal and works like Dijkstra, but building the graph incrementally. Iterative deepening search uses linear space only and is often the preferred blind search algorithm.

Reading

- *Chapter 3: Solving Problems by Searching*, Sections 3.1 – 3.4 [Russell and Norvig (2010)].

  Content: Sections 3.1 and 3.2: A less formal account of what I cover here under "What (Exactly) Is a Problem?" and "How To Put the Problem Into the Computer?". Gives many complementary explanations, nice as additional background reading.

  Section 3.3: Pretty much the same I cover here under "Basic Concepts of Search", except for small changes to the general graph search procedure: I removed a bug, and made it more in line with what is typically used in practice. (Exercise: do you see the differences, and do you see what's the bug in RN?)

  Section 3.4: Pretty much the same I cover here under "Blind Search Strategies", except I left out bidirectional search, and adapted a few notations.

References I

Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (Third Edition)*. Prentice-Hall, Englewood Cliffs, NJ, 2010.