## A "Cornell AI Student" Agent

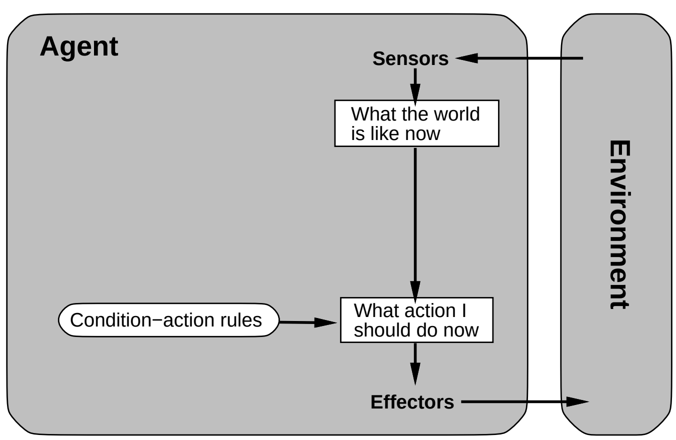
Search problem

**access to environment through sensors:** visual, aural, touch, etc.

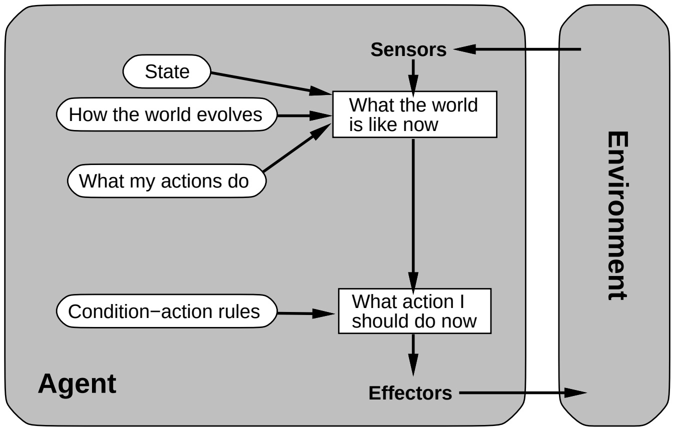
**available actions:** talk, walk, etc.



## A Simple Reflex Agent

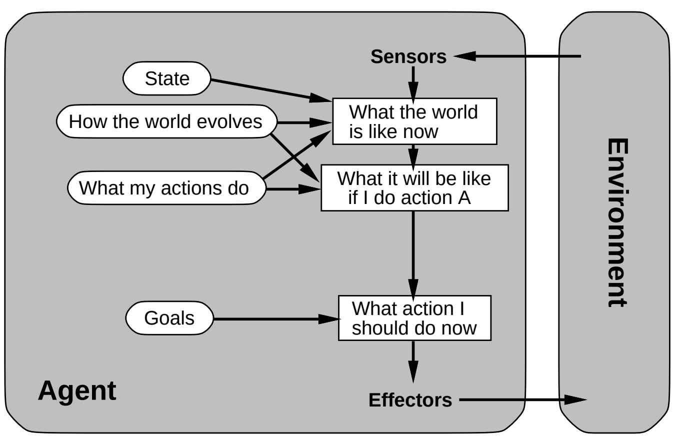


## Agents with Internal State




Slide CS472 – Problem-Solving as Search 3

## Goal-Based Agents




Slide CS472 – Problem-Solving as Search 4

## Human Problem Solving

**Search is a central topic in AI**

— Originated with Newell and Simon's work on problem solving. Famous book: "Human Problem Solving" (1972)

— Automated reasoning is a natural search task

— More recently: Given that almost all AI formalisms (planning, learning, etc.) are NP-Complete or worse, some from of search is generally **unavoidable (no "smarter" alg. available)**.



## Defining a Search Problem

**State space** – described by **an initial state** and the set of possible actions available (**operators**). A **path** is any sequence of actions that lead from one state to another.

**Goal test** – applicable to a single state to determine if it is the goal state.

**Path cost** – a function that assigns a cost to a path; relevant if more than one path leads to the goal, and we want the shortest path.

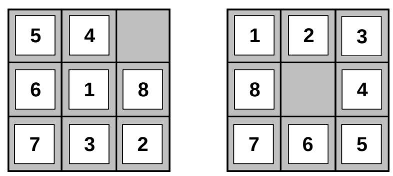
## The 8-Puzzle

**States:** Specifies the location of each of the eight tiles in one of the nine squares

**Operators:** blank moves left, right, up, down

**Goal test:** state matches the goal configuration

**Path cost:** each step costs 1, so path cost = length of path




Start State　　Goal State



## Cryptarithmetic

```
    SEND
 +  MORE
--------
   MONEY
```

Find substitution of digits for letters such
that the resulting sum is arithmetically correct.

Each letter must stand for a different digit.

## Cryptarithmetic, cont.

**States:** a (partial) assignment of digits to letters.

**Operators:** the act of assigning digits to letters.

**Goal test:** all letters have been assigned digits and sum is correct.

**Path cost:** zero. All solutions are equally valid.



### Solving a Search Problem: State Space Search

**Input:**

- Start state
- Goal state or goal test
- Operators

**Output:** legal sequence of states from initial state to goal state.

Search space is **not** stored in its entirety by the computer.

## Generic Search Algorithm

```
L = make-queue/stack(initial-state)
loop
      node = remove-front(L)
      if goal-test(node) = true return( node )
      S = successors(node, operators)
      insert(S,L)
until L is empty
return failure
```



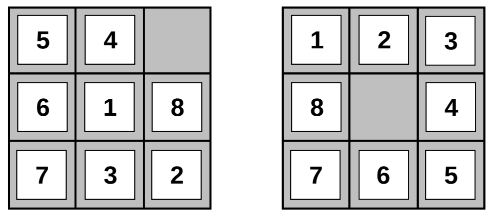
**Search procedure defines a search tree**

**root node** — initial state
**children of a node** — successors of the node
**fringe of tree** — L: nodes not yet expanded

**stack:** Depth-First Search (DFS).
**queue:** Breadth-First Search (DFS).

**Search strategy** — algorithm for deciding which leaf node to expand next.

## Solving the 8-Puzzle




Start State Goal State

What would the search tree look like after the start state was expanded?



## Evaluating a Search Strategy

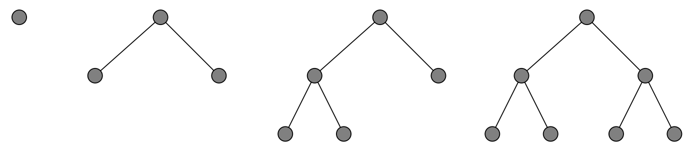
**Completeness:** is the strategy guaranteed to find a solution when there is one?

**Time Complexity:** how long does it take to find a solution?

**Space Complexity:** how much memory does it need?

**Optimality:** does the strategy find the highest-quality solution when there are several different solutions?

## Uninformed search: BFS



Consider paths of length 1, then of length 2, then of length 3, then of length 4,....



## Time and Memory Requirements for BFS – $O(b^d)$

Let b = branching factor, d = solution depth, then the maximum number of nodes expanded is: $1 + b + b^2 + ... + b^d$

| Depth | Nodes | Time | Memory |
|---|---|---|---|
| 0 | 1 | 1 millisecond | 100 bytes |
| 2 | 111 | .1 seconds | 11 kilobytes |
| 4 | 11,111 | 11 seconds | 1 megabyte |
| 6 | $10^6$ | 18 minutes | 111 megabytes |
| 8 | $10^8$ | 31 hours | 11 gigabytes |
| 10 | $10^{10}$ | 128 days | 1 terabyte |
| 12 | $10^{12}$ | 35 years | 111 terabytes |
| 14 | $10^{14}$ | 3500 years | 11,111 terabytes |

**BFS**

Memory is serious problem!

DFS a much better alternative.

Exponential time also a factor, but we'll see later on that a few more "tricks" enable us to effectively search huge state spaces.

E.g., chess: $10^{160}$ / planning: $10^{30}$.

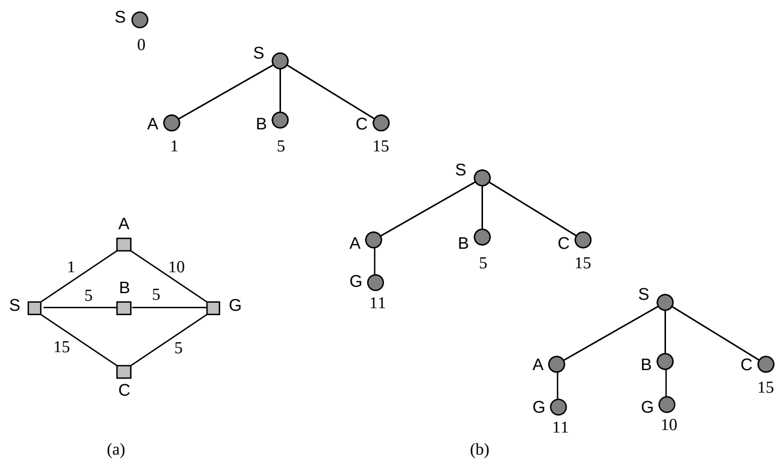


**Uniform-cost Search**

Use BFS, but always expand the lowest-cost node on the fringe as measured by path cost $g(n)$.

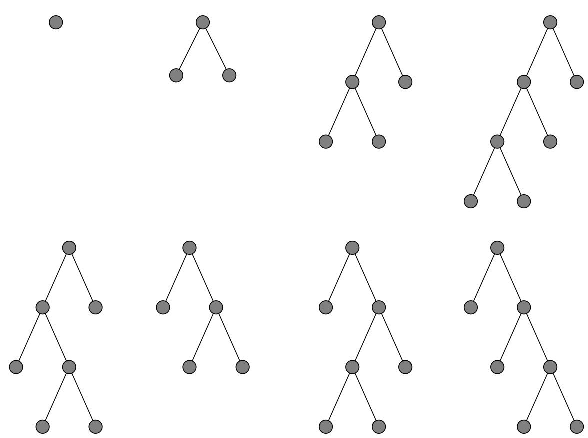
Requirement: $g(\text{Successor}(n)) \geq g(n)$

## Example




**Slide CS472 – Problem-Solving as Search 19**

## Uninformed search: DFS



**Slide CS472 – Problem-Solving as Search 20**

## DFS vs. BFS

| | Complete? | Optimal? | Time | Space |
|---|---|---|---|---|
| BFS | YES | "YES" | $b^d$ | $b^d$ |
| DFS | finite depth | NO | $b^m$ | $bm$ |

**Time**

$m = d$ — DFS typically wins

$m > d$ — BFS might win

$m$ **is infinite** — BFS probably will do better

**Space**

DFS almost always beats BFS



### Which search should I use?

Depends on the problem.

If there may be infinite paths, then depth-first is probably bad. If goal is at a known depth, then depth-first is good.

If there is a large (possibly infinite) branching factor, then breadth-first is probably bad.

(Could try **nondeterministic** search. Expand an open node at random.)

### Iterative Deepening [Korf 1985]

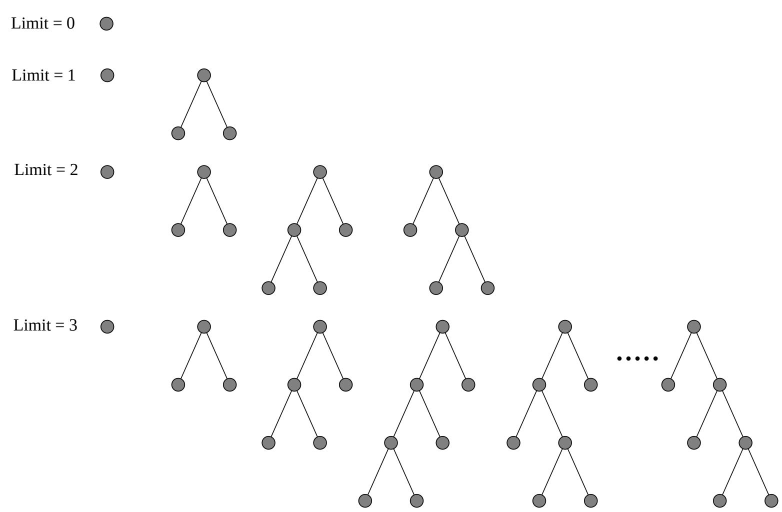
**Idea:**
Use an *artificial* depth cutoff, $c$.

If search to depth $c$ succeeds, we're done. If not, increase $c$ by 1 and start over.

Each iteration searches using DFS.



### Iterative Deepening



Space requirements? Same as DFS. Each search is just a DFS.

Time requirements. Would seem very expensive!! **BUT** not much different from single BFS or DFS to depth $d$.

**Reason:** Almost all work is in the final couple of layers. E.g., binary tree: 1/2 of the nodes are in the bottom layer. With b=10, 9/10th of the nodes in final layer!

So, repeated runs are on much smaller trees (i.e., exponentially smaller).



**Example:**
b=10, d=5, the number of nodes expanded in a DFS
1 + 10 + 100 + 1000 + 10,000 + 100,000 = 111,111
bottom level is expanded once, second to bottom twice...
total number of expansions:
$(d+1)1 + (d)b + (d-1)b^2 + ... + 2b^{d-1} + 1b^d =$
6 + 50 + 400 + 3,000 + 20,000 + 100,000 = 123,456
only about 11% more!

Ratio of ID to DFS: (b+1)/(b-1).
Cost of repeating the work at shallow depths is not prohibitive.

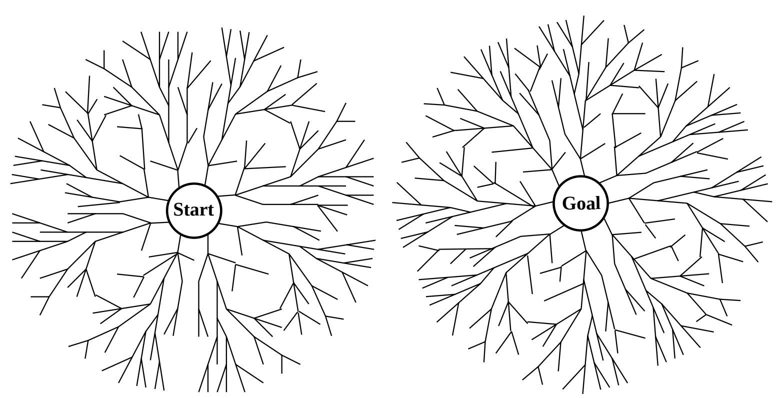
## Cost of Iterative Deepening

**space:** $O(bd)$ as in DFS, **time:** $O(b^d)$

| b | ratio of ID to DFS |
|---|---|
| 2 | 3 |
| 3 | 2 |
| 5 | 1.5 |
| 10 | 1.2 |
| 25 | 1.08 |
| 100 | 1.02 |



## Bidirectional Search



- Search forward from the start state and backward from the goal state simultaneously and stop when the two searches meet the middle.
- If branching factor = b from both directions, and solution exists at depth d, then need only $O(2b^{d/2}) = O(b^{d/2})$ steps.
- Example b = 10, d = 6 then BFS needs 1,111,111 nodes and bidirectional search needs only 2,222.
- Issues:
  - What does it mean to search backwards from a goal?



  - What if there is more than one goal state? (chess).

## Comparing Search Strategies

| Criterion | Breadth-First | Uniform-Cost | Depth-First | Depth-Limited | Iterative Deepening | Bidirectional (if applicable) |
|---|---|---|---|---|---|---|
| Time | $b^d$ | $b^d$ | $b^m$ | $b^l$ | $b^d$ | $b^{d/2}$ |
| Space | $b^d$ | $b^d$ | $bm$ | $bl$ | $bd$ | $b^{d/2}$ |
| Optimal? | Yes | Yes | No | No | Yes | Yes |
| Complete? | Yes | Yes | No | Yes, if $l \geq d$ | Yes | Yes |



## Cryptarithmetic

Consider search space for cryptarithmetic.
DFS (depth-first search)

Is this (DFS) how humans tackle the problem?

And if not, what do humans do?

**Human problem solving** appears much more **sophisticated!**

For example, we derive new constraints on the fly.

In a sense, we try to solve problems with **little** or **no** search!

In example, we can immediately derive that $M = 1$.
It then follows that $S = 8$ or $S = 9$. Etc. (derive more!)



Capturing such human problem solving strategies is surprisingly difficult. *For example, how do we know to first consider assigning $M$?*

Constraint programming techniques do provide some steps towards this kind of problem solving (next lecture).

Fortunately, computers are **very good at fast search!**
Search speed can **compensate** for lack of higher-level insights into the problem structure.

## Constraint Satisfaction Problems (CSP)

A powerful representation for (discrete) search problems.
Led to “constraint programming”.

A **Constraint Satisfaction Problem (CSP)** is defined by:

**X** is a set of n variables $X_1, X_2, \ldots, X_n$,
each defined by its finite domain $D_1, D_2, \ldots, D_n$.

**C** is a set of constraints $C_1, C_2, \ldots, C_m$.



## Constraints

A **constraint** $C_i$ restricts the set of possible values
that can be assigned to the variables in the constraint.
In other words, a constraint specifies which values are
compatible for the variables in the constraint.
A **solution** is an assignment of values to the variables
that satisfies all constraints.

## Send More Money as a CSP

**Variables:**

$S = \{0, \ldots, 9\}$; $E = \{0, \ldots, 9\}$;

$N = \{0, \ldots, 9\}$; $D = \{0, \ldots, 9\}$; $M = \{0, \ldots, 9\}$;

$O = \{0, \ldots, 9\}$; $R = \{0, \ldots, 9\}$; $Y = \{0, \ldots, 9\}$;

**Constraints:**

send $= 1000 \times S + 100 \times E + 10 \times N + D$;

more $= 1000 \times M + 100 \times O + 10 \times R + E$;

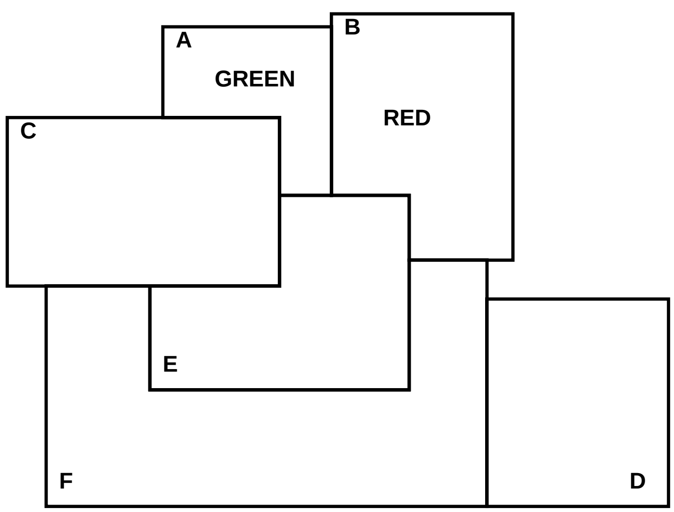
money $= 10000 \times M + 1000 \times O + 100 \times N + 10 \times E + Y$;

send + more = money;

each letter has a different digit (S $\neq$ E, S $\neq$ N, etc);



## Map-Coloring Problem



## Constraint Satisfaction Problems (CSP)

For a given CSP the problem is one of the following:

- find all solutions
- find one solution
  - just a feasible solution, or
  - a "reasonably good" feasible solution, or
  - the optimal solution given an objective
- determine if a solution exists



### How to View a CSP as a Search Problem?

**Initial State** – state in which all the variables are unassigned.

**Operators** – assign a value to a variable from a set of possible values.

**Goal test** – check if all the variables are assigned and all the constraints are satisfied.

**Branching Factor**

**Hypothesis 1** – any unassigned variable at a given state can be assigned a value by an operator: branching factor as high as sum of size of all domains.

**Better approach** – since order of variable assignment not relevant, consider as the successors of a node just the different values of a *single* unassigned variable: max branching factor = max size of domain.

**Maximum Depth of Search Tree**

$n$ the number of variables; all the solutions are at depth $n$. What are the implications in terms of using DFS vs. BFS?



## CSP – Goal Decomposed into Constraints

**How to exploit it?**

**Backtracking** only insert successors if *consistent* with constraints.

**Constraint propagation** "looking ahead" to remove inconsistencies.

**"Looking ahead"**

- **Forward Checking** — each time variable is instantiated, from domains of the uninstantiated variables all of those values that conflict with current variable assignments.
- **Arc Consistency** — state is arc-consistent, if every variable has some value that is consistent with each of its constraints (consider pairs of variables)
- **K-Consistency** generalizes arc-consistency. Consistency of groups of K variables.



- **Branching:**
  **Most-constrained variable heuristic:** choose the variable with the *fewest* possible values.

  **Most-constraining variable heuristic:** assigne a value to the variable that is involved in the largest number of constraints on other unassigned variables.

  **Least-constraining value heuristic:** choose a value that rules out the smallest number of values in variables connected to the current variable by constraints.

Dramatic recent progress in **Constraint Satisfaction.**

For example, we can now handle problems with **10,000** to **100,000** variables, and up to **1,000,000** constraints.